Dipartimento di Informatica e
Scienze dell'Informazione

An Active and Deductive Object-Oriented Data Model

by

Giovanna Guerrini

Theses Series DISI-TH-1998-2

DISI, Università di Genova

v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

An Active and Deductive Object-Oriented Data Model

by

Giovanna Guerrini

March, 1998

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Giovanna Guerrini
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
guerrini@disi.unige.it

Date of submission: February 1998

Title: An Active and Deductive Object-Oriented Data Model

Advisor: Prof. Elisa Bertino
Dipartimento di Scienze dell'Informazione
Università di Milano, Italy
bertino@dsi.unimi.it

Supervisor: Prof. Maurizio Martelli
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
martelli@disi.unige.it

Ext. Reviewers:
Stefano Ceri
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
ceri@elet.polimi.it

Susan Urban
Department of Computer Science
Arizona State University, USA
s.urban@asu.edu

Abstract

This thesis investigates the issues arising from the introduction of active and deductive rules in object-oriented data models. This research topic is at the confluence of three among the most promising research areas in the database field, namely object-oriented databases, active databases and deductive databases. These three research areas, though moving along different directions, try to overcome the inadequacy of traditional data models and database languages (in particular, of the relational ones) with respect to advanced applications.

Object-oriented database systems overcome the limitations of relational ones with respect to the emerging data-intensive applications due to their ability of directly representing in the model complex objects and storing in the database not only data but also the operations that can be performed on data. Deductive rules enhance a database system with inferential capabilities, providing the support for derived data. They allow to represent in a database not only data, but also *knowledge*, making it possible to infer additional information from the stored data and providing also the capability of expressing recursive data. Deductive rules can moreover be used for restructuring data, so that applications can customize shared data, through the definition of views. An active database system is a database system in which operations are automatically executed when specified events happen and particular conditions are met. Thus, active rules enhance traditional database systems with reactive capabilities, providing powerful and efficient mechanisms for database integrity constraints, authorization, statistic gathering, monitoring and alerting.

This thesis investigates how deductive and active rules can be effectively integrated in an object-oriented data model. In particular, in the thesis we formally define an object-oriented data model, its rule-based query and method definition languages, its active rule language for specifying triggers, and its view mechanism. The semantics of the method definition and active rule languages are specified, and issues related to typing and method dispatching are also discussed.

Acknowledgements

To my supervisor, Elisa Bertino, who was able to communicate me her love for research, for her continuous help and support and for all the opportunities she gave me during these years.

To all those provided me helpful comments on the thesis, among them, let me mention Stefano Ceri, Susan Urban, Elena Zucca, Maurizio Martelli.

Part of the work reported in this thesis has been developed in cooperation with other researchers. An acknowledgement is due to all of them: Renè Bal, Isabella Merlo, Jesus Garcia-Molina, Barbara Catania, and Luca Rusca, since from each of them I learnt something. During this years of PhD research, I also enjoyed working with Elena Ferrari. Though the work developed with her is not part of this thesis, many of her ideas and suggestions influenced the thesis.

This thesis originates from my work in the IDEA Project. Thanks are due to Stefano Ceri and to all the IDEA group at Politecnico di Milano, for their kindness and willingness. Thanks to Stefano are also due for having supported me in the period before I started my PhD and for having taught me that "the true researcher never gives up", when climbing a mountain under a hurricane, in Leysin, Switzerland, back in '94.

To all my PhD colleagues in Genova, for all what we have shared, and since it had been very pleasant working here. To Giorgio, who very often remembered me that I had to write a thesis, in the last few months, and for all the rest ... To Davide, Michela, Sandro, Luigi, Paola, Floriano, Gianna and Isabella (again!). To Gianni, and the other "vecchietti", since the times of 208bis probably were the nicest of my PhD.

To all my friends. To Monica, who never denied me a bed in Milano, for the lots of things we had been through together. To Luca, Serena, Riccardo, Monica, Maurizio, Miriam, Roberto, Marina, Marco, Barbara. To Fatima and Denise, who hosted me when I really felt homeless, and to Fabrizio, Enrico, Gianni and all the other "fantastic friends".

To the Guerrini's (grande famiglia) for always caring about me. In particular, to mum and dad, who started all this ... To Bruno, for his patience and support during these years. To Aldina and Manlio, for their help and love. Finally, to Mara, since nothing would make sense without her, and ... to beg her pardon for all the time this thesis has stolen her.

Table of Contents

Introduction

The area of data models has always been a very active research area that has a considerable impact on data management technologies. Indeed, the evolution of data models has marked the various generations of database management systems (DBMS). In particular, since the definition of the relational model, research has progressed to develop "new-generation" data models. A number of modeling requirements can be devised for those new models, that include the possibility of defining and manipulating complex objects, user-defined operations, type hierarchies, mechanisms for supporting views and integrity constraints, inferential and reasoning capabilities, reactive capabilities. A number of research and development directions in the data model area can be identified, that try to address the above requirements; some of those directions are based on the integration of the database technology with programming language and artificial intelligence technologies. To date, three main directions can be devised in the area of data models: object-oriented data models, integrating database technologies with object-oriented programming; deductive data models, integrating database technologies with logic programming; active data models, resulting from the introduction in the database field of the production rule paradigm developed in artificial intelligence. Object-oriented data models [42] are establishing themselves as the most promising among those directions, mainly because of industrial interest that has resulted in the development of a large number of object-oriented database systems and prototypes. Moreover, several significant application have been developed on object-oriented DBMS (OODBMS), from areas such as geographical information systems, telecommunications, multimedia systems, spatial databases.

Object-oriented data models, however, still lack inferential and reasoning capabilities, and reactive capabilities. Thus, there is a growing interest in enriching the object-oriented data model with active and/or deductive rules. The use of rule-based languages in database systems has a number of advantages: these languages are easy to use and to understand, indeed, they are high-level languages if compared to traditional programming and database languages. Moreover, they have proved their effectiveness in extending *relational-based* languages with inferential and reactive capabilities. Therefore, an active, deductive, object-oriented data model seems to meet all the requirements of next generation database systems.

Deductive rules [62] are the usual rules of logic programming, exploited in the database field to define derived or intensional data, that is, data not stored in the database. Indeed, those rules allow to derive new information from the facts explicitly stored in the database and allows to express in a convenient way recursively defined data. A deductive rule has the form

$$template \leftarrow query$$

where *template* specifies the structure of the derived data, and *query* is the expression that computes the derived data. Deductive rules can be added to object-oriented databases for different purposes. First of all, the deductive rule language (which usually is a subset of a logic programming language) can be used as method definition language and as query language, bringing advantages of declarativity and uniformity. Moreover, rules can be used to express the population of derived classes and to define derived attribute values. In addition, deductive rules can be used to define inter-class relationships (views).

Active databases [65] results from the introduction in the database field of the notion of active rule (or *production rule* or *trigger*) from artificial intelligence. An active rule is a syntactical construct to define the reaction of the system. Active rules are usually specified by means of the *event-condition-action* (ECA) paradigm, that is, they have the form

ON event IF condition DO action

Events are monitored and their occurrence activates (triggers) the rule, a condition is a declarative formula that must be satisfied in order for the action to be executed; an action is a sequence of database operations (possibly causing state changes). An active database system is able to autonomously perform certain operations in response to certain events occurring or certain conditions being satisfied. By contrast, in conventional database systems (passive ones) data is created, retrieved, modified and deleted only in response to explicit invocations by users or application programs. Active database systems are significantly more powerful than their passive counterparts: they can efficiently perform functions that in passive database systems must be encoded in applications, and they facilitate applications that are beyond the scope of passive database systems. Active rules can be used for integrity constraint monitoring and enforcement, for situation monitoring and alerting, for the specification of consequential actions or user notification upon certain situations.

The integration of rules in an object-oriented data model introduces several interesting issues. The introduction of rules in a rich data model, such as the object-oriented one, requires understanding how rules interact with object-oriented modeling constructs, such as inheritance, and how methods and rules can be combined. In this thesis we formally define an active and deductive object-oriented data model, with views and integrity constraints.

A first contribution of the thesis is thus the formal definition of a reference object-oriented data model, which constitutes a formal basis on which issues related to rules are addressed. Formal definitions exist only for *toy* data models, not covering all the features of the models of real object-oriented databases. Our reference model is very rich, in that it covers a number of modeling features not covered by other models. Among these features let us mention the distinction between values and objects, the support for named and constrained value types as well as for explicitly populated value collections, the presence of class attributes and class methods, the support for composite objects and referential integrity, the possibility of defining attributes intensionally. Another important characteristic of our reference model is the support for object migration, that is, the possibility for an object to change its class membership.

A peculiarity of our reference object model is that objects can belong to several most specific classes, that is, an object can be instance of multiple classes[1]. There are different motivations for this. The most intuitive one is that in real world objects do belong to several most specific classes; in a model with multiple inheritance this can be handled by defining a subclass of all the involved classes, this solution, however, leads to a lot of artificial subclasses, sometimes referred to as *intersection classes*. Moreover, multiple class direct membership must be handled in models supporting derived classes or views.

The defined model exploits the deductive rule language as query language, and the active rule language as method definition language, where the event activating the execution of the trigger implementing a method is the call of the corresponding method. In the thesis we specify both the syntax and the semantics of the rule-based set-oriented method definition language. The basic idea is to express methods as *condition* $\rightarrow$ *action* constructs, where the condition is a declarative formula and the action is a sequence of update operations and method calls. Methods with an empty action part are side-effect free and can be invoked in conditions of other methods (and in any formula of our deductive language). The most relevant aspect of such a rule-based language is its set-orientedness, since the action is executed on the set of bindings satisfying the condition. This implies the possibility of exploiting efficient query evaluation techniques when executing methods. To the best of our knowledge, the only work dealing with set-oriented methods in object-oriented databases is [16]. An important advantage of methods expressed through rule-based languages is that method behavior can be clearly characterized in terms of the set of updates performed, this allows us to investigate issues such as behavior preserving for method overriding.

Typing aspects are considered in the context of the developed object-oriented data model

[1] According to the usual terminology, an object is an *instance* (or a *direct member*) of a class if that class is the most specific one, in the inheritance hierarchy, to which the object belongs. Whenever an object o is an instance of a class c then o is also a *member* of all the superclasses of c.

and method definition language. This means considering restrictions on the redefinition of inherited features, by giving a precise characterization of type refinement, and choosing of an attitude towards type checking (static or dynamic). A type system for rules has also been defined. This type system is exploited to check the type correctness of queries and operations. We remark that typing issues in object-oriented rule languages have not been much investigated (an exception is the DTL query language [20]).

A third contribution of this thesis is the definition of a semantics for the active component of the model. An important neglected aspect in active object-oriented databases concerns trigger semantics. Little work has been carried on towards the specification of a semantic framework for triggers. This, and the fact that the execution model varies from system to system, constitute a serious obstacle to the diffusion of active database systems. Moreover, the work carried on towards the specification of a semantics for active rule language has been developed in the context of the relational data model, whereas semantic foundations for active object-oriented data models are missing. Our model supports trigger inheritance and overriding. Thus another contribution of the thesis is to clarify a neglected issue in existing active object-oriented data models, namely trigger inheritance and overriding. Existing systems provide only simplistic solutions with respect to trigger inheritance and only few systems support trigger overriding. In our opinion, there are several cases in which rule overriding is useful. However, we believe that rules should not be overridden in a uncontrolled way, rather they should only be *refined* in subclasses, that is, they can be overridden provided that (*i*) the redefined rule is triggered each time the overridden one would do; (*ii*) the redefined rule does at least what the overridden one would do. Syntactical condition ensuring that are defined in the thesis.

The model also includes the possibility of defining views. A fourth contribution of the thesis is thus the development of a view mechanism for object-oriented databases well-suited to all view applications, namely external schemas, authorization and schema evolution. The existing view models are tailored to specific applications; this has a strong influence on the view model. Views in our model are virtual classes, that can be organized in virtual schemas (referred to as schema views), providing different viewpoints on the object database.

The last topic dealt with in the thesis is method dispatching. When objects belong to several most specific classes, conflicts among different definitions may arise. If an object has several most specific classes, the object takes the union of the features of all the classes to which it belongs. Conflicts among the definitions of features in different classes may however arise. Such conflicts resemble conflicts due to multiple inheritance. However, conflicts due to multiple inheritance can be detected at compile time whereas an object may become and cease to be an instance of a class at run time; the situation is thus more complicate for multiple class direct membership. If an object can belong to more than one class, there are additional problems in determining *the* class of an object which implies the method implementation to be executed. There is a fundamental choice to be taken for solving these conflicts: whether or not taking care of the context from which the feature is referred. Of course, a context-dependent approach is more suitable for view support. Different dispatching techniques for objects with multiple most specific classes are analyzed, and compared with respect to a number of dimensions, among which we would like to mention the efficiency of dispatching and whether or not the technique allows for static type checking.

The data model we formally define originates from the Chimera active, deductive, object-oriented data model[2] [63, 64], which has been developed in the context of the ESPRIT Project 6333 IDEA. The model we consider, however, extends Chimera with direct support for referential integrity and composite objects, with views, and with trigger overriding. In the thesis, moreover, besides formally defining the semantics of the model, we also deal with issues which have not been previously investigated for Chimera, such as multiple class direct membership, resolution of name conflicts due to multiple class direct membership, method dispatching, trigger inheritance, typing aspects.

The thesis is organized as follows.

Chapter 1 presents the basic notions of object-oriented, active and deductive data models, discussing and comparing existing approaches to the introduction of rules in object-oriented data models, and relating the model we define in the thesis with related work.

Chapter 2 introduces the reference object-oriented data model. It formally introduces the notions of values and objects, types and classes, inheritance, schema and database.

Chapter 3 is devoted to the method definition language. It introduces the syntax of the language, presents typing rules, and specifies the semantics of the language. It finally discusses some semantic properties of methods, such as method refinement in subclasses, in the context of the defined language.

Chapter 4 deals with the reactive component of the language. It specifies the syntax and the semantics of the trigger language, and discusses the problem of trigger overriding.

Chapter 5 is devoted to view support in our model. It presents a view definition language and shows how views can be integrated in the object-oriented data model.

Chapter 6 discusses method dispatching in object-oriented data models supporting multiple class direct membership. Two different dispatching techniques are proposed and compared under a number of dimensions.

Finally, some concluding remarks and possible topics of future works are presented in the conclusions.

[2] A Chimera is monster of Greek mythology with a lion's head, a goat's body, and a serpent's tail; each of them represents one of the three components of the language.

For ease of reference, Appendix A summarizes the symbols most frequently used in the thesis.

Chapter 1

Active and Deductive Rules in Object-Oriented Data Models

In this chapter we survey proposed approaches to the integration of rules in object-oriented data models. First, we introduce the basic notions of object-oriented databases and discuss the features of recently proposed object-oriented data models. Then, we discuss the efforts to incorporate deductive and active rules in object-oriented data models. The chapter is organized as follows. Section 1.1 is devoted to object-oriented data models, Section 1.2 deals with active rules and active OODBMSs, whereas Section 1.3 presents deductive object databases. Finally, Section 1.4 concludes the chapter discussing how the model we will present in the thesis relates to already developed work in the areas of active and deductive object data models.

1.1 Object-Oriented Data Models

Traditional database management systems (DBMSs), based on the relational data model, do not successfully handle data typically managed by design and manufacturing systems (CAD/CAM, CIM), scientific and medical databases, geographic information systems, multimedia databases. Those classes of applications have requirements and characteristics different from those typical of traditional database applications for business and administration. In particular, they are characterized by highly structured data, by long transactions, by the need for new data types to store images and texts, and by non-standard operations peculiar to the application. Object-oriented databases [33, 42, 58, 118] aim at fulfilling those requirements. The object-oriented approach provides the required flexibility not being constrained by the data types and query languages available in traditional database systems. One of the most remarkable features of object-oriented databases is the possibility they give to the programmer of specifying both the structure of complex objects and the operations that can be executed on them.

Object-oriented databases result from the integration of database technology with the object-oriented paradigm developed in the programming languages and software engineering areas. The basic principle of the object-oriented approach in programming is to consider the program as composed by independent objects, grouped in classes, communicating among each other through messages. The same concepts, however, have been introduced also in other computer science areas, like knowledge representation languages, and they have often been interpreted in different ways.

In this section, we first introduce the basic concepts of object-oriented data models, then we focus on some advanced aspects, namely composite objects and referential integrity, multiple class direct membership and typing aspects. Finally, we compare the data model of best-known OODBMSs with respect to a number of dimensions.

1.1.1 Basic Concepts

In what follows, we will introduce the main concepts of object-oriented data models.

1.1.1.1 Objects

An object-oriented database is a collection of objects. In an object-oriented database each real-world entity is represented by an object, having a state and a behavior. The state consists of the values for object attributes; the behavior is specified by the methods that act on the object state, invoked by the corresponding operations. One of the most important properties of an object is that of having an identity, different from that of any other object, and immutable during the object lifetime.

Many object-oriented database systems actually do not require each entity to be represented as an object, rather they distinguish between objects and values. The differences between values and objects are the following [25]:

- values are universally known abstractions, and they have the same meaning for each user; objects, by contrast, correspond to abstractions whose meaning is specified in the context of the application;
- values are *built-in* in the system and do not need to be defined; objects, by contrast, must be introduced in the system through a definition;
- the information represented by a value is the value itself, whereas the meaningful information represented by an object is given by the relationships it has with other

objects and values; values are therefore used to describe other entities, whereas objects are the entities being described.

Thus, values are elements of built-in domains, whereas objects are elements of uninterpreted domains. Typical examples of values are: integers, reals, strings. To each object an immutable identifier is assigned, whereas a value has no identifier, rather it is identified by itself.

Object Identity Each object is uniquely identified by an object identifier (OID), providing it with an identity independent from its value. The OID is unique in the system, and it is immutable, that is, it does not depend on the state of the object. Object identifiers are usually not directly visible and accessible by the database users, rather they are used internally to the system for identifying objects and for allowing an object to reference another object as the value of one of its attributes. This supports the specification of interconnected objects, and let objects share a component.

The notion of object identifier is quite different from the notion of *key* employed in the relational model to identify a tuple of a relation. A key is defined as the value of one or more attributes, and it can be modified, whereas an OID is independent from the value of an object state. In particular, two different objects have different OIDs even when their attributes have the same values. Moreover, a key is unique with respect to a relation, whereas an OID is unique in the entire database. The use of OIDs has a number of advantages with respect to the use of keys as an identification mechanism. First of all, being OIDs implemented by the system, the application programmer does not need to select the appropriate keys for various sets of objects. Moreover, being OIDs implemented at a low level in the system, better performances are achieved. A disadvantage in the use of OIDs with respect to keys could be the fact that no semantic meaning is associated with them. Note, however, that very often in relational systems, for efficiency reasons, users employ semantically meaningless codes as keys, especially when foreign keys need to be used.

The notion of object identity introduces at least two different notions of object equality:

- **equality by identity**: two objects are *identical* if they are the same object, that is, if they have the same identifier;
- **equality by value**: two objects are *equal* if the values for their attributes are recursively *equal*.

Obviously, two identical objects are also equal, whereas the converse does not hold. Some object-oriented data models also provide a third kind of equality, known as *shallow value* *equality* by which two objects are equal, though not being identical, if they share all attributes.

Object Structure In an object-oriented database the value associated with an object (that is, its state) is a complex value which can be built starting from other objects and values, using some type constructors. Complex (or structured) values are obtained by applying those constructors to simpler objects and values. Examples of primitive values are integers, chars, strings, booleans and reals. The minimal set of constructors that a system should provide include sets, lists, and tuples. In particular, sets are crucial since they are a natural way to represent real world collections and since they allow multivalued attributes to be defined. The tuple constructor is important since it provides a natural way to represent the properties of an entity; lists and arrays are similar to sets, but they impose an order on the elements of the collection and they are needed in many scientific applications. Those constructors can be nested in an arbitrary way. A complex value can contain as components (references to) objects.

Object-oriented databases provide thus an *extensible type system*, that allows the user to define new types, modeling the types best suited to the data that will be handled by the application. The types provided by the system and those defined by the user can be used exactly in the same way.

Methods Objects in an object-oriented databases are manipulated through methods. A method definition usually consists of two components: a *signature* and an *implementation*. The *signature* specifies the method name, names and types of method arguments, and the type of the result, for methods returning a result value. Thus, the signature is a specification of the operation implemented by the method. Some object-oriented systems do not require the specification of argument types; however, that specification is required in systems performing static type checking. The method implementation consists of a set of instructions expressed in a programming language. Various OODBMSs exploit different languages. For instance, Orion exploits Lisp, GemStone a Smalltalk extension, namely OPAL, O_2 a C extension, namely CO_2; other systems, among which ObjectStore and Ode, exploit C++.

The use of a *general purpose* computationally complete programming language to code methods allows the whole application to be expressed in terms of objects. There is thus no more the need, typical of relational DBMSs, of embedding the query language (e.g. SQL) in a programming language.

Encapsulation In a relational database system, queries and application programs acting on relations are usually expressed in an imperative language incorporating statements of the data manipulation language (DML), and are stored in a traditional file system rather than in the database. In that approach, therefore, there is a sharp distinction between programs and data, and between query language and programming language. In an object-oriented database data and operations manipulating them are *encapsulated* in a single structure: the object. Data and operations over them are thus designed together and are stored in the same system. Encapsulation thus provides a sort of "logical data independence", allowing modifications on types without need of modifying applications using them.

The notion of encapsulation in programming languages derive from abstract data types. In this view, an object consists of an interface and an implementation. The interface is the specification of the operations that can be executed on the object; these operations are the only part of the object which is visible from outside. Implementation, by contrast, contains data, that is, the representation or state of the object, and methods, describing, in some programming language, the implementation of each operation. This principle, in the database context, is reflected in the fact that an object contains both programs and data, with a variation: in the database context it is not clear whether or not the structure defining the type of an object is part of the interface. In the programming language context, the data structure is usually part of the implementation, and, thus, not visible. For example, in a programming language the data type `list` should be independent from the fact that lists are implemented as arrays or as dynamic structures, thus this information is correctly hidden. By contrast, in the database context, the knowledge of a class attributes, and references made through them to other classes, can be useful.

Some OODBMSs, like Orion, allow to read and write the object attribute values, thus violating encapsulation. The reason for that is to simplify the development of applications that simply access and modify object attributes. Those applications are obviously very common in the database context. Strict encapsulation would require writing many trivial methods. Other systems, like O_2, allow to specify which methods and attributes are visible in the object interface. Those attributes and methods are said *public*, whereas those that cannot be seen from outside the object are said *private*. Finally, some other systems, including GemStone, by contrast, force strict encapsulation.

1.1.1.2 Classes

Instantiation is the mechanism offering the possibility of exploiting the same definition to generate objects with the same structure and behavior. Object-oriented languages provide the notion of class as a basis for instantiation. In this respect a class acts as a *template*, by specifying:

- a structure, that is, the set of instance attributes;
- a set of operations, defining the instance interface;

- a set of methods, implementing operations.

Given a class, through a *new* operation, objects answering to all messages defined for the class can be generated. Obviously, the attribute values must be stored separately for each object. However, there is no need to replicate message and method definition, which are associated with the class.

There are, moreover, some class features that cannot be seen as attributes of its instances, such as the number of class instances present in each moment in the database or the average value of an attribute. An example of operation which is invoked on classes rather than on objects is the *new* operation for creating new objects. Some object-oriented data models, like those of GemStone and Orion, allow the definition of attributes and methods characterizing the class as an object, which are, thus, not inherited by the class instances.

In almost all object-oriented data models, each attribute has a *domain*, specifying the class of possible objects that can be assigned as values to the attribute. If an attribute of a class c has a class c' as domain, each c instance takes as value for the attribute an instance of c', or of a subclass of its. Moreover, an **aggregation** relationship is established between the two classes. An aggregation relationship between classes c and c' specifies that c is defined in terms of c'. Since c' can be in turn defined in terms of other classes, the set of classes on the schema is organized in a *aggregation hierarchy*. Actually, it is not a hierarchy in a strict sense, since classes can be recursively defined.

Extent and Persistence Mechanisms Besides being a *template* for defining objects, in some systems the class also denotes the collection of its instances, that is, a notion of **extent** is related to the class. This aspect is important since the class is the basis on which queries can be formulated. Queries are meaningful only when they are applied to object collections. In systems in which classes have no extensional functionality, queries are applied to sets, and different sets can contain instances of a same class. Automatically associating an extent with each class (like in the Orion system) has the advantage of a simpler use; by contrast, systems (like O_2 and GemStone) in which classes define only a specification and an implementation of objects, while queries are issued against *collections* handled by the user, provide a greater flexibility.

An important issue concerns the *persistence* of class instances, that is, how objects are made persistent (that is, inserted in the database) and are eventually deleted (that is, removed from the database). In relational databases, explicit statements (like `INSERT` and `DELETE` in SQL) are provided to insert and delete data from the database. In object-oriented databases two different approaches can be adopted:

- Persistence is an implicit property of all the class instances; the creation (through the *new* operation) of an instance has also the effect of inserting the instance in the database, thus the creation of an instance automatically implies its persistence. This approach is usually adopted in systems in which classes also have an extensional functionality.

- Persistence is an orthogonal property of objects; the creation of an instance does not have the effect of inserting the instance in the database. Rather, if one wants an instance to survive to the program that created it, he must make it explicitly persistent, for example, by assigning it a name, or by inserting it in a persistent collection of objects. This approach is usually adopted in systems in which classes have no extensional functionality.

For what concerns object *deletion*, two different approaches are possible:

- The system provides an explicit delete operation. The possibility of explicitly deleting objects poses the problem of referential integrity; if an object is deleted and there are other objects referring to it, references are not valid any longer (they are known as *dangling pointers*). The explicit deletion approach is adopted by the Orion and Iris systems.

- The system does not provide an explicit delete operation. A persistent object is deleted only if all references to it have been removed (a periodic *garbage collection* is performed). This approach, adopted by the GemStone and O_2 systems, ensures referential integrity.

Migration Since objects represent real-world entities, they must be able to reflect the evolution in time of those entities. A typical example is that of a person which is first a student, then an employee, then he retired. This situation can be modeled only if an object can become instance of a class different from the one from which it has been created. This evolution (known as object *migration*) allows an object to modify its features, that is, attributes and operations, and still retain its identity [40]. Object migration among classes introduces, however, semantic integrity problems. If, indeed, the value for an attribute a of an object o is another object o', instance of the class domain of a, and o' changes class, if the new class of o' is no more compatible with the class domain of a, this will result in o containing an illegal value for a. For this reason migration is currently not supported by most existing systems.

Classes and Types Some object-oriented systems support, in addition to the notion of class, the notion of *type*. The notions of type and class are often used in an interchangeable way; however some systems support both these notions. The distinction between those two concepts is not clear; however some differences are quite accepted. The notion of type comes from the programming language context. This notion is used to refer to the specification of the interface of a set of objects. A type is thus a set of objects exhibiting the same externally observable behavior. This notion is mainly used at compile time to check program correctness. A class, by contrast, provides the implementation for a set of objects of the same type and it provides operations for creating objects. A class is thus a set of objects with exactly the same internal structure, that is, the same attributes and the same methods.

In an object-oriented database the functionalities required to the system with respect to a class are those of interface, implementation and extension handling. The interface functionality is thus assigned to the type definition, whereas a class extends the type definition with information on names and domains of attributes and methods, that is, it represents the implementation of a type. In a sense, the class definition encloses the corresponding type definition. Another distinction which is often made between types and classes in object-oriented databases, is that a type has mainly a specification functionality, whereas the class has a more dynamic functionality, related to its extent. Indeed, while a type specifies the common features of a set of objects, a class is characterized by the two fundamental aspects of *object factory*, that is, the possibility of creating new objects, and of *object warehouse* or *object repository*, that is, the capability of having an extent, i.e. a set of instances.

To conclude, we remark that in principle an object-oriented data model should provide three different notions, namely:

- the notion of type, with the meaning of specification of a set of objects;
- the notion of class, with the meaning of structure and implementation of a set of objects;
- the notion of object collection, to model type extents.

However, this would have heavy consequences on other aspects of the model and on implementation; for example, three different inheritance hierarchies should be defined, and with each object both its type and its class should be associated.

Note, finally, that in most OODBMSs there is not a clear distinction among the three notions, and in most cases there is a one-to-one relationship between types and classes.

1.1.1.3 Inheritance

Inheritance allows a class, said *subclass*, to be defined starting from the definition of another class, said *superclass*. The subclass inherits attributes, operations and methods of its superclass; a subclass may have in addition some specific, non inherited features. Inheritance is a powerful reuse mechanism. By using that mechanism, indeed, when defining two classes their common properties, if any, are identified; those common properties will be included in a superclass of theirs. The definitions of the two classes will by contrast specify only their distinguishing specific properties. This approach not only reduces the amount of code to be written, but it has also the advantage of giving a more precise and concise description of the world being modeled.

Some systems allow a class to have several direct superclasses, in this case we talk of *multiple inheritance*, whereas other systems impose the restriction to a single superclass, *single inheritance*. The possibility of defining a class starting from other classes simplifies the task of class definition. However, in case of multiple inheritance, some conflicts may arise. To solve those conflicts some different strategies can be adopted:

- an ordering is imposed on superclasses, and conflicting features are inherited from the superclass which precedes the others in the ordering;
- an explicit qualification mechanism is exploited, that is, the user explicitly specifies from which superclass each conflicting feature is inherited.

In literature and in various object-oriented languages there are quite different inheritance notions. In the knowledge representation context, for instance, inheritance has a quite different meaning from the one it has in object-oriented programming languages. In the former context, indeed, a subclass defines a *specialization* with respect to features and behaviors of the superclass, whereas in the latter the emphasis is on attribute and method reuse. Different inheritance notions can then be considered, corresponding to which three different hierarchies can be distinguished:

- *subtype hierarchy*: expressing the consistency among type specifications, by specifying subtype relationships allowing the substitutability of a subtype instance in each context where a supertype instance is expected [55];
- *implementation hierarchy*: allowing code sharing among classes;
- *classification hierarchy*: expressing inclusion relationships among object collections.

Each hierarchy refers to different properties of the type/class system; those hierarchies are however generally merged in a single inheritance mechanism.

Overriding, Overloading and Late Binding The notion of *overloading* refers to the possibility of adopting the same name for different operations. This possibility is extremely useful in the object-oriented context. Consider as an example [19] a `display` operation receiving as input an object and displaying it. Depending on the object type, different display mechanism are exploited: if the object is a figure, it should appear on video; if the object is a person, its data should be printed in some way; if the object is a graph, a graphical representation of it should be produced. Another problem arises for displaying a set of objects the type of whose members is not known at compile-time.

In an application developed on a conventional system, three different operations `display_graph`, `display_person` and `display_figure` would be defined. This forces the programmer to know all possible object types, to know all associated display operations and to use them properly. For example:

```
for x in X do
    begin
      case of type(x)
         person:  display_person(x);
         figure:  display_figure(x);
         graph:  display_graph(x);
      end;
    end;
```

In an object-oriented system, by contrast, the `display` operation is defined for the more general type in the system. Thus, the operation has a single name and can be used indifferently on various objects. The operation implementation is *redefined* for each type; this redefinition is known as *overriding*. As a result, a single name denotes different programs and the system takes care of selecting the appropriate one at each time during execution. Thus the code shown above is compacted as:

```
for x in X do display(x)
```

This provides several advantages. Class implementors write the same number of methods, but the application designer does no more need to take care of that; the resulting code is simpler and easier to maintain, since the introduction of a new type does not require to modify applications. In a following moment, objects of another type, say information on some products, could be displayed, indeed, by simply defining a class `product`, providing a proper (re)definition of the `display` operation.

To support this functionality, however, the system can no more bind operation names to the corresponding code at compile time, rather it must do that at run-time: this late translation is known as *late binding*.

Thus, the notion of *overriding* refers to the possibility for a class of redefining attributes and methods it inherits from its superclasses; the inheritance mechanism allows thus to specialize a class through additions and substitutions. Overriding implies *overloading*, since an operation shared along a class hierarchy can be implemented in different ways in those classes; therefore, the same operation name denotes different implementations. In most object-oriented systems overriding is handled through *late binding*, that is, the choice of which method is to be used is taken when the class of the object, on which the method is invoked, is known, thus at run-time rather than at compile time.

1.1.2 Composite Objects

The core model we have so far described does not capture integrity constraints and semantic relationships which are important in many applications. In particular, an important semantic relationship is the *composition* relationship. In an object-oriented data model, an object, called *composite object*, can be defined by aggregating other objects, called *component objects* [114]. The relationship relating a composite object with its components is called *aggregation* or *part-of* relationship. The fact that an object is defined as an aggregate of other objects raises a number of requirements. A first relevant constraint is related to referential integrity. Referential integrity ensures that whenever the value of an attribute of an object is another object, the referenced object exists. Violations to referential integrity must be detected upon any attempt to delete a referenced object. Violations to referential integrity can be repaired by forbidding the deletion, by setting to null the value of the attribute containing the reference or by cascade deleting also the object containing the reference. In addition to referential integrity, two other important constraints are related to composite objects, namely exclusiveness and dependency. An exclusiveness constraint specifies that an object must be component of only one composite object (possibly, with respect to a certain class or through a certain attribute). A dependency constraint specifies whether the existence of the component object is independent from the existence of the composite objects containing it, that is, whether the deletion of a composite object implies the deletion of its components. In addition of being relevant from the semantic point of view, the notion of composite object as a logical entity is useful in enhancing performance because a composite object can be considered a locking, authorization and clustering unit. In what follows we present composite object models and discuss referential integrity.

1.1.2.1 Composite Object Models

A lot of work has been carried on composite objects and part-whole relationships, in the data modeling [18, 101], knowledge representation [141, 174] and software engineering [73] areas. There is obviously a trade-off between the complexity of the model and the efficiency and simplicity of its realization. We restrict ourselves to composite object models typical of OODBMSs, in particular, we focus on the composite object model developed for Orion

[114].

An initial composite object model was proposed and implemented in the Orion project [113]. Tests carried out on a number of Orion applications showed that the concept of composite object is extremely useful. However, the initial model had several shortcomings. The first problem is that a component object must belong to a single composite object (exclusiveness property). This restriction is somewhat too strict for some applications; for example, in an hypertext management system the same chapter could quite reasonably belong to two different books. The second problem is that the model requires that the composite object is built in top-down mode. A component object cannot therefore be created if the father object[1] was not created first. This restriction means that composite objects cannot be created in bottom-up mode, that is, by assembling already existing objects. Finally, the model requires the existential dependence of the component objects from the composite objects to which they belong. If a composite object is deleted, all the component objects are automatically deleted by the system. This is useful as it means that the applications does not have to search for and explicitly delete all the component objects. However, in certain situations it means that it is not possible to reuse the components of a deleted composite object for creating a new composite object.

An extended model overcoming the above shortcomings has also been developed in the Orion project [114]. In this model, two types of references - weak and composite - are defined between objects. A weak reference is a usual reference between objects on which no additional semantics is superimposed. An object o has a reference to an object o' if this reference is the value of an attribute of o. A composite reference is a reference on which the part-of relationship is superimposed. A composite reference can, in turn, be *exclusive* or *shared*. In the former case, the referred object must belong to a single composite object, whereas in the latter case it can belong to several composite objects. The semantics of a composite reference is then refined by introducing the distinction between *dependent* and *independent* composite references. In the former case, the existence of the objet referred to is dependent upon the existence of the object to which it belongs, whereas in the latter case, it is independent. The deletion of a composite object results in the deletion only of the component objects which are dependent for their existence. The objects whose existence is independent are not deleted. Obviously, because the dependence/independence is orthogonal with respect to the exclusiveness/shared status, four possible types of composite references are obtained. The reference type defined in the first composite object model above [113], coincides with the exclusive dependent composite reference in the extended model, whereas other types of references were not supported by the original model. In case of shared dependent composite references, an object can be dependent upon several objects; this means that the deletion of a composite object results in the deletion of a shared component object only if all the other references to the object have been removed.

[1]The father object of object o is the object of which o is a direct component.

A further model, based on the previous one, has been proposed in [41] for dealing with multilevel object-oriented secure databases. That model extends the composite references defined in [114] by introducing two new forms of exclusiveness for composite references: *exclusiveness with respect to a class*, meaning that two instances of the same class cannot share a component, whereas instances of other classes can refer that component; *exclusiveness with respect to a class hierarchy*, meaning that two members of the same class cannot share a component.

1.1.2.2 Referential Integrity

Referential integrity defines a relationship among attributes referencing an entity in a different entity. In an object-oriented data model this means that if an object o contains a reference to an object o', the referred object, i.e. o', must exist. In some OODBMS, such as GemStone and O_2, referential integrity is automatically ensured, in that those systems do not support explicit object deletion, rather they use a *garbage collection* mechanism to determine which objects can be removed not being referred any longer by any other object. In systems with an explicit delete operation, if a referred object is deleted, the problem of *dangling pointers* may arise.

Referential integrity is not a peculiar constraint of object-oriented data models, rather it is present in all data models. The concept of referential integrity, originally developed by Codd [74] and further investigated by Date [77] and, more recently, by Shaw [159], is included in the SQL2 standard [52] which was accepted by ANSI and ISO in 1992. In this section, we briefly discuss the various options for referential integrity in SQL2. Referential integrity in the relational data model is an integrity constraint between a set of attributes (called *foreign key*) of a relation C (called *child*) and a set of attributes that is the primary key of a relation P (called *parent*). Integrity constraint violations are caused by update operations on the parent and child relations. According to the SQL2 standard, insert and update operations on the child relation are forbidden (backed out) if these would result in database states violating referential integrity. For deletions and updates of tuples in the parent relation, by contrast, in SQL2 different repairing actions can be specified. This is accomplished through clauses ON UPDATE and ON DELETE in the definition of the parent relation, stating how to handle modifications that may violate referential integrity. For each of the clauses above, different repairing actions can be specified:

- CASCADE: in case of update, the new values in the key are propagated to the referencing children, whereas in case of deletion the referencing children are also deleted;
- SET_NULL: the foreign key attributes in the referencing tuples of the child relation are set to null;

- SET_DEFAULT: the foreign key attributes in the referencing tuples of the child relation are set to a given default value;
- NO_ACTION: no action is taken; referential integrity remains violated and if no other operation is executed to correct the mismatch of the corresponding tuples, the complete work of the transaction will be backed out.

There is another important referential action not introduced in the SQL2 standard, but supported by most of the relational DBMS: RESTRICT. The semantics of this referential action is to forbid any change (update or delete) of a parent tuple as long as there are referencing child tuples. Although this action is not in the SQL2 standard, we include it in our discussion.

A problem with referential integrity constraints, as specified in SQL2, results from the possibility of interference when performing multiple referential actions on a tuple. That is, a straightforward implementation may lead to a *non-determinism* in the result of a user operation. The SQL2 standard prevents such non-determinism through the specification of a complex test carried out during the execution of referential actions. A detailed discussion of this approach can be found in the work by Markowitz [129].

1.1.3 Multiple Class Direct Membership and Roles

As we have seen, most object-oriented data models are based on the notion of class; objects with the same structure and behavior are grouped into classes, which are in turn organized in inheritance hierarchies. In most models, real-world entities are represented as instances of the most specific class in which they can be classified[2]. Thus, real-world entities are partitioned into a set of disjoint classes.

However, when modeling application domains, objects are often found that naturally belong to several most specific classes. Consider as an example, a class `person`, whose instances can be classified along orthogonal dimensions, such as `student`, `noble`, `male` and `female` (the corresponding hierarchy is illustrated in Figure 1.1(a)). According to the intuitive semantics, a person can be both a student, a noble and a female at the same time. Thus, the object representing this person does not have a unique most specific class, rather it has a set of most specific classes [34].

Although the above situation can be easily represented in a model with multiple inheritance by defining a subclass (say `noble_student_female`) of all the involved classes, this solution may lead to a lot of artificial subclasses, sometimes referred to as *intersection classes*

[2]An object o can belong to a set of classes S. We call an element of S that has no subclass in S a *most specific* class of object o.

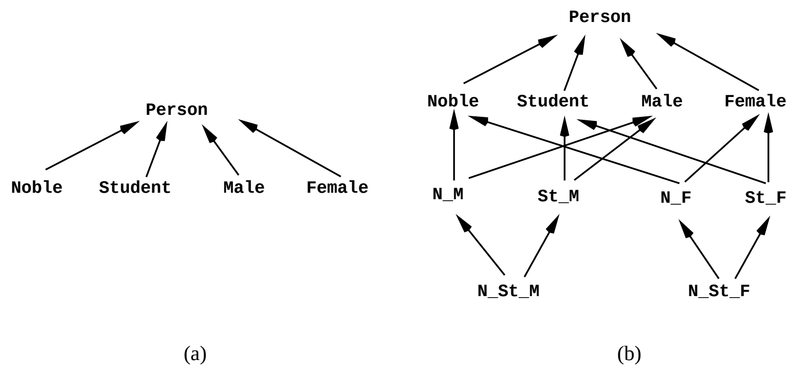

Figure 1.1: (a) Example of class hierarchy, extended in (b) with some meaningful subclasses

[158]. Referring to the hierarchy above, the meaningful subclasses of the *person* class are six (see Figure 1.1(b)). Thus, this approach can lead to a combinatorial explosion of sparsely populated classes, whose sole purpose is to allow an instance to have multiple most specific classes, without adding new state or behavior. Another problem with the multiple inheritance approach is that it only provides a single behavioral context for an object [148]. Indeed name conflicts among features in the superclasses are solved once for ever in the subclass definition (for example by imposing an order on superclasses, or with an explicit qualification mechanism) and the selected feature is the only one always considered whatever the context of the object reference is.

Several papers have pointed out the relevance of a more flexible binding between objects and classes, but only few of them elaborate on how to solve name conflicts or how to perform dispatching when objects are members of several most specific classes. The need of overcoming the restriction of the exclusive link between an object and its most specific class has been for instance stressed in [78]. The restriction that an object must be associated with a single most specific type was first overcome in the Iris OODBMS. In Iris, arbitrary types can be added to an object. Iris, however, does not support context-dependent behavior since the entire set of types of an object is visible in every context. To avoid conflicts, two different types of an object cannot have different methods with the same name. Stein extensively discusses in [164] the motivations for supporting multiple class direct membership, but her approach reduces an instance of multiple classes to an instance of a (virtual) subclass-by-multiple-inheritance of its classes. She does not discuss which policy to adopt for conflict resolution. Moreover, no context-dependent behavior can be supported by her approach.

Other proposals investigating how to support a flexible link between an object and the class it belongs to can be categorized into two main classes: object systems supporting state-based classification of objects and views, and object systems supporting roles. The following subsections are devoted to the description of those classes of approaches.

1.1.3.1 Predicate Classes

A predicate (or derived) class has all the properties of a usual class, including a name, a set of superclasses, a set of attributes and a set of methods. In addition, a predicate class has a predicate expression. A predicate class represents the subset of the members of its superclass(es) that also satisfy the predicate. Whenever an object is a member of the superclasses of the predicate class, and the predicate expression evaluates to true on the object, the object is automatically considered as a member of the predicate class. An object member of a predicate class has all the attributes and methods of this class. If the object state later changes and the predicate expression no longer evaluates to true, the object is excluded from the predicate class. Among languages supporting predicate classes we recall Cecil [68].

The predicate expression can test the value or the state of an object, thus supporting a form of implicit classification based on attribute values in addition to explicit classification based on types supported by traditional classes. Predicate classes are classes whose extents are not explicitly manipulated. Rather, they are derived by extracting from extents of other classes the objects satisfying a predicate. Predicate classes are thus also the base for *views* in object-oriented data models. Examples of view mechanisms for object-oriented databases based on derived classes are presented in [138, 150, 155, 162, 163] and they will be further discussed in Chapter 5.

In models supporting predicate classes it is difficult to ensure that an object is a direct member of a unique class, since it depends on the population predicates being disjoint. When methods are associated with predicate classes, method dispatching depends not only on the dynamic type of an argument, but also on its dynamic value or state. Method dispatching for predicate classes will be considered in Chapter 6.

1.1.3.2 Roles

Object models with roles support evolving objects (that is, objects migrating among classes) and objects that cannot be exclusively classified in a single class. In such object models, two different hierarchies are provided: a class (type) hierarchy and a role hierarchy. The role hierarchy is a tree of special types, called *role types*. The root of this tree defines the time-invariant properties of an object. The other nodes represent properties (types) that the object may acquire and loose during its lifetime. At any point in time, an entity is represented by an instance of the root type and an instance of every role type whose role it currently plays. When an entity acquires a new role, a role-specific instance of the appropriate role type is created; when it abandons a role, the role-specific instance is destroyed. Thus, the role concept supports the dynamic nature of entities and their non-exclusive classification. Moreover, entities can exhibit role-specific behavior and roles can be used to restrict access to a particular context. The main drawback of models with roles is that the different hierarchies (role and class ones) highly increase the complexity of the model. Such complexity impacts both the system architecture and the application development. For example, users must choose which features to model as classes and which as roles.

A first approach based on role hierarchies has been proposed by Sciore [158]. In his approach, real-world entities are modeled as object hierarchies where inheritance is determined on a per-object basis, thus merging class-based and prototype-based approaches. When an object receives a message, it either directly replies to the message or delegates the message to its parents. The observed behavior thus depends on the organization of the object hierarchy. A similar approach is proposed in [119]. Richardson and Schwartz [148] have introduced the concept of *aspect* to model roles in strongly typed object-oriented languages for database systems. More recently, Wieringa in [173] has pointed out that objects may reference a particular role of an object and not only an object itself. The relevance of roles in object-oriented analysis has been pointed ou by Pernici [143], Papazoglou [139], Martin and Odell [130]. The Fibonacci object data model [12] and the model proposed in [94] are quite similar. In both of them a role hierarchy can be associated with a root class; an object in this class can play any role belonging to the hierarchy. In both models, messages are dispatched according to the roles the object plays (though differently). Method dispatching in object data models with roles is discussed later on in Chapter 6.

1.1.4 Typing Issues

Typing is a very relevant aspect of programming languages. The basic idea behind typing is to partition data values according to the use intended for them, and thus to check for a correct use of data. This data classification technique is known as typing. In a typed programming language, each expression has an associated type, determining how the that expression can be used. This improves the program readability, reliability and efficiency. A type declaration can be seen as the statement of an invariant property of data. Type correctness of a programs means that all type declarations are valid invariants over data. A language is said to be type safe if no run-time type errors can occur in a program which has been proved type correct. An alternative to static type checking is that of verifying type correctness dynamically, that is, at run-time. Static type checking is obviously preferable for what concerns efficiency.

Type systems for object-oriented programming languages are more complex than those developed for traditional programming languages. This is due to the presence of user-defined types and subtype relationships. A basic principle related to this relationship is *substitutability* [55]: each instance of a subtype can be used in every place an instance of the supertype is expected.

To ensure substitutability and type safety of object-oriented languages, signature of operations are allowed to be redefined under the following constraints:

- the domain of an input parameter can be replaced in a subclass with a proper supertype (*contravariant* redefinition for operation input parameters);
- the domain of an output parameter can be replaced in a subclass with a proper subtype (*covariant* redefinition for operation output parameters).

This "contravariant for input, covariant for output" rule for the redefinition of operation signatures ensures that the new signature may be used everywhere the old one can be used, and guarantees that no type errors occur due to the redefinition of operations. There has been a long debate on whether this "contravariant for input" rule, besides being type safe, is also intuitive to use. A covariant redefinition of operation input parameter domains, by contrast, is more intuitive but it is not type safe. Some recent work [56] have tried to reconcile covariant and contravariant rules for the redefinition of operation input parameters, relying on the notion of multi-method (see Chapter 6).

In object-oriented data models, moreover, a form of domain specialization for attributes is also required. Thus, in many object-oriented data models a subclass may redefine the domain of an inherited attribute by specializing (or refining) it, that is, replacing it by a proper subtype (*covariant* redefinition of attribute domain). This covariant redefinition of attributes (the domain of an attribute may be specialized in subclasses) reflects what is usually needed when creating a taxonomy of classes; indeed, when specializing a class the designer usually needs to add new attributes or to specialize existing ones. The problem arising because of the covariant redefinition of attributes in inheritance hierarchies has been firstly recognized by Cardelli [54] and is shown by the following example.

Example 1.1 *Consider a class* `person` *with an attribute* `spouse` *having domain* `person` *and a class* `noble` *(subclass of* `person`*) which refines the domain of the attribute* `spouse` *in* `noble`[3]*. Suppose that in class* `person` *an operation* `set-spouse` *is defined, which takes as input a value of type* `person` *and assigns that value to attribute* `spouse`*. This operation is not redefined in class* `noble`*. Now consider two variables* `P`*,* `N` *both declared of type* `person`*. The expression* `N.set-spouse(P)` *is statically type correct but, if at run-time* `N` *is instantiated to a noble and* `P` *to a (not noble) person, it results in trying to assign a value of type* `person` *to an attribute whose domain is* `noble`*.* ◇

There are many possible alternatives for dealing with covariantly redefined attributes. The approach taken by Cardelli in [54] and adopted by Galileo [13] is to syntactically

[3]Here, more or less realistically, we suppose that the spouse of a noble is always a noble.

distinguish among mutable (assignable) and immutable (not assignable) entities and to disallow covariant redefinition for mutable entities (immutable entities may be assigned their value at object creation only). Thus, all the attributes that must be redefined cannot be updated; on the contrary, if an attribute should change its value during the lifetime of its enclosing object, then it may only be inherited, but not redefined, by subclasses. This solution is obviously not the most adequate for the database context, in which objects are long-living evolving entities.

Another possible solution to the problem is to limit the context in which subtype substitution is allowed. A proposal in this direction is by Connor and Morrison [76]; their approach guarantees static type safety at the expense of limiting the expressiveness of the language and introducing a lot more typing information inside programs. Note moreover that in their type system, type specifications are exact, that is, a variable declared of type T cannot be instantiated at run-time with an instance of a subtype of T.

The Eiffel language [133] allows the covariant redefinition of attributes and catches every attempt of incorrect assignment at program linking time, using an approximate and pessimistic algorithm which computes an approximation of the possible types of the objects that may be referenced by each entity at each point of the program where the entity is used. The TDF technique [145] implements a similar concept, but differs in the algorithm used to compute the type sets: TDF adopts a data-flow technique, while Eiffel's global verification does not take into account the possible control flow of the program. Both techniques, though, loose much of their precision when persistent data are involved, since it is not possible to determine the types of the objects referenced by persistent variables. In the example, the bindings for variables `N` and `P` may be taken from the extents of classes manipulated by programs different from the one examined.

O_2 does not restrict either the assignment or the covariant redefinition of attributes and checks the validity of every assignment dynamically, possibly generating a run-time error. This approach is thus to consider the domains of attributes as integrity constraints, to be checked at run-time, rather than dealing with them as type constraints, to be checked statically. Thus, whenever a value is assigned to an object attribute it is dynamically checked whether the value is appropriate for the domain.

Another typical data model feature that can cause problems with respect to type checking is object migration, that is, the possibility for an object to dynamically change its type. This could result in a variable declared of type T in denoting a value which is no more a legal value for type T.

Another feature of object-oriented models that may cause some problem with respect to type checking is related to predicate classes, as shown by the following example.

Example 1.2 *Consider a class* `employee` *with a derived subclass* `rich-employee` *with the* *population predicate* `X.salary > 5000`. *Consider, moreover, a variable* `X` *defined with type* `rich-employee`. *If the value of* `X.salary` *is modified to* `4000`, `X` *can no longer be an instance of the class* `rich-employee`. *This leads to an inconsistency.* ◇

To avoid such problem, different solutions can be adopted. Two of them, namely

- disallowing a variable to be declared with a type corresponding to a derived class;
- disallowing updates on the attributes appearing in the population predicate

are conservative solutions. If the population predicate can only be falsified by updates on the object on which it is evaluated[4] the above solutions can be refined by allowing a variable to be declared with a type corresponding to a derived class but disallowing updates on the attributes appearing in the population predicate to be applied to that variable. Such an approach prevents, by static checks, a variable of type T from referencing at run-time an object which is not an instance of the class corresponding to T. This approach emphasizes the type checking view.

An alternative solution is to regard population constraints as other constraints and thus to check them at run-time. This approach does not ensure that a variable of type T, with T corresponding to a derived class, always references a member of the class corresponding to T. Rather a check is performed at run-time to detect whether the variable references an object that meets the population constraint. If not, an error is raised.

This approach requires some type checking at run-time and thus it is potentially less efficient. Note, however, that the above solutions are not exclusive. They can be combined to obtain a good compromise between semantic richness and efficiency. For example, a variable can be allowed to be declared of a type corresponding to a derived class, and updates on the attributes appearing in the population predicate can also be applied to that variable, but run-time checks for that variable (and only for that one) must be performed. More sophisticated solutions, based on flow analysis of application code [145], could also be investigated.

1.1.5 Comparison Among Data Models of Existing Systems

During the last years, several object-oriented database system have been developed, both as experimental prototypes and as commercial systems. Among them, we recall the Orion/Itasca system, developed at MCC, the Iris/OpenODB system, developed at Hewlett-Packard laboratories, the Ode system, developed at AT&T Bell Labs, the GemStone system

[4]This is not always true, e.g. if the population predicate makes use of aggregate operators.

	GemStone	Iris	O_2	Orion	ObjectStore	Ode	ODMG
reference	[48]	[83]	[23, 80]	[110, 116]	[137]	[9]	[59]
class extent	NO	YES	NO	YES	NO	YES	YES[a]
persistence	R	A	R	A	R	2op	A[b]
explicit deletion	NO	YES	NO	YES	YES	YES	YES[b]
direct access to attributes	NO	YES	P	YES	P	P	YES
domain specification for attributes	O	M	M	M	M	M	M
class attributes and methods	YES	NO	NO	YES	NO	NO	NO
relationships	NO	YES	NO	NO	YES	NO	YES
composite objects	NO	NO	NO	YES	NO	NO	NO
referential integrity	YES	NO	YES	NO	YES[c]	NO	YES[c]
multiple inheritance	NO	YES	YES	YES	YES	YES	YES
migration	L	YES	NO	NO	NO	NO	NO
exceptional instances	NO	NO	YES	NO	NO	NO	NO

Legenda:

R = root persistence, A = automatic, 2op = two different *new* operations.

P = only for public attributes.

O = optional, M = mandatory.

L = in limited form.

[a] For those classes in which definition an `extent` clause is specified.

[b] In C++ OML created objects are automatically persistent and explicit deletion is supported, in Smalltalk OML persistence is by root and there is no explicit delete operation.

[c] Referential integrity is ensured for relationships but not for attributes.

Table 1.1: Comparison among data models of most common OODBMSs

of ServioLogic, the ObjectStore system of Object Design, the O_2 system of O_2 Technology, ONTOS of Ontologic, Objectivity of Objectivity Inc., Versant of Versant Technology. Those systems represent only a partial list of available OODBMSs. A proposal of standard object data model has also been proposed by ODMG. Table 1.1 compares some of these systems along a number of dimensions.

In particular, we distinguish systems in which classes also have an extensional functionality, that is, in which with a class the set of its instances is automatically associated, from those in which object collections are defined and handled by the applications. We point out, moreover, the persistence mechanism adopted, distinguishing among systems in which all objects are automatically created as persistent, systems in which persistence is ensured by linking an object to a persistence root (usually an external name), and systems supporting two different creation operations, one for creating temporary objects, the other one for creating persistent objects. The different policies adopted with respect to encapsulation are also shown, distinguishing among systems forcing strict encapsulation, systems allowing to directly access attribute values and systems distinguishing among private and public features.

An important concept which exists in many semantic models and in models for the conceptual design of databases [70] is the *relationship*. An relationship is a link between entities in applications. An relationship between a person and his employer (*) is one example; another (classic) example is the relationship between a product, a customer and a supplier (**) which indicates that a particular product is supplied to a particular customer by a particular supplier. Associations are characterized by a *degree*, which indicates the number of entities participating in the relationship, and by *cardinality constraints* which indicate the minimum and maximum number of relationships in which an entity can participate. For example, relationship (*) has degree 2, that is, it is binary, and its cardinality constraints are (0,1) for person and (1,n) for employer. This reflects the fact that a person can have at most one employer, whereas an employer can have more than one employee. Referring to maximum cardinality constraint, relationships are partitioned in one-to-one, one-to-many and many-to-many relationships. Finally, relationships can have their own attributes; for example, relationship (**) can have attributes `quantity` and `unit price`, indicating, respectively, the quantity of the product supplied and the unit price quoted. In most object-oriented data models relationships are represented through object references.. This, however, imposes a directionality on the relationship. Some models, by contrast, allow the specification of binary relationships, without proper attributes.

Finally, the O_2 system allows the specification of exceptional instances, that is, of objects that can have additional features and/or redefine (under certain compatibility restrictions) features of the class of which they are instances.

1.2 Active Rules in Object-Oriented Databases

An *active* database system is a database system which automatically performs certain operations in response to certain events occurring or certain conditions being satisfied [65]. Active databases enable important applications, such as alerting users that a given event has occurred, reacting to events by means of suitable actions, and controlling the invocation of operations and procedures. Examples of functions that can be effectively performed by active database systems, but that in traditional database systems must be encoded in applications or require special-purpose subsystems, are integrity constraint enforcement, monitoring, authorization, statistics gathering and view handling.

Active database systems are centered around the notion of *rule*. Rules are syntactic constructs by means of which the reactions of the system are specified. Active rules, often referred to as triggers, are usually defined according to the *event-condition-action* (ECA) paradigm. Events are monitored and their occurrences cause the rule to be triggered; a condition is a declarative formula that must be satisfied in order for the action to be executed, whereas the action specifies what must be done when the rule is triggered and its condition is true.

Most of the research and development efforts on active databases and commercial implementations have focused on active capabilities in the context of relational database systems. Recently, however, several approaches have been proposed to incorporate active rules into object-oriented database systems. The advantages of adding active rules to relational databases are well recognized. A relational database does not provide comprehensive means to formally state the semantics of data, the high-level semantic operations on data and the integrity constraints. Semantics is by contrast coded in application programs. Constraints and rules are thus hidden in application programs and it is difficult to enforce and maintain them. Since object-oriented databases incorporate behavior as well as data, it is less obvious why active rules are still useful in object-oriented systems. In relational active database systems the behavior is specified either by rules or by applications; in object-oriented active database systems, by contrast, behavior can be specified by rules, by applications or by methods. However, specifying the semantics of data through rules has three important advantages over coding it into methods. First, the behavior represented by methods must be explicitly invoked by the user or by applications, while active rules are autonomously activated. Second, the semantics represented by a single rule often needs to be replicated in the code of several methods. Consider for instance integrity constraints: since a single constraint can be potentially violated by the execution of different methods, the code for enforcing the constraint should be replicated in all those methods. Finally, a specialized trigger subsystem, internal to the database system, supports a more efficient active behavior processing compared to the approach where the active behavior is coded into methods.

The paradigm shift from the relational model to the object-oriented one warrants a re-examination of the functionalities as well as the mechanisms by which reactive capabilities are incorporated into the object-oriented data model [26]. There are several factors not present in relational database systems that complicate the extension of object-oriented database systems to include active behavior. Among them, let us mention that in object-oriented data models, in contrast to a fixed number of predefined primitive events of the relational model, every method/message is a potential event. Moreover, issues related to scope, accessibility, and visibility of object states with respect to rules should be investigated. Other important, open issues concern trigger inheritance and overriding. Because inheritance is a central notion of the object-oriented data model, the definition of a proper approach to rule inheritance is crucial.

In the last few years, there has been a growing interest in extending object-oriented database systems with reactive capabilities. One of the earliest object-oriented active database projects is HiPAC [79, 131]. HiPAC provides an expressive active database rule language with flexible execution semantics, supporting a full complement of coupling modes through a nested transaction model. There are some projects on active object-oriented databases that are follow-on of HiPAC: among them, *Sentinel* [17] and REACH [46]. One of the best known active object-oriented database systems is Ode [89, 91, 90], which extends the O++ database programming language with facilities for expressing rules in the form of constraints and triggers. Among other projects we would like to mention Adam [81], NAOS [75], SAMOS [88] and TriGS [104].

Table 1.2 compares these systems along a number of dimensions. Among the considered systems, NAOS and TriGS are extensions of commercial object-oriented database systems with active capabilities. All the proposals support ECA rules; however, in some systems, such as Ode, conditions are part of the event specification as a mask which qualifies the event and can refer to parameters of the method call defining the event.

In an active rule, the *event* specifies what causes the rule to be triggered. Relevant triggering events are internal events related to database operations, temporal events, external (that is, raised by the application), and user-defined events. Database operations are object accesses, attribute value updates, object creations and deletions, method executions and calls to transaction primitives (e.g. commit). In Chimera, also object migration between classes is included among events. In addition, in some systems it is possible to specify whether a rule must be triggered after or before its triggering event. Triggering a rule before its event is meaningful when methods are considered as events, and allows method preconditions to be tested. Triggering events may also be *composite*, that is, combinations of other events. Useful operators for combining events are logical operators, such as conjunction, disjunction and negation, and sequences. Some systems allow events to be parameterized; when a parameterized event occurs, values related to the event are bound to the event parameters, and these parameter values can be referenced in rule conditions

	HiPAC	Ode	Adam	NAOS	TriGS	SAMOS	Chimera
Reference	[79]	[89, 91]	[81]	[75]	[104]	[88]	[61]
o-o data model	OODAPLEX	new	new	O_2	GemStone	new	new
primitive events	messages db ops temporal external	messages db ops temporal	messages	messages db ops user-def	messages	messages db ops temporal user-def	messages db ops migrations
event composition	YES	YES	NO	NO	NO	YES	YES[a]
parametric events	YES	NO	YES	NO[c]	NO[b]	YES	NO
parameter passing	$E \to C, A$ $C \to A$	$E \to A$	$E \to C, A$	$E \to C, A$ $C \to A$	$E \to C, A$[b]	NO	$C \to A$
conditions on past states	NO	NO	NO	NO	NO	NO	YES
net effect	NO	NO	NO	YES	NO	NO	optional
evaluation mode	immediate deferred decoupled	immediate deferred decoupled[d]	immediate	immediate deferred[e]	user-specified	immediate deferred decoupled	immediate deferred[e]
untargeted/ targeted rules	untargeted	targeted	untargeted[f]	untargeted	both	both	both
overriding	NO	YES[g]	NO	NO	YES	NO	NO

Legenda:

[a] In [61] only disjunction of events is considered; an extension of Chimera with other kinds of event composition is described in [134].

[b] The only parameter is the object receiver of the event (on which the rule is being executed).

[c] Delta tables are used for referring to objects affected by events.

[d] This coupling mode is between Event and Action; immediate and deferred coupling modes between Event and Condition can be supported by means of event composition.

[e] This coupling mode is between Event and Condition; coupling mode between Condition and Action is always immediate.

[f] Triggers are defined outside the scope of classes, however they are indexed on classes for efficient detection of relevant events.

[g] Rules can be overridden provided that they are not activated in the superclass.

Table 1.2: Comparison of active object-oriented data models

or actions. Some systems only consider as an implicit parameter the identifier of the object receiver of the event, that is, the object on which the rule is executed.

In an active rule, the *condition* specifies an additional condition to be checked once the rule is triggered and before the action is executed. Conditions are predicates over the database state. If the rule language supports parameterized events, the condition language includes a mechanism for referencing the values bound to the event parameters. In some systems, values related to the condition can be passed to the action. In the most common approach for parameter passing, the condition is expressed as a query returning data, that are then passed to the rule action. Moreover, some systems allow triggers to refer to past database states.

In an active rule, the *action* is executed when the rule is triggered and its condition is true. Possible actions include database operations and calls to application procedures. Several active database rule languages allow sets of actions to be specified in rules, usually with an ordering, so that the actions in a set are sequentially executed.

The considered systems differ not only with respect to the supported rule language, but also for the rule execution semantics. First of all, active database rule execution can be either *instance-oriented* or *set-oriented*. With an instance-oriented execution, a rule is executed once for each database "instance" triggering the rule and satisfying the rule condition. By contrast, rule execution is set-oriented if a rule is executed once for all database instances triggering the rule and satisfying the rule condition. The issue of instance-oriented versus set-oriented execution is obviously related to the issue of rule processing granularity. The most straightforward approach is to evaluate a triggered rule condition and to execute its action within the same transaction in which the triggering event occurs, at the soonest rule processing point. However, for some applications it may be useful to delay the evaluation of a triggered rule condition or the execution of its action until the end of the transaction; or it may be useful to evaluate a triggered rule condition or execute its action in a separate transaction. These possibilities result in the notion of *coupling modes* [131]. One coupling mode can specify the transactional relationship between a rule triggering event and the evaluation of its condition, while another coupling mode can specify the transactional relationship between a rule condition evaluation and the execution of its action. Possible coupling modes are:

- *Immediate*: takes place immediately, within the same transaction. For example, an immediate mode for the action with respect to the condition means that the action is executed as soon as the condition is evaluated (if the condition is true).
- *Deferred*: takes place at the commit point of the current transaction. For example, a deferred mode for the condition with respect to the event means that if the event occurs during a transaction, then the condition is evaluated at the end of that transaction. The deferred mode can be useful, for example, for rules that enforce

integrity constraints, since a transaction may execute several operations that violate a constraint (thus triggering the relevant rules), but the transaction may restore the constraint before it reaches its commit point.

- *Decoupled*: takes place in a separate transaction. This mode can further be split into *dependent decoupled*, where the separate transaction is not spawned unless the original transaction commits, and *independent decoupled*, where the separate transaction is spawned regardless of whether the original transaction commits. The decoupled mode can be useful when many rules are triggered, in order to decompose the resulting large transaction into a set of smaller ones.

Among the considered systems, all but Chimera support an instance-oriented rule execution. Chimera supports set-oriented rule execution, while NAOS rule execution is instance-oriented for immediate rule and set-oriented for deferred ones.

A further aspect in rule execution is to consider the *net effect* of a sequence of operations performed in the triggering transaction. For instance, if a rule is triggered by the creation of an object, but this object happens to be deleted before the actual execution of the rule, the rule should not be executed. The net effect of events is computed based on the classical composition of pairs of operations applied to the same object[5]. Another difference among existing systems is whether rule definitions are attached to classes. Attaching rule definitions to classes enhances modularization and allows an efficient detection of relevant events while there are sometimes useful rules triggered by events spanning sets of objects possibly from different classes (untargeted rules).

Finally, an extremally relevant notion for active rule semantics is that of rule *priority*. The execution semantics for active rules sometimes requires that one rule is selected from a set of eligible rules. For this reason, an active database rule language may include a mechanism for specifying rule priorities. Priorities might be specified by ordering the set of rules, by declaring relative priorities between pairs of rules, or by assigning a numeric priority value to each rule. Relative priorities are the most flexible approach, since they subsume the other two types of priority specification. Ordering the entire set of rules may not be necessary for achieving the correct behavior, while numeric priorities can be difficult to use since they may need to be adjusted as the set of rules evolves.

The approach taken by the majority of the systems for rule inheritance is to simply apply all rules, defined in a class, to the entire extent of the class, that is, to all the members of the class itself. No rule overriding is supported by those systems. Only few systems [91, 104] support rule overriding, though in a completely uncontrolled way.

[5]Note that net effect is supported with different semantics in NAOS and Chimera. Whereas in NAOS the elimination of events due to net effect composition results in de-triggering of rules, this is not true for Chimera.

1.3 Deductive Rules in Object-Oriented Databases

Deductive rules [62] are the usual rules of logic programming, exploited in the database field to define derived or intensional data, that is, data not stored in the database. Indeed, those rules allow to derive new information from the facts explicitly stored in the database and allows to express in a convenient way recursively defined data.

A deductive rule has the form

$$template \leftarrow query$$

where *template* specifies the structure of the derived data, and *query* is the expression that computes the derived data.

A deductive database consists of a finite set of *facts* and *clauses*. Facts are assertions about an application domain, such us "John is the father of Harry". Rules are sentences which allow facts to be deduced from other facts. An example of a rule is: "If X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z". Note that rules, in order to be general, usually contain *variables* (in our case, X, Y, and Z). In Datalog both facts (H) and rules ($H \leftarrow B_1, \ldots, B_n$) are represented as definite Horn clauses where H and each B_i are atoms of the form $p(t_1, \ldots, t_k)$ such that p is a *predicate symbol* and the $t_j's$ $(1 \leq j \leq k)$ are *terms*. The symbols "$\leftarrow$" and "," denote the logical implication and conjunction, respectively. A term is either a *constant* or a *variable*. An example of deductive rules is the following, specifying, in a Datalog-like syntax, a recursive view that starting from an extensional relation `parent` computes an intensional relation `ancestor`:

```
ancestor(X,Y) ← parent(X,Y)
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)
```

Any Datalog program consists of facts and rules and must satisfy the following *safety conditions*: each fact is *ground* (that is, contains no variables) and each variable which occurs in the head of a rule must also occur in the body of the same rule. These conditions guarantee that the set of all answers to a query which can be derived from a Datalog program is a finite set of ground substitutions.

The knowledge of an application domain in Datalog is organized into two components, the *Extensional Database (EDB)* and the *Intensional Database (IDB)*. The *EDB* is a set of ground facts, while the *IDB* is a set of Datalog clauses. The predicate symbols occurring in *EDB* and in *IDB* are divided into two disjoint sets: the *EDB-predicate symbols*, which are the ones occurring in the *EDB* and the *IDB-predicate symbols*, which occur in *IDB* but not in *EDB*. Moreover, it is required the head predicate symbols of each clause in *IDB* are

IDB-predicate symbols. EDB-predicate symbols may occur in *IDB*, but only in a clause body.

Several research proposals have attempted to combine object-orientation, databases and logical languages [38]. A first research direction uses logic to formalize the notions underlying object-oriented data models, resulting in the so called logics for objects. The approaches in this area consider (complex) objects as terms, take into account a notion of schema, and put a great emphasis on formal semantics. No update and evolution aspects are considered.

Another research direction is that of employing logical languages as query languages for object-oriented databases, leading to the definition of object-oriented logical languages primarily used for querying object-oriented databases. The most significant example of such a language is IQL [5]. The last direction we consider[6] is that of extending Datalog to specify object-oriented databases. In such extensions, the logical component is used to specify the schema of the database, and a distinction is made between base relations and derived relations. Examples of such systems are LOGRES [51], COMPLEX [95] and Datalog^{meth} [6].

Table 1.3 summarizes some of the differences among different approaches to the integration of the object and deductive paradigms in the database context. The table points out that most of the approaches do not consider state evolution of deductive objects. In addition many of the considered approaches [5, 95, 51] do not consider the behavioral component of objects, that is, the methods. We think that this is an important issue because it overcomes the dichotomy between data and operations of the relational model. In what follows we discuss in greater details some of the mentioned proposals.

1.3.1 Logics for Objects

The approaches we classify in this subsection consider objects as terms and do not take into account state evolution of objects. These proposals originate from Maier's O-logic [128], and evolve in C-logic [72] and the revised O-logic [109]. The final effort in this direction, which extends the previous ones, is *Frame Logic* (F-logic) [108]. It extends previous works with higher-order features to accommodate methods within complex terms. The motivations underlying those approaches is the development of a logical framework for representing complex objects, thus overcoming the absence of formal foundations for object-oriented data models. Those approaches attempt to define model-theoretic semantics for object-oriented logics. C-logic and O-logic deal with complex objects and object identity, taking into account an inheritance hierarchy on classes, while F-logic extends them by

[6]Other proposals exploit deductive rules to define intensionally populated classes. These proposals has been already mentioned and will be discussed in Chapter 5.

	Object Granularity	Updates	Methods	Schema	Inheritance	Message Passing	Oid Invention
F-logic	Term	NO	YES	YES[a]	YES	NO	NO
COL	Term	NO	NO	YES	NO	NO	NO[b]
IQL	Term	YES[c]	NO	YES	YES	NO	YES
LLO	Term	NO	YES	YES	YES	NO	NO
DTL	Term	NO	NO[d]	YES	YES	NO	NO
LOGRES	Term	YES	NO	YES	YES	NO	YES
COMPLEX	Term	NO	NO	YES	YES	NO	NO
Datalog^{meth}	Theory[e]	NO	YES	YES	YES	YES	NO

Legenda:

[a] A schema level is present, but the languages does not distinguish between types and instantiations of types, collapsing the subtype and instance-of hierarchies in a single relationship.

[b] The approach does not support the notion of oid.

[c] Only creation of objects is supported, whereas object deletions and state evolution are not considered.

[d] The TM data model considers methods, but they are not declaratively expressed by the DTL query language.

[e] In Datalog^{meth} an object is identified by an identifier and associated with a class. Object behavior is associated with classes, whereas the object state is represented as a set of predicates relating properties to object identifiers.

Table 1.3: Comparison among different approaches to the integration of deductive and object data models

considering also methods and schema.

Both C-logic and O-logic model complex objects by considering extended terms with an object identifier, a type and a structured value which is specified by a record structure with a number of labels each having assigned a value. Properties of an object are thus represented by labeled values. Both approaches consider sets in that they support non-functional set oriented labels. Inheritance is taken into account by modeling the set of classes as a partially ordered set. A dynamic notion of type (i.e. its extent, instead of its structure) is considered.

F-logic extends the previous logics by incorporating the notions of methods and schema. However, in F-logic the schema level and instance level are not distinguished. Thus a unique ISA hierarchy represents both the subtype and instance-of relationships. In F-logic the distinction among objects, classes and relationships is abolished. Thus inheritance, methods and schema can be reasoned about in the formalism. In F-logic both classes and individual objects are taken from the same domain and and are organized in a lattice. In F-logic inheritance is built into the semantics, that is, whenever an elements p precedes v in the ordering associated with the lattice, properties of p also holds for v. They support a monotonic overriding (not a complete one), that is, the most specific information for an object is added to more general ones, without nullifying it. When inheritance contradicts the other information, a local inconsistency arises.

Although F-logic formally has a first order semantics, it is able to model some higher order features such as sets, class-subclass hierarchies and schema. Schema can be reasoned about because labels, that is attribute names, are virtually indistinguishable from objects. F-logic also supports declarative method specification, through rules. Declarative definitions of methods are possible, being methods essentially labels with parameters.

1.3.2 Logic as a Query Language for Object-Oriented Databases

The first approach in this direction has been a Datalog extension to deal with structured objects, defining query languages whose corresponding data model is not a 1NF relational model, rather an extended relational model. COL (*Complex Object Language*) [3] is an example of a deductive language for value-based models supporting structured objects, even if it does not support either the notion of class or inheritance. COL is a Datalog extension allowing the manipulation of structured values obtained using tuple and set constructors. Besides relations, base and derived ones, the language supports data functions. Data functions are multi-valued functions defined either extensionally (base ones) or intensionally (derived ones). A data function is a relation satisfying a functional dependency. Those functions support the manipulation of structured values. In COL besides types obtained by applying tuple and set constructors, union types are considered, since they support the manipulation of heterogeneous sets. The language does not consider neither object identity nor inheritance.

IQL IQL (*Identity Query Language*) [5] is an extension of COL with the concepts of object identity and inheritance. It is centered around the concept of object identifier (oid) as a powerful primitive for database query languages. Informally, oids in IQL are typed pointers. Oids are used for three main purposes: *(i)* representing structures with sharing and cycles; *(ii)* set manipulation[7]; *(iii)* enriching the expressive power of the query language. IQL data model is an object based model, with a clear distinction between schemas and instances. The model supports both relations and classes.

The IQL language is rule based, static typecheckable and complete. It is obtained by enriching Datalog to obtain an object based language with sufficient expressive power to invent oid. Besides tuple and set types, IQL supports oid invention and a weak mechanism of oid assignment. Oid invention is performed through variables appearing in rule heads but not in rule bodies. Oid invention is used to store in objects query results, for set manipulation and to ensure the language completeness. All algebraic transformations on complex objects may be expressed in IQL. IQL also supports inheritance, which is however not handled directly, but using the notion of union types.

The semantics of the language is defined in terms of an inflationary fixpoint operator, extended with the mechanism of oid invention. We remark that the language supports object creation, but neither forms of object deletion nor state update functionalities are available.

LLO LLO (*Logic Language for Objects*) [127] is a declarative query language for an object-oriented data model, including also methods. A method is defined as a clause, whose head is the method interface and whose body is the method implementation. A query is seen as a message and the unification algorithm takes care of message handling. Metavariables (i.e. variables which assume types as values) are introduced in the data model, used for data abstraction, information hiding and inheritance. Inheritance hierarchies are obtained by instantiating metavariables.

In LLO each object has a state and a behavior. The structure of an object is described by its type. A predicate name is seen as an object identifier whose extension is the object value. The structural part of an object consists of an oid and a complex state. The object state may be directly accessed if the object is not encapsulated, otherwise interface methods must be defined. An LLO rule is a method. Methods are used to access object attributes and for deriving new information from object states. Both method overloading

[7] In IQL object identifiers are used for the same purposes of grouping in $\mathcal{LDL}$ and data functions in COL.

and method overriding are supported. Method inheritance is handled following class/type hierarchies. Behavior inheritance is thus considered.

DTL DTL (*DataTypeLog*) [20] has been designed as a query language for a database specified in the TM language. TM [21] is a high-level specification language for object-oriented database schemas, incorporating predicative sets as first-class objects and static constraints in the context of multiple inheritance and full static typecheckability. In TM objects have an object identifier used for sharing and referential integrity. Unlike F-logic, a distinction is made between types and instantiations of types.

DTL incorporates general set constructions, combining predicates and (multiple) inheritance. It offers the possibility of freely navigating through the terms by successive projections on attribute components. DTL is a typed extension of Datalog supporting complex terms (with record and set constructors), attribute projections, and using strict typing rules for equality predicates. An ISA predicate, that allows a more liberal comparison of specialized expressions with generalized ones, is provided. Predicate inheritance is also considered in DTL. It specifies a simple but powerful way of combining multiple inheritance with predicates in logic programming.

1.3.3 Object-Oriented Extensions of Datalog

In this subsection we consider systems that have a built-in deductive approach, that is, which do not use logic only as a query language but in which logic is deeply used in the specification of the data model.

LOGRES The LOGRES language [51] integrates an object-oriented data model with a rule based approach for expressing queries and updates. The data model allows object sharing and supports inheritance hierarchies, while the query language is a Datalog extension to support general type constructors. The static structure of the database is specified through type equations and integrity constraints. The LOGRES data model supports both classes and relations, whose schema is defined through type equations. The main characteristics of the model are static typecheckability, separation of schema and instance levels, associations and complex type constructors, data functions. The LOGRES rule language is a typed extension of Datalog characterized by a deterministic semantics, data functions, complex typed variables, object creation through oid generation[8], negation in rule bodies and in rule heads. Negated literals in rule heads are interpreted as tuple deletions, thus LOGRES supports an explicit deletion mechanism.

[8]As in IQL, unbound oid variables in rule heads cause the generation of a new oid.

The language provides a modularization functionality, in that it is possible to define modules encapsulating queries and updates. A module is a collection of type equations and rules. Moreover update modules may be defined to provide state evolution. When specifying the application of a module to a database, the application mode must be specified, thus different rule semantics are supported. Modules partition rules into various components, with different semantics corresponding to different module application modes. The declarative semantics of rules is preserved, whereas the control strategy is inserted into modules.

The most remarkable characteristic of LOGRES is its proposal of inserting updates in a declarative language, by keeping separated logic in rules and control in modules, whose application mode determines the effect on the database. Intensional updates are considered, too. Explicit deletions in rule heads and oid invention are introduced in a deterministic framework with an inflationary semantics. We remark moreover that LOGRES provides a unit of modularity, but this notion is orthogonal to the notion of object, which -also in this approach- is that of a term.

COMPLEX COMPLEX [95] is a system whose language (C-Datalog) is an extension of Datalog with constructs for handling complex objects, object identity and (multiple) inheritance. Such language supports completely declarative programming, and query evaluation may be performed either top down or bottom up. In C-Datalog there are two kinds of entities: classes, whose instances are base facts stored in the database, and derived relations, which describe relationships among class instances. Class instances (objects) have identifiers and are organized into a hierarchy. By contrast, an instance of a derived relation, called object association, is a tuple of objects describing relationships among objects that are not stable, rather they are derived by using deductive rules. The separation between schema and instance levels is maintained. With each class name, a set of ground facts, called the class definition, is associated representing the class proper instances. The definitions of classes must satisfy a number of constraints, namely oid uniqueness and referential and type integrity. The definition of a derived relation is a set of rules used for inferring the instances in its extension. Given a schema, the corresponding C-Datalog program is a set of definitions, one for each entity of the schema.

Class atoms may appear in rule bodies and support a form of functional composition that allows to navigate through objects using their connections. C-Datalog allows complex objects and aggregation links to be modeled. However, no constructors of set and tuple for structuring types of such objects are provided. In COMPLEX an object is modeled as a tuple of a relation.

Methods & Rules In [6] an extension of Datalog with methods, inheritance, overloading and late binding is considered. The resulting language is called Datalog^{meth}. The main concern of the work is inheritance with overriding. Classes are structured in a class hierarchy, and each class has an extension, the set of oids of the objects assigned to the class, and an associated behavior, which is the set of its methods. Methods and predicates are used to express properties of objects. Methods are functions, whereas predicates are not. Methods are specified at class level, that is, the rules which define methods are attached to classes. By contrast, facts for each predicate are specified for each object, by specifying the object attribute values. A resolution for Datalog^{meth} is specified, taking into account also overriding. This can be seen as an application of metaprogramming.

1.4 Discussion

To date, several commercial OODBMSs have been developed. The most known ones are O_2, GemStone, Iris/OpenODB, Orion/Itasca. A standard data model has also been proposed by the ODMG group. Among them, only Iris support a limited form of integrity constraints (through the use of key/non key qualifiers), intensionally defined attributes, declarative method implementation and the possibility of expressing views through functions not associated with a specific type. Moreover, extensions of O_2 with views [162] and triggers [75] are being independently investigated. More recent proposals of object-oriented data models incorporate, though to different extents, rules. The TM data model, developed at the University of Twente, supports a deductive query language and allows declarative method implementation as well as integrity constraint specification. The *Cocoon* data model, from the University of Ulm, allows the specification of limited forms of integrity constraints and supports a view mechanism. Finally, the *Ode* database system, developed at AT & T Bell Labs, is a prototype of active OODBMS, supporting integrity constraints and triggers.

The field of object-oriented databases suffers for the absence of a common reference data model, with a formal (that is, precise) definition. There have been, however, some formal definitions of object-oriented data models. We will consider and compare them in detail with our proposal in Chapter 2. However, those models do not adequately cover all features of a real object-oriented data model. Some of them are only concerned with specific aspects.

As far as languages are concerned, two different kinds of languages are of interest in object-oriented databases: the query language (as in relational databases) and the method definition language. An important requirement of the query language is declarativity which results in ease of use and provides potential for query optimization. Different declarative query languages for object-oriented databases have been proposed [1, 43] and some preliminary analysis on their expressive power have been performed. The use of deductive languages as query languages for object-oriented databases has also been proposed. For what concerns method definition language, in most object-oriented databases methods are specified in an object-oriented programming language (for instance, SmallTalk for GemStone, an object-oriented extension of C, CO_2, for O_2, C++ for Ode). The use of a rule-based method definition language has not been much investigated yet. The use of rule-based method definition languages has several advantages: declarativity, set-orientedness, the possibility of exploiting efficient query execution techniques. Another important advantage is that the behavior of methods expressed through rule-based languages can be clearly analyzed in terms of the set of updates performed.

Integrity constraints are declarative specifications of properties that must be satisfied by any database instance. In object-oriented database systems, constraints are typically used to restrict the admissible extent of a class, to impose restrictions on the state of individual objects, or to specify general laws that must be obeyed by sets of objects possibly belonging to different classes. Among the most common types of constraints we have already mentioned referential integrity and constraints associated with composite objects. Integrity maintenance in object-oriented database systems has so far received limited attention. Most mechanisms for constraint enforcement supports very limited types of constraints. This support consists usually in either forbidding or rolling back the operations that violate constraints.

There are presently two approaches to ensure that the database satisfies its constraints after certain actions. The first one is based on modifying methods in accordance to the specified integrity constraints. This direction is followed by the works of Schewe et al. [152, 153] and by the TM data model [21, 96]. Under this approach, integrity maintenance is obtained by an appropriate method modification. Those approaches, however, modify methods in different ways. In the approach of Schewe et al., given some arbitrary methods and some constraints, a *consistent specialization* of the methods with respect to the constraints is built. This consistent specialization is a new set of methods, performing the "same" actions, but without violating the constraints. This approach handles a limited set of constraints. By contrast, in the approach used in TM [96], for each update method an adorned version is created that checks the constraints of the class to which the method refers. For each method, an adorned method is created such that if it calls an update method, it calls its adorned version. Constraint checks are translated into preconditions on the actual data manipulation. Designers are allowed to define corrective actions upon expected constraint invalidation. This can be done on a per-method and per-constraint basis and may result in several adorned method versions, depending on application requirements.

The second approach relies on the use of active rules for integrity enforcement. Indeed, integrity constraints can be expressed by triggers, activated by any event likely to violate the constraint. The condition of the trigger tests for the actual occurrence of violations; if a violation is detected, then the action of the trigger either issues a rollback command or performs database manipulations for repairing constraint violations. This approach is the one taken by the Ode object-oriented database system [89], by the proposal for constraints

	Our model	GemStone	Iris	O2	Orion	TM	Ode	Cocoon	ODMG
Reference		[48]	[83]	[80]	[116]	[21]	[89]	[157]	[59]
User-defined value types	YES	NO	NO	YES	NO	YES	NO	NO	YES
Value collections	YES	NO	NO	NO	NO	NO[a]	NO	NO	NO
Class attributes and class methods	YES	YES	NO	NO	YES	YES	NO	NO	NO
Multiple inheritance	YES	NO	YES	YES	YES	YES	YES	YES	YES
Object migration	YES	NO	NO	NO	NO	NO	NO	YES	NO
Multiple class direct membership	YES	NO	YES[b]	NO	NO	NO	NO	NO	NO
Composite objects	YES	NO	NO	NO	YES	NO	NO	NO	NO
Referential integrity	YES	YES	NO	YES	NO	YES	YES	YES	YES[c]
Integrity constraints	YES	NO	L.F.	NO	NO	YES	YES	L.F.	NO
Intensionally defined attributes	YES	NO	YES	NO	NO	NO	NO	NO	NO
Declarative method implementation	YES	NO	YES	NO	NO	YES	NO	NO	NO
Triggers	YES	NO	NO	NO	NO	NO	YES	NO	NO
Views	YES	NO	YES[d]	NO	NO	NO[a]	NO	YES	NO

Legenda:

L.F. limited form.

[a] Value classes and views could be simulated in TM by using the module mechanism provided by this language.

[b] In Iris arbitrary types can be added to an object. To avoid conflicts, however,two different types of an object must not have different methods with the same name.

[c] Referential integrity is enforced for relationships, but it is not for attributes.

[d] Views in Iris are supported as functions implemented in OSQL which are not associated with a specific type.

Table 1.4: Comparison of our reference data model with some other object-oriented data models

in the O_2 object-oriented database system [132], by the works of Urban et al. [105, 167] and those of Ceri et al. [60, 84]. The set of triggers representing the constraints on the database must be however analyzed (either at design time or run-time) to ensure that they exhibit the desired behavior with respect to termination and determinism (uniqueness of final state) [84, 105].

Starting from this scenario the contribution of this thesis is to formally define an object-oriented data model with deductive and reactive capabilities. Most active object-oriented and deductive object-oriented proposals consider very simple object-oriented data models. Our proposal, by contrast, enriches a very rich object-oriented data model with support for active and deductive rules. Deductive rules are employed in our model for declaratively expressing integrity constraints, for specifying the implementation of derived attributes, for formulating queries, for specifying view population. A declarative set-oriented method definition language is also provided. A rich trigger language is finally provided for the model, and it truly integrates reactive capabilities with the basic modeling concepts of the object-oriented paradigm.

As we have already mentioned in the introduction, our reference model originates from the Chimera active, deductive, object-oriented data model. However it extends Chimera with direct support for referential integrity and composite objects, with views[9], and trigger overriding.

Table 1.4 compare the model considered in the thesis with some existing object-oriented data models. This comparison points out that our model covers a number of modeling features not covered by other models, since we consider a very rich and complex object-oriented data models, and we have developed a type system for the language. We would like to point out, moreover, that, though the model is a quite complex model, most of the features it supports are rather similar to features supported by modern object-relational DBMSs, like, for instance, DB2 [66] and Illustra [102]. In particular, those data models support: user-defined value types (primitive type extension in ORDBMSs); classes; triggers and integrity constraints. In addition, the model supports deductive rules, which allow to express constrained types, derived attributes, and views. However, these features are orthogonal, and thus non-redundant, with respect to the features listed above, supported by ORDBMSs.

[9]Chimera only supports object-preserving and set-tuple views (see Chapter 5.)

Chapter 2

A Formal Definition of the Reference Object-Oriented Data Model

In this chapter we formally introduce the reference object-oriented data model. This model [97] is a quite rich object-oriented data model, in that, besides supporting the notion of object, class, and inheritance typical of the object-oriented paradigm, it also supports user-defined named and constrained value types, extended types, composite objects and referential integrity, object migration and multiple class direct membership, class attributes and class methods, integrity constraints.

The chapter is organized as follows. Section 2.1 provides an informal overview of the data model. Sections 2.2 and 2.3 introduce types and values, respectively, whereas Sections 2.4 and 2.5 deal with classes and objects. Inheritance is considered in Section 2.6, whereas Section 2.7 introduces the notions of schema and database. Finally, Section 2.8 concludes the chapter.

2.1 Overview of the Data Model

The main concepts of the data model can be summarized as follows.

- **Objects:** are described by *attributes*, which can be *stored* or *derived* (e.g. defined by deductive rules), and are manipulated by *operations*, which can be *built-in* or *user-defined*. Each object is equipped with an object identifier (OID), which is unique within the entire database, is generated by the system upon the object creation, and never changes.

- **Values:** both *primitive* and *complex* values are supported. Complex values are built by using constructors like set, list, record.

- **Classes and inheritance:** objects with similar structure and behavior are grouped in classes. Classes are organized in inheritance hierarchies. Multiple inheritance is supported. An object can belong to several classes, even classes not related by inheritance, that is, multiple class direct membership is supported. Class-attributes and class-operations are supported, called c-attributes and c-operations respectively. They are not inherited by objects of the class, rather they are used to characterize classes as objects.

- **Value Types:** complex values are defined as instances of value types. Thus, value types provide the same function as concrete types commonly found in programming languages. A value type is *constrained*, if predicates are associated with the type definition specifying the legal values for the types, it is *unconstrained* otherwise. A value type is *extended* if its extent is explicitly defined (e.g. the user must specify the values belonging to the type), whereas the extent of other value types is implicitly defined. Therefore, extended value types provide the equivalent of enumerated domains.

- **Constraints:** constraints are predicates that must always be verified by the objects in the database. Constraints are expressed by deductive rules, in denial form, that is, by specifying conditions that cannot be satisfied by any legal database state. Some particular classes of integrity constraints have moreover an ad-hoc support in the model, namely those concerned with referential integrity and exclusiveness and dependency constraints for composite objects.

We now elaborate on some of the above concepts and present examples to illustrate them.

Objects and Values The model supports both values and objects. The main differences between those two notions can be characterized as follows. First, objects are abstract, non-symbolic elements of the application domain; values are symbolic, printable elements. A second important difference is related to the notion of *identity*. Objects are described by attributes; however, their identity does not depend on the attribute values. Changing the values of an object's attributes does not change the object identity. A primitive value is identified by the value itself, whereas a complex value is identified by the values of all its components. Therefore, changing a component in a complex value changes the "identity" of the value. Finally, objects can be manipulated by user-defined operations, whereas values can only be manipulated via pre-defined operations.

A number of constructors for building complex values, including the constructors set, list, and record, are provided. Constructors can be applied to: atomic values, complex values, object references. Therefore, constructors can be nested and a complex value may refer to an object.

Value types A value type definition consists of a structure specification, which is mandatory, and a constraint specification, which is optional. A constraint specification, in turn, consists of the *constraint interface*, that is, the constraint name and the type of results, and of the *constraint implementation*. Constraints are implemented as deductive rules. Those deductive rules have heads that may contain atoms with variables. Therefore, upon integrity checking a constraint may return values. Those return values are the values that have violated the constraint. Therefore, return values are used as the basis for explanation support.

The following examples illustrate value type definitions. Example (3) illustrates the notion of extended value type.

1. ```
   value  type complex:
          record-of (re-part: integer, im-part: integer)
   ```
   A value type, structured as a record of two components, is defined.

2. ```
   value  type dates:
          set-of(record-of (day:integer, month:integer, year:integer))
   ```
   A value type, structured as a set whose elements are records consisting of three components, is defined. This definition exemplifies nesting of constructors.

3. (a)
   ```
   value  type postalCode:  integer
          constraints improperCode (Code:  postalCode)
                      improperCode(Code) ← Code < 0;
                      improperCode(Code) ← Code > 9999
   ```
   (b)
   ```
   extended  type postalCode:  integer
             constraints improperCode (Code:  postalCode)
                         improperCode(Code) ← Code < 0;
                         improperCode(Code) ← Code > 9999
   ```
   In the above example, both definitions of `postalCode` have the same implementation for the constraint with name `improperCode`. Note that when definition (3.a) is used, the valid values are all 4-digit integers in the range (0, 9999). By contrast, when definition (3.b) is used, the valid values are all 4-digit integers in the range (0, 9999)

explicitly inserted into the type extent. Therefore, only the values that are inserted[1] by the users into the extended type `postalCode` are valid values for this type.

Therefore, the use of value types, extended value types and constrained value type supports: (*i*) domain definition through range specification; (*ii*) domain definition through explicit enumeration; (*iii*) definition of validity conditions for values. A large number of modeling capabilities for domains can thus be supported.

Classes A class definition consists of two components: the *signature*, specifying information for the use of the instances of the class; the *implementation*, providing an implementation for the signature. Furthermore, each class is associated with an *extent* collecting all instances of the class. The signature consists of a number of clauses, including the specification of: attributes, operations, constraints, class-attributes, class-operations, class-constraints and superclasses. The signature also specifies for each attribute whether the attribute is derived or not, and whether it is simple or composite. The implementation of a class must specify an implementation for all derived attributes, derived class-attributes, operations, class-operations, constraints and class-constraints that are specified in the signature. Note that the implementation of operations may be specified by an update rule, that is, a rule with update atoms. In Chapter 3 we will consider in detail the method definition language.

Classes are organized into inheritance hierarchies. A subclass may redefine an inherited feature but the redefinition is subjected to the following constraints:

- an attribute may be redefined in a subclass by specializing (or refining) its domain, that is, replacing its domain by a proper subtype (*covariant* redefinition);
- an operation may be redefined in a subclass by specializing its signature; signature specialization may be obtained by replacing at least one of the domains of the input parameters with a proper supertype or at least one of the domains of the output parameters with a proper subtype (*contravariant* redefinition for input parameters, *covariant* redefinition for output ones).

Our approach towards type checking is to privilege static type checking for what concerns method invocations; attribute type declarations, by contrast, are seen in our model as integrity constraints, and, thus, the correctness of the values assigned to them is checked at run-time. This is due to the fact that our model, besides allowing covariant redefinition of attribute domains, also supports constrained and extended types.

[1]An insert operation is provided for adding values into extended types. Such operation is not defined for other value types.

2.2 Types

In our model both the notions of type and of class are provided. The notion of type is similar to that of concrete (structural) type. The notion of class is similar to that of an abstract data type coupled with an extent[2]. Note, however, that full encapsulation of object structures is not enforced in our model.

Each class is related to a type (which we call type of the class) which describes the structure of the objects belonging to the class. In order to type variables that have to be instantiated with objects belonging to a given class, we allow the use of a class name as a type. A value type is a type in the previously introduced meaning, whereas an object type is the type corresponding to a class (the class name). We provide a uniform notion of type (regarding class names as types) to handle in uniform way type checking.

We postulate the existence of a collection of basic domains $D_1, \ldots, D_n$ which includes at least the types `integer`, `real`, `bool`, `character` and `string`. Furthermore, let $\mathcal{OI}$ denote a set of object identifiers, and $\mathcal{CI}$ a set of class identifiers. The elements of $\mathcal{OI}$ are identifiers of objects, used as a mean to refer to and distinguish objects in the system. By contrast, $\mathcal{CI}$ contains class identifiers, that is, class names. In the following we make use of a set of type names $\mathcal{TN}$, a set of attribute names $\mathcal{AN}$, a set of method names $\mathcal{MN}$ and a set of constraint names $\mathcal{PN}$. We assume all the introduced sets to be mutually disjoint[3].

In the following, we let i vary over $\mathcal{OI}$, c over $\mathcal{CI}$, Tn over $\mathcal{TN}$ and a over $\mathcal{AN}$. The following definitions formalize the various notions of types.

Definition 2.1 *(Predefined Basic Value Types). The set of predefined basic value types $\mathcal{BVT}$ is $\{D_1, \ldots, D_n\}$, that is, it is the collection of all the basic domains.* □

Class names can be used in definitions of types. This is due to the fact that attributes of types, structured as records, are allowed to have classes as domains. The definition of collections, structured as sets or lists, of instances of classes is also supported. Those collections are used, for example, to store query results.

The following definition states that each class name is a (object) type, thus introducing a uniform notion of type.

Definition 2.2 *(Object Types). The set of object types $\mathcal{OT}$ is defined as the set of class identifiers $\mathcal{CI}$.* □

[2]We refer to the notion of concrete and abstract types as used in Galileo [13].

[3]This assumption is only useful to simplify definitions and results, but it is used only at a theoretical level, while in the language it does not hold. One of the most remarkable characteristics of object-oriented data models is indeed the independence of each class definition from the others.

The distinction between value and object types is needed in order to distinguish types, whose instances are object identifiers in $\mathcal{OI}$ (namely values used to identify objects), from types, whose instances are pure values. The distinction between objects and values is a rather subtle issue of object-oriented data models. According to Beeri [25], the difference between these two notions can be characterized as follows: values are used to describe other entities, whereas objects are the entities being described. That is, the information carried by a value is the value itself, whereas the relevant information about objects is carried in the relationships they have with other objects and values. As a consequence, objects have an identifier and a state, which is a value, representing the information carried by the object. By contrast, the identifier of a value is the value itself. Therefore we may say that value types are types whose instances do not have an explicit identifier, whereas object types are types whose instances have explicit identifiers.

Definition 2.3 *(Value Types). The set of value types $\mathcal{VT}$ is inductively defined as follows*

- *the predefined basic value types are value types ($\mathcal{BVT} \subseteq \mathcal{VT}$);*
- *if T is a value type or an object type then*

  $list\text{-}of(T)$ *and*

  $set\text{-}of(T)$

  are value types called (predefined) structured value types, respectively indicated as list type and set type;
- *if $T_1, \ldots, T_n$ are value types or object types and $a_1, \ldots, a_n$ are distinct labels in $\mathcal{AN}$*[4]*, then*

  $record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$

  is a value type called (predefined) structured value type, indicated as record type;
- *if T is a value type (either basic or structured), Tn is a type name in $\mathcal{TN}$, and Tn is declared as a name for T through a declaration*

  $Tn : T$

  then Tn is a value type called user-defined (named) value type. $\square$

Example 2.1 *The following are examples of value types:*

[4]Note that the possible labels in record types are attribute names.

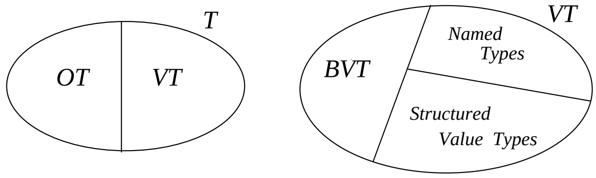

Figure 2.1: Types

```
integer
boolean
set-of(character)
list-of(boolean)
record-of(re:real, im:real)
set-of(record-of(day:integer, month:integer, year:integer))
```

`complex` *and* `dates`*, introduced by the following type declarations, are (named) value types:*

```
complex:  record-of(re:real, im:real)
dates:  set-of(record-of(day:integer, month:integer, year:integer)).
```

Finally, if `person` *belongs to* $\mathcal{CI}$*, then*

```
set-of(person)
record-of(n:string, s:person)
```

are (structured) value types. $\diamond$

According to the above definition, the set of predefined value types is an infinite set. Note, however, that each schema contains only a finite subset of predefined value types, that is, those types that are named or used somewhere in the schema. Therefore a schema is always finite.

The set of types $\mathcal{T}$ is thus defined as the union of value types $\mathcal{VT}$ and object types $\mathcal{OT}$. A type is therefore either a value type or an object type. We provide the notion of type because in some situations we want to talk of "type" disregarding whether it is a value type or an object type. Figure 2.1 depicts types, emphasizing that the set of object types and the set of value types partition the set of types, that is, they are disjoint and their union is the set of types $\mathcal{T}$.

Example 2.2 *Consider the class names* `person` *and* `department`*, both belonging to* $\mathcal{CI}$*. Then the types* `person` *and* `department` *are examples of object types, whereas the types*

```
record-of(name:string, dep:department)
list-of(person)
```

are value types. $\diamond$

Type names are assigned only to value types, not to object types. The same name cannot be assigned to two different types. That is, a name introduced by a declaration of the form $Tn : T$ (according to the last item in Definition 2.3) cannot be redeclared as $Tn : T'$. This condition ensures that the function *type*, defined below, is really a function, that is, the value type associated with a name is unique. Note that named types, basic value types and structured value types are disjoint sets, and they partition the set $\mathcal{VT}$ of value types. Each type in $\mathcal{VT}$, indeed, is either a named type, or a basic value type, or a structured type. As we will discuss in Section 2.6 inheritance relationships link named types and basic value types or named types and structured value types. By contrast, object types and value types constitute disjoint inheritance hierarchies.

Named value types are introduced as a mean to bind a type name to a value type. This mechanism is useful for future references to the, possibly structured, type and for type renaming. Note, however, that a name is assigned to a type only to simplify references to that type and that we have complete equivalence and substitutability of the type with respect to the name. Therefore, we disregard type names when considering type equivalence and compatibility. For this purpose, a (total) function $type : \mathcal{T} \rightarrow \mathcal{T}$ is defined, that for each type T, returns the type obtained from T by substituting each type name with the value type identified by the name. This function only affects value types. However, it has been defined on all types because object types may be components of value types.

Definition 2.4 *Let $T \in \mathcal{T}$ be a type. Then*

$$type(T) = \begin{cases} T & \text{if } T \in \mathcal{BVT} \text{ or } T \in \mathcal{OT} \\ list\text{-}of(type(T')) & \text{if } T = list\text{-}of(T') \\ set\text{-}of(type(T')) & \text{if } T = set\text{-}of(T') \\ record\text{-}of(a_1 : type(T_1), \ldots, a_n : type(T_n)) & \text{if } T = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n) \\ type(T') & \text{if } T \in \mathcal{TN} \text{ and it has been defined} \\ & \text{ by a declaration } T : T' \end{cases}$$

$\square$

The function has been defined by induction on the type structure of T by recursively substituting each type name in a value type with the value type identified by the name (also as component of structured types) and to deal also with named types defined in terms of other named types.

Example 2.3 *Consider the following type definitions.*

```
nat:  integer
pos:  integer
np-rec:  record-of(a:nat, b:pos)
s:  set-of(np-rec)
```

Then

$type$(`nat`) = $type$(`pos`) = `integer`
$type$(`np-rec`) = `record-of(a:integer, b:integer)`
$type$(`s`) = `set-of(record-of(a:integer, b:integer))`
$type$(`list-of(np-rec)`) = `list-of(record-of(a:integer, b:integer))`.

$\diamond$

2.2.1 Constrained Types

In addition to user-defined value types introduced as synonyms for (and abbreviations of) predefined types, the model provides the possibility of defining types denoting a proper subset of the set of values of a user-defined value type. In such cases, types are restricted by means of constraints expressing conditions on the values of the types. Types, defined by restriction on other types, are called *constrained types*. A constrained user-defined value type is a named type for which not only a structure but also a constraint is specified. Constraints associated with constrained types are expressed as deductive rules, whose definition is presented in Chapter 3. Intuitively, a rule expressing a constraint is a deductive rule containing in its head an atom and whose body consists of a conjunction of atoms or negated atoms. Atoms are built using predicate symbols. A number of predefined predicate symbols exist, including the equality and set/list membership. Rules are implicitly universally quantified as in Datalog. Rules used to express constraints for a constrained type have a constraint atom in their head, where a constraint atom is an atom built by applying a constraint name to a single argument whose type is the value type itself.

Definition 2.5 *(Constrained User-defined Types). If $Tn \in \mathcal{TN}$ is a type name, $T \in \mathcal{VT}$ is a value type, R is a set of deductive rules and Tn is declared as a name for the type whose structure is T and whose constraint is R, by a declaration*

Tn : T `constraint` R

then Tn is a constrained user-defined type. In the following struct(Tn) denotes the value type T (which is the structural component of the type, describing the structure of type elements) and constr(Tn) denotes the set of rules R. $\square$

Rules in R express a condition that must not be verified by the values of the type. Constraints are thus expressed in denial form. They state what should not be the case for any legal database state, rather than expressing invariant conditions of the database. In case of constraint violation, the parameter(s) of the constraint are bound to values characterizing the "violated" instance of the constraint.

Example 2.4 *Consider the type name* `postalCode`*, the basic value type* `integer` *and the following set R of rules:*

$$\{ \texttt{improperCode(Code)} \leftarrow \texttt{Code} < \texttt{0}, \\ \texttt{improperCode(Code)} \leftarrow \texttt{Code} > \texttt{9999} \}$$

Then the type declaration `postalCode`*:* `integer constraint` *R defines* `postalCode` *to be a constrained value type.* $\diamond$

Let $\mathcal{CT}$ denote the set of constrained user-defined types. Constrained user-defined value type are a special case of named types. Indeed, with a type name not only a value type is associated but also a constraint which expresses restrictions on the value type. Thus

$$\mathcal{CT} \subseteq \mathcal{TN} \subseteq \mathcal{VT} \subseteq \mathcal{T}.$$

Note that, unlike named value types, constrained types are not equivalent to the type structure assigned to the name, rather they are a proper subtype of that type. Indeed the set of values of a constrained type is a subset of the set of values of the type structure assigned to that type. Thus, when talking of constrained types the complete substitutability of the type with respect to the name is lost. By contrast, a constrained type is a subtype of the value type that describes its structure, in that all the legal values of the former are also legal values for the latter. Consider for example the value type

`record-of(day:integer, month:integer, year:integer)`.

Now consider the constrained type `date`, such that

$$\begin{aligned} struct(\texttt{date}) &= \texttt{record-of(day:integer, month:integer, year:integer)} \\ constr(\texttt{date}) &= \{ \texttt{improperDate(Date)} \leftarrow \texttt{day} \leq \texttt{0}, \\ &\quad\;\; \texttt{improperDate(Date)} \leftarrow \texttt{day > 31}, \\ &\quad\;\; \texttt{improperDate(Date)} \leftarrow \texttt{month} \leq \texttt{0}, \\ &\quad\;\; \texttt{improperDate(Date)} \leftarrow \texttt{month > 12}, \\ &\quad\;\; \texttt{improperDate(Date)} \leftarrow \texttt{year} \leq \texttt{0} \} \end{aligned}$$

Obviously, each date is a legal value for the value type `record-of(day: integer, month: integer, year: integer)`, but the converse is not true. Thus, `date` is a subtype of that type.

Similarly, if we consider the constrained type `summerdate`, specified as

$$
\begin{array}{l}
\mathit{struct}(\texttt{summerdate}) = \texttt{date} \\
\mathit{constr}(\texttt{summerdate}) = \{\ \texttt{improperSDate(SDate)} \leftarrow \texttt{month < 6}, \\
\qquad\qquad\qquad\qquad\quad \texttt{improperSDate(SDate)} \leftarrow \texttt{month > 8}\ \}
\end{array}
$$

then `summerdate` is a subtype of `date`.

Therefore, a partial ordering on value types exists such that unconstrained value types are not comparable under the ordering. By contrast, each constrained type is related to an unconstrained type (of which it is smaller) which is the type that describes its structure (the type returned by the *type* function[5]). We formalize these notions in Section 2.2.3 when defining an ordering on types.

2.2.2 Extended Types

In addition to predefined and user-defined value types, the notion of extended type is provided. The notion of extended value type can be characterized as follows. Predefined and user-defined value types are considered as "abstract domains", in the sense that the set of instances of the type is never made explicit. An explicit enumeration of the individual instances of the type is not possible. By contrast, extended value types are used to control the extent of a user-defined value type by enumerating its values. When defining an extended value type not only a value type with the same name and definition is generated, but also the storage for an explicit extent is allocated. Thus extended value types are a notion rather similar to that of enumeration types of programming languages (e.g. Pascal) but with two differences: (i) in an extended value type not only the explicit enumeration is provided but also a value type that specifies the structure of values of the (explicit) extent; (ii) the extent of an extended value type is not fixed, rather it can be modified. Constraints may be defined on extended value types as well, thus restricting their explicit extents.

Definition 2.6 *(Extended Value Type). Let $ET \in \mathcal{TN}$ be a type name, $T \in \mathcal{VT}$ be a value type, S be a set of values of type T, let finally T and S be associated with ET by a declaration*

[5] The *type* function of Definition 2.4 is easily extended to constrained types by defining $type(T) = type(struct(T))$ whenever $T \in \mathcal{CT}$.

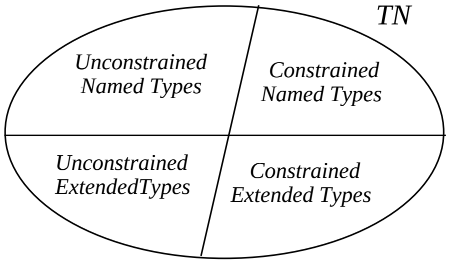

Figure 2.2: Named types

$ET : T$ `ext` S

then ET is an extended value type. If type T is a constrained type, then ET is said constrained, it is said unconstrained otherwise. In the following struct(ET) denotes the value type T (which is the type describing the structure of the extended type) and ext(ET) denotes the set S which maintains the explicit, time-varying extent of type ET. $\square$

Example 2.5 *Consider the type name* `colour` $\in \mathcal{TN}$*, the basic value type* `string` *and the set* $S = \{$`green, blue, red, yellow`$\}$*. Then the type declaration* `colour`*:* `string ext` S *defines* `colour` *as a (unconstrained) extended value type.*

Consider now the type name `realpostalcode` $\in \mathcal{TN}$*, the value type* `postalcode` *of Example 2.4 and the set* $S = \{$`5574, 7982, 6827, 1343`$\}$*. Then the type declaration* `realpostalcode` *:* `postalcode ext` S *defines* `realpostalcode` *as a (constrained) extended value type.* $\diamond$

Let $\mathcal{ET}$ denote the set of extended value types. The inclusion relationships

$$\mathcal{ET} \subseteq \mathcal{TN} \subseteq \mathcal{VT} \subseteq \mathcal{T}$$

hold. Extended value types are types rather than classes, though they have an explicitly enumerated collection of instances, that is, an explicit extent, since their instances are values, and not objects. They may be seen also as a sort of enumeration types.

The partitioning of named value types $\mathcal{TN}$ is depicted in Figure 2.2. Note that the sets of constrained extended value types, unconstrained extended value types, constrained value types and unconstrained value types are disjoint sets, but, however, inheritance relationships can exist between types in different sets, as we will discuss in Section 2.2.3.

Note that the extent S of an extended value type is not a fixed set, rather it can be updated by means of appropriate operations. Therefore, an extended value type could be regarded as a class (cf. Section 2.4) in that it consists of a type, an explicit extent and two predefined update operations. However, user-defined methods and other features of object classes are not supported for extended value types. This is the reason why we have preferred to refer to them as types.

Note that we have subtype relationships when considering extended value types, as well. Indeed, if an extended value type ET is defined in terms of a value type T, then all the elements of ET are elements of T, while the converse is not true (unless $ext(ET)$ coincides with the whole extension of type T). Thus, the extended value type is a subtype of the value type which specifies its structure, that is, $struct(ET)$. Referring to the example above, `colour` is a subtype of the type `string`, while `realpostalcode` is a subtype of the type `postalcode`.

We remark that we cannot define extended value types in terms of extended value types, that is, $\forall\ ET \in \mathcal{ET} : struct(ET) \in \mathcal{VT} \setminus \mathcal{ET}$, thus no subtype relationships hold among extended value types.

2.2.3 Subtype Relationship

As we have already mentioned, when regarding class names as types and when considering constrained types and extended value types, type extensions are not disjoint due to ISA relationships among classes and among types. Therefore, two different types may have a common supertype. In such a case, values, instances of those types, can be regarded as values of their most specific common supertype, and therefore these values are compatible.

First of all, we address the issue of type equality. Two types are equal either if they are identical or if they are record types obtained by permuting their fields. Indeed, the order of fields in record types is immaterial. Therefore the value type

$$record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$$

is equal to the type

$$record\text{-}of(a_{j(1)} : T_{j(1)}, \ldots, a_{j(n)} : T_{j(n)})$$

where $j : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ is a bijective function. Two named types are equal only if they have the same name.

Inheritance relationships among classes are described by an ISA hierarchy established by the user. This ISA hierarchy represents which classes are subclasses of (inherit from) other classes. This information is expressed as a function

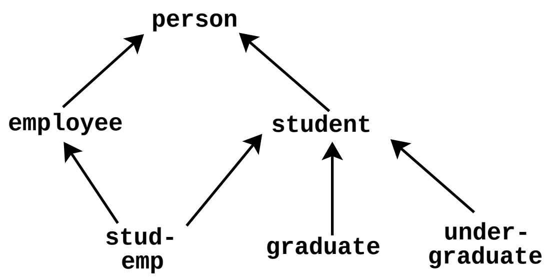

Figure 2.3: Subtype relationship among object types of Example 2.6

$$ISA : \mathcal{CI} \rightarrow 2^{\mathcal{CI}}$$

such that for each $c \in \mathcal{CI}$, $ISA(c)$ is the set of the (direct) superclasses of c. The function $ISA^* : \mathcal{CI} \rightarrow 2^{\mathcal{CI}}$ is the transitive closure of the subclass relationship, that is, it returns all the superclasses (also the indirect ones) of a given class.

Function ISA^* is defined as follows:

$$ISA^*(c) = \begin{cases} \emptyset & \text{if } ISA(c) = \emptyset \\ \cup_{c' \in ISA(c)} ISA^*(c') & \text{otherwise} \end{cases}$$

Example 2.6 *Referring to the inheritance hierarchy in Figure 2.3 the ISA relationships are the following:*

$ISA(\texttt{person}) = ISA^*(\texttt{person}) = \emptyset$
$ISA(\texttt{employee}) = ISA^*(\texttt{employee}) = \{\texttt{person}\}$
$ISA(\texttt{student}) = ISA^*(\texttt{student}) = \{\texttt{person}\}$
$ISA(\texttt{stud-emp}) = \{\texttt{employee}, \texttt{student}\}$
$ISA^*(\texttt{stud-emp}) = \{\texttt{employee}, \texttt{student}, \texttt{person}\}$
$ISA(\texttt{graduate}) = ISA(\texttt{under-graduate}) = \{\texttt{student}\}$
$ISA^*(\texttt{graduate}) = ISA^*(\texttt{under-graduate}) = \{\texttt{student}, \texttt{person}\}$.

$\diamond$

Besides the ISA ordering on classes, and thus on object types, another ordering must be considered, namely the ordering on value types due to constrained value types and to extended value types. Consider the function

$$ISA_V : \mathcal{VT} \rightarrow \mathcal{VT}$$

that, given a constrained value type or an extended value type, returns the (possibly constrained) value type in terms of which it has been defined (which is its parent in the ISA hierarchy). Formally:

- if $T \in \mathcal{CT}$ is a constrained type, then $ISA_V(T) = struct(T)$;
- if $T \in \mathcal{ET}$ is an extended type, then $ISA_V(T) = struct(T)$;
- $ISA_V(T) = \emptyset$ otherwise, that is, if $T \in \mathcal{VT} \setminus (\mathcal{CT} \cup \mathcal{ET})$.

Let ISA_V^* denote the transitive closure of the ISA_V relationship. We remark that our ISA_V^* hierarchy is such that $T \in ISA_V^*(T') \Rightarrow type(T) = type(T')$.

The subtype relationship $\leq_T$ is defined as follows. Note that the subtype relationship for basic types is the identity.

Definition 2.7 *(Subtypes). Given $T_1, T_2 \in \mathcal{T}$, T_2 is a subtype of T_1 (denoted as $T_2 \leq_T T_1$) if and only if one of the following conditions holds:*

- $T_1 = T_2$;
- $T_1 \in ISA^*(T_2)$;
- $T_1 \in ISA_V^*(T_2)$;
- *T_1 and T_2 are non extended unconstrained value types (that is, $T_1, T_2 \in \mathcal{VT} \setminus (\mathcal{CT} \cup \mathcal{ET})$) and $type(T_1) = type(T_2)$;*
- $T_2 = set\text{-}of(T_2')$, $T_1 = set\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
- $T_2 = list\text{-}of(T_2')$, $T_1 = list\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
- $T_1 = record\text{-}of(a_1 : T_1', \ldots, a_n : T_n')$, $T_2 = record\text{-}of(a_1 : T_1'', \ldots, a_n : T_n'')$ *and for each* i, $1 \leq i \leq n$, $T_i'' \leq_T T_i'$. □

Example 2.7 *Consider the set of object types* {`person`, `student`, `employee`, `stud-emp`, `graduate`, `under-graduate`} *with the ISA hierarchy of Example 2.6. Consider the set of named types* {`date`, `summerdate`, `postalcode`, `realpostalcode`, `colour`} *such that (as illustrated in Figure 2.4):*

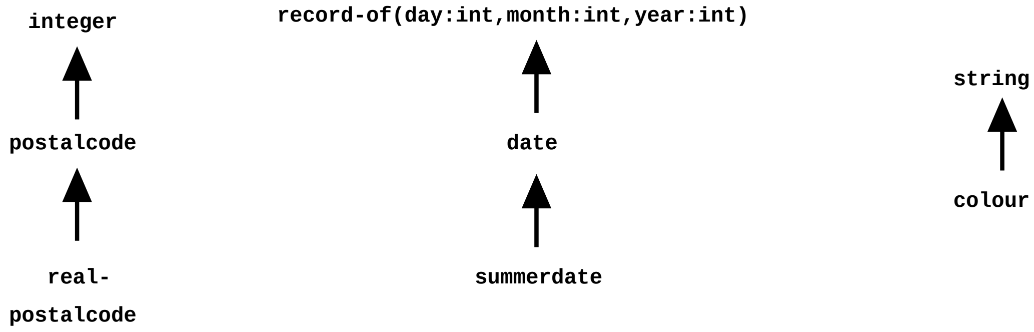

Figure 2.4: Subtype relationship among value types of Example 2.7

$ISA^*_V(\texttt{date}) = \{\texttt{record-of}(\texttt{day} : \texttt{integer}, \texttt{month} : \texttt{integer}, \texttt{year} : \texttt{integer})\}$
$ISA^*_V(\texttt{summerdate}) =$
$\{\texttt{date, record-of}(\texttt{day} : \texttt{integer}, \texttt{month} : \texttt{integer}, \texttt{year} : \texttt{integer})\}$
$ISA^*_V(\texttt{colour}) = \{\texttt{string}\}$
$ISA^*_V(\texttt{postalcode}) = \{\texttt{integer}\}$
$ISA^*_V(\texttt{realpostalcode}) = \{\texttt{postalcode}, \texttt{integer}\}.$

Then, the following subtype relationships hold:

$\texttt{graduate} \leq_T \texttt{student} \leq_T \texttt{person}$
$\texttt{list} - \texttt{of}(\texttt{stud} - \texttt{emp}) \leq_T \texttt{list} - \texttt{of}(\texttt{student}) \leq_T \texttt{list} - \texttt{of}(\texttt{person})$
$\texttt{summerdate} \leq_T \texttt{date}$
$\texttt{date} \leq_T$ `record-of(day:integer, month:integer, year:integer)`
`record-of(a:colour, b:postalcode)` $\leq_T$ `record-of(a:string, b:integer)`.

$\diamond$

Finally, if T_1, T_2 are types in $\mathcal{T}$ and $T_1 \leq_T T_2$ and $T_1 \neq T_2$ hold, T_2 is a *proper subtype* of T_1 (denoted as $T_2 <_T T_1$).

The set $\mathcal{T}$ of types with the ordering $\leq_T$ is a poset. Indeed $\leq_T$ is a partial order: it can be easily checked that it is reflexive, antisymmetric and transitive. Given the poset $(\mathcal{T}, \leq_T)$ and a set $TS \subseteq \mathcal{T}$ we may consider the upper bound of TS (the type $T \in \mathcal{T}$ such that $\forall T' \in TS : T' \leq_T T$). Intuitively an upper bound of a set of classes is a class c superclass of all the classes in the set. Consider for instance the hierarchy depicted in Figure 2.3. The upper bound of the set $\{$`employee, person`$\}$ is `person`, as the upper bound of the set $\{$`employee`, `student`$\}$ and of the set $\{$`graduate`, `employee`$\}$. By contrast,

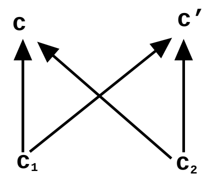

Figure 2.5: Inheritance hierarchy showing lub non-uniqueness

the set {`graduate`, `undergraduate`} has two upper bounds: `student` and `person`. This can be easily generalized to structured types with object types as components.

We may consider the notion of *least upper bound* (*lub*, $\sqcup$) in our poset. An upper bound T for a set TS is the least upper bound if and only if for all upper bounds T', $T \leq_T T'$ holds. Therefore, the *lub* of a set of types is the most specific among the supertypes of the types in the set. Referring to the above example the *lub* of {`graduate`, `undergraduate`} is `student`, since it is more specific than `person`.

Note, however, that because of the considered inheritance hierarchies (see Section 2.6), in which the existence of a root type is not imposed, the lub does not always exist. As a simple example, the lub of two types without common supertypes does not exist. Furthermore, the lub may not be unique. Consider for example the following situation. Let c and c' be two classes not related by the ISA hierarchy, and c_1 and c_2 be two classes both subclasses of both c and c' (this situation is illustrated in Figure 2.5). Then $c_1 \sqcup c_2$ would be either c or c'. We may construct a new type being the required (unique) lub, but this would imply adding a new class to the class hierarchy. Thus, we must look for a (already defined) lub class, which, indeed, may not be unique.

We remark that our lub construction is not based only upon typing aspects, rather it takes the user-defined ISA hierarchy into account. That is, it is not possible to compute the lub of two classes having comparable underlying types, but belonging to different class hierarchies. This is quite different from other well-known systems, especially those systems developing a class taxonomy in the context of knowledge representation systems [141]. However, we think that it is not correct to allow these kinds of situations, where values which are not related to each other (only have the same underlying type) are treated as *homogeneous*. Furthermore, we must take the user-defined ISA hierarchy into account to determine which methods are applicable. Thus, the lub construction in our model is stricter than the lub construction in other systems.

2.3 Values

In this section we introduce the set of legal values, and their corresponding types. First of all we introduce the set of legal values $\mathcal{V}$, and for each legal value we specify the type of which it is an instance. Note that a value may be an instance of different types.

Concerning the definition of legal values, it is important to note that oids in $\mathcal{OI}$ are handled as values. Indeed, an object identifier $i \in \mathcal{OI}$ stands for (i.e., it is a reference to) an object o, so it is a value of an object type in $\mathcal{OT}$. More precisely, the oids of objects that belong to a class c are legal values for the object type c. Furthermore, since an object, instance of a class c, is a member of all the superclasses of c, for a given class we consider as legal values of the corresponding object type, all the oids of objects that belong either to the class or to one of its subclasses. For example, consider a class `person` with a subclass `employee`, the oid of an object of class `employee` is a legal value for the object type `person`.

By contrast, when considering value types, the extensions of all the predefined basic value types are assumed to be disjoint. However, when a structured value type has as a component type an object type c, the oid of an object, instance of a subclass of c, is a legal value of this component. In this sense we deal with "*heterogeneous*" structures. Consider for example the type `set-of(person)`, being `person` a class with a subclass `employee`. Let i_1 be the oid of an instance of class `person` and i_2 the oid of an object of class `employee`. Then $\{i_1, i_2\}$ is a legal value for the type `set-of(person)`.

Let us consider a function π assigning an extent to each extended type and to each class. Actually π is a pair of functions (π_V, π_O) such that

$$\pi_V : \mathcal{ET} \rightarrow 2^{\mathcal{V}}$$

assigns a set of values to each extended value type in $\mathcal{ET}$ (*value assignment*), and

$$\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$$

assigns a set of object identifiers to each class in $\mathcal{CI}$ (*oid assignment*). Thus, if c is a class name in $\mathcal{CI}$, $\pi(c)$ is the set of the identifiers of objects instances of c. $\pi(c)$ is also referred to as the proper extent of class c. Given a class c, $\pi^*(c)$ denotes, by contrast, the global extent of class c, that is, the set of objects belonging to c either as instances or as members. $\pi^*(c) = \bigcup_{c' \leq_T c} \pi^*(c')$ holds, whereas $\pi(c) = \pi^*(c)$ for the classes without subclasses (leaves of the inheritance hierarchy).

belonging to c, both as instances and as members.

For each predefined basic value type D_i in $\mathcal{BVT}$ we postulate a non-empty extension, denoted by $dom(D_i)$. For example the extension of the basic value type `integer` is the set of integer numbers $\mathbb{Z}$, while the extension of the basic value type `bool` is the set containing the two boolean values, that is, $\{true, false\}$.

Definition 2.8 *(Values). The set of legal values $\mathcal{V}$ is defined as follows:*

- *null is a (basic) value of type T for each type $T \in \mathcal{T}$;*
- *for each predefined basic value type $D_i \in \mathcal{BVT}$ if $v \in dom(D_i)$ then v is a (basic) value of value type D_i (the extensions of the predefined basic value types are postulated);*
- *$i \in \mathcal{OI}$ is a (basic) value of object type c, $c \in \mathcal{OT}$, whenever $i \in \pi^*(c)$;*
- *let $v_1, \ldots, v_n$, $n \geq 0$ be values of types $T_1, \ldots, T_n$, respectively, such that a most specific common supertype T among $T_1, \ldots, T_n$ exists, then*

  $\{v_1, \ldots, v_n\}$

  is a (structured) value of value type set-of(T), called set value;
- *let $v_1, \ldots, v_n$, $n \geq 0$ be values of types $T_1, \ldots, T_n$, respectively, such that a most specific common supertype T among $T_1, \ldots, T_n$ exists, then*

  $[v_1, \ldots, v_n]$

  is a (structured) value of value type list-of(T), called list value;
- *if $v_1, \ldots, v_n$, $n \geq 0$ are values, and each v_i $(1 \leq i \leq n)$ has type T_i, and $a_1, \ldots, a_n \in \mathcal{AN}$ are distinct labels, then*

  $(a_1 : v_1, \ldots, a_n : v_n)$

  is a (structured) value of value type record-of$(a_1 : T_1, \ldots, a_n : T_n)$, called record value[6]*.* □

Example 2.8 *Provided that i_1, i_2, i_3 are oids in $\mathcal{OI}$, such that i_1 belongs to the extent of class* `department` *and i_2, i_3 belong to the extent of class* `person`*, examples of values are the following.*

- `1`, `27`, `342` *are values of type* `integer`*;*
- *$true$ and $false$ are values of type* `boolean`*;*

[6]We recall that for records we have the permutability of fields, that is, a value $(a_1 : v_1, \ldots, a_n : v_n)$ is identical to a value $(a_{j(1)} : v_{j(1)}, \ldots, a_{j(n)} : v_{j(n)})$ with $j : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ bijective function.

- {`'a'`, `'k'`, `'z'`} *is a value of type* `set-of(character)`*;*

- $[true, false, false, true]$ *is a value of type* `list-of(boolean)`*;*

- `(re: 8.24, im: 2.18)` *is a value of type* `record-of(re:real, im:real)`*;*

- {`(day:8, mont:10, year:1969)`, `(day:5, mont:10, year:1988)`} *is a value of type*
`set-of(record-of(day:integer, month:integer, year:integer))`*;*

- `(name:'john', dep:` i_1`)` *is a value of type*
`record-of(name:string, dep:department)`*;*

- $\{i_2, i_3\}$ *is a value of type* `set-of(person)`*.* $\diamond$

According to the above definition, and similarly to most object-oriented data models, oids are values. Indeed they are the values able to uniquely identify objects. References among objects and object sharing are supported by object identifiers. The state of an object, which is a value, may contain object identifiers, representing references to other objects. By contrast, oids, unlike other values, cannot be directly manipulated by the users. This is why oids are not included in terms (see Chapter 3). Indeed oids cannot appear in expressions written by the user, that is, constraints, queries or updates, because they cannot be explicitly used. Rather, only object variables, including special variables like `Self`, may appear in constraints, queries or updates. Oids can only be bound to variables through queries or retrieved by path expressions through navigation.

Finally note that according to Definition 2.8 above, for an oid i to be a legal value for an object type c, it is only required that i belongs to the extent of c. However, as we will discuss in what follows, if an oid $i \in \mathcal{OI}$ identifies an object belonging to a class c, that is, if i belongs to the extent of c, the state of the object identified by i is a legal value of the type which describes the structural component of class c. Therefore, the membership to the extent of a class ensures the structural consistency of the object with respect to the class. This condition will be formally defined in Section 2.5, after having introduced the definitions of class and object.

Definition 2.8, besides defining the set of legal values, implicitly states typing rules for values. The following definition explicitly states the typing rules implicit in Definition 2.8.

Definition 2.9 *(Typing Rules for Values). The typing rules for values are the following.*

$$\frac{}{null : T} \quad T \in \mathcal{T}$$

$$\frac{v \in dom(D_i)}{v : D_i} \quad D_i \in \mathcal{BVT}$$

$$\frac{i \in \pi^*(c)}{i : c} \quad i \in \mathcal{OI}, c \in \mathcal{CI}$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{\{v_1, \ldots, v_n\} : set\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{[v_1, \ldots, v_n] : list\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{(a_1 : v_1, \ldots, a_n : v_n) : record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)} \quad a_1, \ldots, a_n \in \mathcal{AN}$$

□

Note that the above typing rules do not determine a unique type for each value. The above typing rules are used to check whether a database state is structurally consistent, that is, if the state of each object meets the requirements of the structural part of its class definition. In particular, in order to ensure the correctness of the database, oids must be correctly typed. Indeed, if an oid is the value of an object attribute, the oid must meet the type requirements expressed by the attribute domain. Therefore, the oid must denote an object member of the class specified as attribute domain. In Chapter 3 we deal with the typing of expressions, giving typing rules for terms and addressing the problem of assigning a unique type to each object reference.

In case of a named type as domain, the compatibility of the value is checked first of all with respect to the value type describing its structure; then, in case of constrained types the constraint is checked, while in case of extended types the membership to the explicit extent is checked. These concepts are formalized by Definition 2.10 which exactly defines the set of legal values of each type.

We define now the extension, that is, the set of legal values, of each type. The extension of a type is used, in our model, to represent the denotational semantics of the type, that is, the set of values that may be legal instances of the type. The extension of a type depends on the explicit extent for such types, like extended value types and classes, that have one. To model these extents we consider function $\pi = (\pi_V, \pi_O)$ introduced above. To make explicit that the interpretation of types can only be given under an assignment π, we denote with $[\![T]\!]_\pi$ the extension of the type T under the assignment π.

Definition 2.10 *(Type Interpretation). $[\![T]\!]_\pi$ denotes the extension of the type T under the assignment π, defined as follows:*

- $\forall T \in \mathcal{T}$: $null \in [\![T]\!]_\pi$;
- $\forall D_i \in \mathcal{BVT}$: $[\![D_i]\!]_\pi = dom(D_i)$;
- $\forall c \in \mathcal{OT}$: $[\![c]\!]_\pi = \pi^*(c)$;
- $\forall T \in \mathcal{T}$: $[\![set\text{-}of(T)]\!]_\pi = 2^{[\![T]\!]_\pi}$, *where* 2^S *denotes the power set of the set* S;
- $\forall T \in \mathcal{T}$: $[\![list\text{-}of(T)]\!]_\pi = \bigcup_{n \in \mathbb{N}^+} \{[v_1, \ldots, v_n] \mid v_i \in [\![T]\!]_\pi \;\; \forall i, 1 \leq i \leq n\}$;
- $\forall T_1, \ldots, T_n \in \mathcal{T}$: $[\![record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)]\!]_\pi = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN}, v_i \in [\![T_i]\!]_\pi \;\; \forall i, 1 \leq i \leq n\}$ [7];
- $\forall Tn \in \mathcal{TN} \setminus (\mathcal{CT} \cup \mathcal{ET})$: $[\![Tn]\!]_\pi = [\![type(Tn)]\!]_\pi$;
- $\forall Tn \in \mathcal{CT}$: $[\![Tn]\!]_\pi = \{v \mid v \in [\![struct(Tn)]\!]_\pi,\; v \textit{ meets}^8\; constr(Tn)\}$;
- $\forall ET \in \mathcal{ET}$: $[\![ET]\!]_\pi = \pi^*(ET)$.

For a correct instantiation of an extended value type ET, *we require that*

$$\pi^*(ET) \subseteq [\![struct(ET)]\!]_\pi. \qquad \square$$

Informally, the extensions of predefined basic value types are postulated, the extension of classes and extended types is defined as their explicit extent, while the extension of structured types is defined quite intuitively in terms of the extension of the component types. The extension of a constrained value type is the set of the legal values for the type which describes the structure restricted to those that meet the population constraint. In fact some elements belonging to the extension of the type describing the structure may not meet the formula.

The following result holds, relating the extension of a type to its supertype extensions.

Theorem 2.1 *If* $T_2 \leq_T T_1$, *then for any assignment* π $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_\pi$ *holds.*

Proof Sketch: The theorem is proved by induction on the structure of types T_1 and T_2.

Basis: We distinguish four cases, corresponding to the basis of Definition 2.7 (first four items).

[7]Note that lists are a particular case of records in which labels are the positions in the list. So we may have stated $[\![list\text{-}of(T)]\!]_\pi = \{(1 : v_1, \ldots, n : v_n) \mid v_i \in [\![T]\!]_\pi \;\; \forall i, 1 \leq i \leq n\}$.

[8]Since $constr(Tn)$ is expressed in denial form by a set of rules R, v meets $constr(Tn)$ if v falsifies the bodies of all the rules in R.

- $T_1 = T_2$

  if $T_1 = T_2$ then, $\forall \pi \; [\![T_1]\!]_\pi = [\![T_2]\!]_\pi$ and thus, trivially, $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_\pi$;

- $T_1, T_2 \in \mathcal{OT}, T_1 \in ISA^*(T_2)$

  $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_\pi, \forall \pi$ follows from subset inclusion of extents (see extensional well-definedness of inheritance hierarchies, formally stated in Section 2.6) since, by Definition 2.10, $[\![T_i]\!]_\pi = \pi^*(T_i) = C_i.state.extent$, being $C_i.id = T_i$, $i = 1, 2$;

- $T_1, T_2 \in \mathcal{VT}, T_1 \in ISA_V^*(T_2)$

  two further cases can be distinguished:

  - $T_2 \in \mathcal{CT}$, and $T_1 = struct(T_2)$
    since by Definition 2.10 $\forall \pi \; [\![T_2]\!]_\pi = \{v \mid v \in [\![struct(T_2)]\!]_\pi$, v meets $constr(T_2)\}$, $[\![T_2]\!]_\pi \subseteq [\![struct(T_2)]\!]_\pi = [\![T_1]\!]_\pi$ follows;
  - $T_2 \in \mathcal{ET}$, and $T_1 = struct(T_2)$
    since, by Definition 2.10, $\forall \pi \; [\![T_2]\!]_\pi = \pi^*(T_2) \subseteq [\![struct(T_2)]\!]_\pi = [\![T_1]\!]_\pi$;

- $T_1, T_2 \in \mathcal{VT} \setminus (\mathcal{CT} \cup \mathcal{ET})$, and $type(T_1) = type(T_2)$

  since, by Definition 2.10, $\forall \pi \; [\![T_1]\!]_\pi = [\![type(T_1)]\!]_\pi = [\![type(T_2)]\!]_\pi = [\![T_2]\!]_\pi$;

Inductive step: We can distinguish three cases, corresponding to the inductive cases of Definition 2.7 (last three items).

- $T_1 = set\text{-}of(T_1')$, $T_2 = set\text{-}of(T_2')$, $T_2' \leq_T T_1'$

  by inductive hypothesis, $[\![T_2']\!]_\pi \subseteq [\![T_1']\!]_\pi, \forall \pi$. Thus, $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_t$ follows since $[\![T_i]\!]_\pi = 2^{[\![T_i']\!]_\pi}$, $i = 1, 2$ (see Definition 2.10);

- $T_1 = list\text{-}of(T_1')$, $T_2 = list\text{-}of(T_2')$, $T_2' \leq_T T_1'$

  by inductive hypothesis, $[\![T_2']\!]_\pi \subseteq [\![T_1']\!]_\pi, \forall \pi$. Thus, $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_\pi$ follows since $[\![T_i]\!]_\pi = \bigcup_{n \in \mathbb{N}^+} \{[v_1, \ldots, v_n] \mid v_j \in [\![T_i']\!]_\pi, j = 1, \ldots, n\}$, $i = 1, 2$ (see Definition 2.10);

- $T_1 = record\text{-}of(a_1 : T_1', \ldots, a_n : T_n')$, $T_2 = record\text{-}of(a_1 : T_1'', \ldots, a_n : T_n'')$, and $\forall i, 1 \leq i \leq n$: $T_i'' \leq_T T_i'$

  by inductive hypothesis, $\forall \pi$ and $\forall i, 1 \leq i \leq n$: $[\![T_i'']\!]_\pi \subseteq [\![T_i']\!]_\pi$. Thus, $[\![T_2]\!]_\pi \subseteq [\![T_1]\!]_\pi$ follows since $[\![T_1]\!]_\pi = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN}, \; v_i \in [\![T_i']\!]_\pi, \forall i, 1 \leq i \leq n\}$ and $[\![T_2]\!]_\pi = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN}, \; v_i \in [\![T_i'']\!]_\pi, \forall i, 1 \leq i \leq n\}$.

$\square$

We have no result on the converse of Theorem 2.1, because the ordering on types $\leq_T$ we consider keeps into account only the ISA relationships explicit in type definitions, but it does not consider those implicit in type constraints. Indeed, consider for example a constrained type $\mathtt{geq5}$ such that $struct(\mathtt{geq5}) = \mathtt{integer}$ and $constr(\mathtt{geq5}) = \{\mathtt{imprgeq5(X)} \leftarrow \mathtt{X} < \mathtt{5}\}$, and a constrained type $\mathtt{geq7}$ such that $struct(\mathtt{geq7}) = \mathtt{integer}$ and $constr(\mathtt{geq7}) = \{\mathtt{imprgeq7(X)} \leftarrow \mathtt{X} < \mathtt{7}\}$. Then, both $\mathtt{geq5} \leq_T \mathtt{integer}$ and $\mathtt{geq7} \leq_T \mathtt{integer}$ hold, while $\mathtt{geq5}$ and $\mathtt{geq7}$ are not related in the $\leq_T$ ordering. By contrast $[\![\, \mathtt{geq7} \,]\!] \subseteq [\![\, \mathtt{geq5} \,]\!]$, being each integer number greater or equal to 7 an integer number greater or equal to 5.

An immediate consequence of Theorem 2.1 is the following.

Corollary 2.1 *If $T = T_1 \sqcup T_2$ then for any assignment π $[\![\, T_1 \,]\!]_\pi \subseteq [\![\, T \,]\!]_\pi$ and $[\![\, T_2 \,]\!]_\pi \subseteq [\![\, T \,]\!]_\pi$.*

Proof Sketch: The result immediately follows from Theorem 2.1 since, by definition of lub, $T = T_1 \sqcup T_2$ implies $T_1 \leq_T T$ and $T_2 \leq_T T$. $\Box$

We have moreover the following soundness result for our typing rules.

Theorem 2.2 *(Soundness of Typing Rules for Values). Let T be the type deduced for a value v according to rules in Definition 2.9, then $v \in [\![\, T \,]\!]_\pi$.*

Proof Sketch: The theorem is proved by induction on the derivation tree for $v : T$.

Basis: We distinguish three cases, depending on the typing rule used in the tree.

- Rule 1:

  if we deduce type T for v by the first rule, then $v = null$. For the first item in Definition 2.10, $null \in [\![\, T \,]\!]_\pi$, $\forall T \in \mathcal{T}$;

- Rule 2 - $T \in \mathcal{BVT}$:

  $v : T$ can be deduced by the second rule if $v \in dom(T)$. Then, for the second item in Definition 2.10, $v \in [\![\, T \,]\!]_\pi$, since $[\![\, T \,]\!]_\pi = dom(T)$;

- Rule 3 - $T \in \mathcal{CI}$:

  $v : T$ can be deduced by the third rule if $v \in \pi^*(T)$. Then, for the third item in Definition 2.10, $v \in [\![\, T \,]\!]_\pi$, since $[\![\, T \,]\!]_\pi = \pi^*(T)$.

Inductive Step: We distinguish three cases, depending on the typing rule used in the tree.

- Rule 4 - $T = \mathit{set\text{-}of}(T')$:

  $v : T$ can be deduced by the fourth rule if $v = \{v_1, \ldots, v_n\}$ and $v_i : T_i, i = 1, \ldots, n$, has been deduced. Then, by inductive hypothesis, $v_i \in [\![T_i]\!]_\pi$. Since $T' = \bigsqcup_{i=1}^{n} T_i$, $[\![T_i]\!]_\pi \subseteq [\![T']\!]_\pi$ follows by Corollary 2.1, and thus $v_i \in [\![T']\!]_\pi$, $i = 1, \ldots, n$. Hence, $\{v_1, \ldots, v_n\} \in 2^{[\![T']\!]_\pi}$ and thus $\{v_1, \ldots, v_n\} \in [\![T]\!]_\pi$ for the fourth item in Definition 2.10;

- Rule 5 - $T = \mathit{list\text{-}of}(T')$:

  $v : T$ can be deduced by the fifth rule if $v = [v_1, \ldots, v_n]$ and $v_i : T_i, i = 1, \ldots, n$ has been deduced. Then, by inductive hypothesis, $v_i \in [\![T_i]\!]_\pi$. Since $T' = \bigsqcup_{i=1}^{n} T_i$, $[\![T_i]\!]_\pi \subseteq [\![T']\!]_\pi$ follows by Corollary 2.1, and thus $v_i \in [\![T']\!]_\pi$, $i = 1 \ldots n$. Hence, $[v_1, \ldots, v_n] \in [\![T]\!]_\pi$ for the fifth item in Definition 2.10;

- Rule 6 - $T = \mathit{record\text{-}of}(a_1 : T_1, \ldots, a_n : T_n)$:

  $v : T$ can be deduced by the sixth rule if $v = (a_1 : v_1, \ldots, a_n : v_n)$ and $v_i : T_i, i = 1, \ldots n$, has been deduced. Then, by inductive hypothesis, $v_i \in [\![T_i]\!]_\pi$. Hence, $(a_1 : v_1, \ldots, a_n : v_n) \in [\![T]\!]_\pi$ for the sixth item in Definition 2.10. $\square$

The previous theorem states the soundness of our typing rules. Our typing rules for values are not complete. More precisely, they are complete with respect to $\mathcal{T} \setminus \mathcal{TN}$. That is, we never deduce for a value a named type. The result that holds is that if $v \in [\![T]\!]$ and $T \in \mathcal{T} \setminus \mathcal{TN}$, then according to typing rules in Definition 2.9 we deduce type T for v. By contrast, the completeness with respect to the set of all types $\mathcal{T}$ does not hold. As an example we may consider the named type `pos`, defined as the set of natural numbers. According to our typing rules, it does not exist a value v for which we deduce type `pos`, whereas, obviously, $[\![\texttt{pos}]\!] = \mathbb{N} \neq \emptyset$. To obtain a complete type system, we must add rules that allow to deduce named types for values. Note however that a constrained type Tn can be deduced for a value v having type $\mathit{struct}(Tn)$ only if v meets $\mathit{constr}(Tn)$. That is, membership to the type extent is subjected to constraint verification.

Our solution is to keep an incomplete (with respect to named types) type system, and, when the domain for an attribute is a named type Tn, we check that the value for the attribute is a legal value for $\mathit{type}(Tn)$, and, as an additional constraint (that is, not related to typing), we require that the value belongs to the type interpretation $[\![Tn]\!]$.

2.4 Classes

A class has in our model both an intensional and an extensional function. Each class, therefore, besides having a signature, providing the information for the use of the class and its instances, and an implementation, specifying the meaning of class features, also has a state, containing the current values for class attributes and the current class extent. In the following we associate each class both with its proper extent, that is the set of objects belonging to the class as instances, and with its global extent, that is the set of objects belonging to the class as members. Note that the global extent can be derived as the union of the global extents of all the subclasses of the considered one. Proper and global extents coincide for classes with no subclasses. Class signatures and implementations will be discussed in detail in the following subsections.

Definition 2.11 *(Class). A class C is a tuple*

$$C = (id, sign, impl, state)$$

where:

- *$id \in \mathcal{CI}$ is the class identifier;*
- *$sign$ is the class signature (see Definition 2.12);*
- *$impl$ is the class implementation (see Definition 2.15);*
- *$state \in \mathcal{V}$ is a value containing the values for the class attributes plus two set values, containing all the objects belonging to C as members and as instances, respectively; it is a record value of the form*

  $$(a_1 : v_1, \ldots, a_n : v_n, extent : E, proper\text{-}extent : PE)$$

  where $a_1, \ldots, a_n$ are the attribute names of the c-attributes of C; E and PE are the sets of identifiers of objects belonging to class C. □

The attributes *extent* and *proper-extent* in the class state, denote the global and the proper extent of the class, respectively; therefore $PE \subseteq E$ ($E = PE$ if C is a leaf class in the inheritance hierarchy). The oid assignment $\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$, introduced in Section 2.2, is such that, for each class name c, $\pi_O(c) = C.state.proper\text{-}extent$, where C is the (unique) class such that $C.id = c$ (that is, the class identified by c).

Example 2.9 *An example of class is the following, where S and S' denote two oid sets.*

$id = \texttt{person}$
$state = (\texttt{averageAge} : 35, \texttt{lifeExpentancy} : 80, extent : S, proper\text{-}extent : S')$

and sign and impl are as specified in Example 2.10 and 2.13, respectively. $\diamond$

The *definitional component* of a class $C = (id, sign, impl, state)$ is $(id, sign, impl)$, that is, its state-independent components.

2.4.1 Class Signature

The signature of a class provides all the information for the use of the class and its instances. For each class, we maintain structural, behavioral and constraint information. For the structural part of a class, the signature must contain name and domain of each attribute, and whether the attribute is extensional or derived, simple or composite. Dependency and exclusiveness constraints may be associated with composite attributes, whereas the action to be taken upon referential integrity violations may be associated with each attribute. For the behavioral part of a class the signature must contain name, input and output parameters of each method. The signature must contain similar information for the structure and behavior of the class itself, that is, information about class-attributes and class-methods. The signature must finally contain for each class the signature of constraints, both those related to class instances and those related to the class itself.

To provide a uniform modeling of instance features and class features we introduce the notion of *metaclass*. A metaclass is a special class, whose unique instance is a class. By introducing this concept we model in a uniform way attributes and c-attributes, operations and c-operations, constraints and c-constraints. A class is seen as an instance of the corresponding metaclass exactly in the same way an object is seen as an instance of a class. The structural component of class signatures and metaclasses are discussed in detail in the following subsections.

Definition 2.12 *(Class Signature). A class signature is a tuple*

$$(struct, beh, constr, mc)$$

where:

- *struct contains the information about the structure of the instances of the class, and it is a set, containing an item for each attribute of the class, of triples*

$$(a\_name, a\_dom, a\_type)$$

where

- $a\_name \in \mathcal{AN}$ *is the attribute name,*
- $a\_dom \in \mathcal{T}$ *is the attribute domain,*
- $a\_type \in \{ext, der, comp\}$ *is the attribute type, that is, whether it is extensional, derived or composite*[9]*;*

- *beh contains the information about the behavior of the instances of the class, and it is a set, containing an item for each method of the class, of pairs*

$$(op\_name, op\_sign)$$

where

- $op\_name \in \mathcal{MN}$ *is the method name,*
- *op_sign is the signature of the method, expressed as*

$$T_1 \times \ldots \times T_j \rightarrow T_{j+1} \times \ldots \times T_n$$

with $T_1, \ldots, T_n$ *types in* $\mathcal{T}$*, representing, respectively, the domains of the input parameters* $(T_1, \ldots, T_j)$ *and of the output parameters* $(T_{j+1}, \ldots, T_n)$ *of the method*[10]*;*

- *constr contains information about the constraints on the instances of the class, and it is a set, containing an item for each constraint of the class, of pairs*

$$(con\_name, con\_sign)$$

where

- $con\_name \in \mathcal{PN}$ *is the constraint name,*
- *con_sign is the signature of the constraint, expressed as*

$$T_1 \times \ldots \times T_k$$

with $T_1, \ldots, T_k$ *types in* $\mathcal{T}$*, representing domains of the output parameter of the constraint*[11]*;*

---

[9] Derived composite attributes are not considered in our model.

[10] Note that both $j = 0$ (method without input parameters) and $j = n$ (method without output parameters) can hold.

[11] Note that the input parameter of the constraint is always an object of the class on which the method is defined.

- *mc is the identifier of the metaclass corresponding to class C, that is, the class of which C is instance.* $\square$

Example 2.10 *The signature of class* `person` *of Example 2.9 can be as follows.*

$struct$ = { (`name, string`, *ext*), (`vatCode, string`, *ext*), (`birthday, date`, *ext*), (`income, integer`, *ext*), (`age, integer`, *der*) }
beh = { (`changeIncome : integer` $\rightarrow$ `person`) }
$constr$ = { (`tooLowIncome : string`) }
mc = `m-person`. $\diamond$

The identifier of a class C denotes the object type corresponding to C. Such object type is the type of the identifiers of the objects of class C. A value type is implicitly associated with each class, representing the type of values that constitute the state of the class instances. Indeed, if a class C has as *struct* the set $\{(a_1, T_1, at_1), \ldots, (a_n, T_n, at_n)\}$, then each object instance of C must have as state a value of (record) type

$$record\text{-}of(a_1 : T_1, \ldots, a_n : T_n).$$

This type, which describes the structure of the objects of the class, is the *structural type* of the class, and it is denoted by $stype(C)$. The function $stype : \mathcal{CI} \rightarrow \mathcal{VT}$ is the function that given an identifier $c \in \mathcal{CI}$ returns $t \in \mathcal{VT}$ obtained as $stype(C)$ where C is the class identified by c, that is, the (unique) class such that $C.id = c$. For example, referring to the class of Example 2.9, $stype$(`person`) is `record-of(name:string, vatCode:string, birthday: date, income: integer, age: integer)`.

Given a class c, $A(c)$ denotes the set of attributes of that class. Furthermore, $dom(a, c)$, for $a \in A(c)$, denotes the domain of attribute a in class c. Similarly, for the behavioral component of classes, $M(c)$ denotes the set of methods of class c and $sign(m, c)$, for $m \in M(c)$, denotes the signature of method m in class c.

2.4.1.1 Metaclasses

For each class C, a metaclass MC is introduced, defined as follows. Each metaclass has a unique instance, that is, the class to which it corresponds. We remark that the notion of metaclass we refer to is different from the one intended in other object-oriented models and languages, such as Smalltalk, in that our metaclasses are not classes, and, thus, they cannot be instantiated multiple times. Note, however, that the notion of metaclass supported could be easily generalized by enriching the language with primitives for creating metaclasses and instantiating them.

Definition 2.13 *(Metaclass). A metaclass MC corresponding to a class C is a tuple*

$$MC = (id, struct, beh, constr)$$

where:

- $id \in \mathcal{CI}$ *is the identifier of the metaclass;*
- *struct contains the information about the structure of the class C, that is, its class attributes, and it is exactly like struct in Definition 2.12, considering class attributes instead of the instance ones, except for containing two additional extensional attributes: extent with domain set-of(c) and proper-ext with domain set-of(c)*[12]*;*
- *beh contains the information about the behavior of the class C, that is, its class methods, and it is exactly like beh in Definition 2.12, considering class operations instead of instance ones;*
- *constr contains the information about the constraints of the class C, that is, its class constraints, and it is exactly like constr in Definition 2.12, considering class constraints instead of instance ones.* □

Example 2.11 *The metaclass corresponding to class* `person` *of Example 2.9 is the following.*

id = `m-person`
struct = { (`averageAge, integer`, *ext*), (`lifeExpentancy, integer`, *ext*),
(*extent* : `set-of(person)`, *ext*), (*proper-extent* : `set-of(person)`, *ext*) }
beh = { (`changeLifeExpentancy` : `integer` → `m-person`) }
constr = { (`invalidLifeExpentancy` : `integer`) }.

◇

2.4.1.2 Constraints on Attributes

To allow the specification of constraints on attributes, each item in the structural component of a class signature is extended as follows:

- a field *a_excl* specifies the set of classes with respect to which the attribute is exclusive, the field can also specify that the exclusiveness is to be intended with respect to the attribute itself;

[12] We denote with c the identifier of the class C ($C.id$),that is, its corresponding object type.

- a field *a_dep* specifies that, upon the deletion of an object of the class, the (component) object which is the value for the attribute is also deleted, unless it is a component of some other object;

- a field *a_del* specifies the actions to be performed on the object upon the deletion of the object which is the value for the attribute.

Definition 2.14 *(Structural Component in a Class Signature). Each item in the struct component of a class signature is a tuple*

$$(a\_name, a\_dom, a\_type, a\_excl, a\_dep, a\_del)$$

where:

- a_name, a_dom, a_type *are as in Definition 2.12;*

- $a\_excl = (A, S)$ *where:*

  - $A \in \mathcal{BOOL}$ *is the exclusiveness scope: true denotes exclusiveness on the attribute, whereas* $false$ *denote exclusiveness;*
  - $S \in 2^{\mathcal{CI}}$[13] *is the set with respect to which exclusiveness is required;*

- $a\_dep \in \mathcal{BOOL}$*: true denotes that the attribute is dependent;*

- $a\_del \in \{Casc, Setn, Noact, Restr\}$ *specifies the action to be taken upon violation of referential integrity for the attribute: Casc = cascade, Setn = set_null, Noact = no_ action, Restr = restrict.* □

A derived attribute is an attribute whose value is not stored, rather it is computed by means of deductive rules specified in the class implementation. A composite attribute is an attribute to which a semantic "part-of" relationship is superimposed. Non-composite attributes are sometimes referred to as simple or weak attributes. For weak attributes, fields *a_excl* and *a_dep* always have $(false, \emptyset)$ and $false$ values, respectively, since no exclusiveness and dependency constraints can be associated with weak attributes.

Intuitively, exclusiveness with respect to a set of classes S states that two objects instances of classes in S cannot share a component, while exclusiveness on the attribute with respect to a set of classes S states that two objects instances of classes in S cannot have the same component as the value of the specified attribute. Obviously, for that notion to be

[13] Given a set S, 2^S denotes the powerset of set S.

meaningful, each class in S must have that attribute. We remark that exclusiveness is a stronger notion than exclusivess on the attribute.

The a_dep field specifies whether the deletion of the composite object must be propagated to the component objects. As discussed in Chapter 1, this automatic propagation is useful because it saves the application from having to explicitly delete all the component objects. Consider a class C with a composite dependent attribute a. Consider an object o belonging to class C and suppose o is deleted. If in the $C.sign.struct$ component for attribute a, $a\_dep = true$, upon the deletion of o, the value of attribute a of object o (that is, $o.v.a$) is deleted, too, unless it is referred by any other object.

The a_del field specifies how the deletion of an object affects the objects referencing it. Consider a class C, with an attribute a. Consider moreover an object o belonging to class C and the object o' value of attribute a of o. Then, the deletion of o' has different results depending on the value of the a_del field of the $C.sign.struct$ component for attribute a:

- if $a\_del = Casc$ the deletion of o' implies the deletion of o;
- if $a\_del = Setn$ the deletion of o' results in setting the value of attribute a of o to $null$;
- if $a\_del = Restr$ o' cannot be deleted;
- if $a\_del = Noact$ o' is deleted and the value of attribute a of o remaind undefined (dangling pointer).

Example 2.12 *Given two classes* `manager` *and* `employee`*, consider a class* `project`*, the struct component of whose signature contains*

$(\texttt{leader}, \texttt{manager}, comp, (true, \{\texttt{project}\}), false, Casc)$

$(\texttt{staff\_member}, \texttt{employee}, comp, (false, \emptyset), true, Restr)$.

A project is characterized by a leader and a staff member. The same manager cannot lead two different projects, however, a manager (which is also an employee) can be the leader and the staff member of a project (or of different projects), due to the fact that the exclusiveness of attribute `leader` *is restricted to the attribute itself. The deletion of a project does not impact the existence of its leader; by contrast, upon deletion of a manager the project he eventually leads is in turn deleted. By contrast, an employee can be a staff member of several projects. The deletion of a project also results in the deletion of its staff member, if the member is not the staff member, or the leader, of other projects. An employee cannot be deleted, finally, as long as he works for a project.* ◇

2.4.2 Class Implementation

In addition to a signature, classes have an implementation. Roughly speaking, the signature of a class is the definition of all the names and domains associated with that class, whereas the implementation provides the specification of the meaning of each concept associated with a class. These implementations are expressed by means of deductive (passive) rules or, in case of method implementation, of update operations constrained by a declarative formula. In a class implementation:

- (derived) attributes and c-attributes are implemented by means of deductive rules specifying the computation of values by means of a declarative expression;
- constraints and c-constraints are implemented by means of deductive rules associating with a parameterized constraint name a condition that should not hold in any state of the database;
- the implementation of an operation or c-operation is an update rule of the form

  $$condition \rightarrow action$$

  where the condition is a declarative expression, specifying a declarative control upon operation execution, while the operation code is a sequence of update primitives, object creation and deletion, object migration from one class to another, state changes, and extended value type modifications.

Definition 2.15 *(Class Implementation). Given a class C whose signature $C.sign = (struct, beh, constr, mc)$, an implementation for C consists of a set of deductive rules, specifying*

- *an attribute implementation for each derived attribute in struct;*
- *a constraint implementation for each constraint in constr;*

plus a set of update rules specifying an operation implementation for each operation in beh. □

Note that c-attribute, c-constraint and c-operation implementations constitute the meta-class implementation.

We impose some conditions to ensure that a class implementation is consistent with the related class signature. Intuitively, an implementation for each feature specified in the signature must be provided. In addition, type compatibility must be ensured between the types defined in the signature and those returned by the implementation. We deal with these issues in Chapter 3, after having introduced declarative expressions and their typing.

Example 2.13 *The following is an implementation for class* `person` *of Example 2.9.*

```
Self.age = X ← X = 1997 - Self.birthday.year
changeIncome(Amount):
  integer(New), New = Self.income + Amount →
  modify(person.income,Self,New)
tooLowIncome(N) ← Self.income < 5000, N = Self.name
```

The following is, by contrast, the implementation of metaclass `m-person` *of Example 2.11.*

```
Class.averageAge = Y ← integer(Y),
      Y = avg(X.age where person(X))
changeLifeExpectancy(Delta):
      integer(New), Delta < 10,
      New = Class.lifeExpectancy + Delta →
      modify(person.lifeExpectancy, Class, New)
invalidLifeExpectancy(I) ←
      I = Class.averageAge - Class.lifeExpectancy,
      abs(I) > 5
```

◇

2.5 Objects

An object is an *instance* of a class if this class is the most specific class in the hierarchy to which the object belongs. Whenever an object o is an instance of a class C then o is also a *member* of all the superclasses of C. The most specific class to which an object belongs is not necessarily the one in which the object has been created (the first argument of the `create` operation). This is due to the fact that the object may have migrated to another class, by an explicit migration operation (`specialize` or `generalize`).

In our model, an object does not necessarily belong to a unique most specific class. An object may indeed be created as an instance of a class and then specialized in two different subclasses of the original class. However, when considering objects with several most specialized classes new problems arise, because an object does no more have a single most specific class, rather it has a set of them. In such a case, an object takes the union of the features of all the classes to which it belongs. Conflicts among different definitions may however arise. In this section we discuss how these conflicts are solved for what concerns the structural component of objects, while conflicts concerning object behavior (that is, the problem of method dispatching) will be discussed in Chapter 6.

Let us first of all formalize the notion of most specific class.

Definition 2.16 *(Most Specific Class). Given a set of classes C, $ms(C)$ denotes the set $\{c \mid c \in C \wedge \nexists\, c' \in C, c \neq c'$, such that $c \in ISA^*(c')\}$. A class c is said a most specific class in C if $c \in ms(C)$.* □

Definition 2.17 *(Object). An object is a triple*

$$o = (i, v, CS)$$

where

- *$i \in \mathcal{OI}$ is the object identifier of o;*
- *$v \in \mathcal{V}$ is a value, called state of o;*
- *$CS \subseteq \mathcal{CI}$ is the set of most specific classes to which o belongs, that is $CS = \{c \mid c \in \mathcal{CI} \wedge i \in \pi(c)\} = ms(\{c \mid c \in \mathcal{CI} \wedge i \in \pi^*(c)\})$.* □

Each object is required to be an instance of (at least) a class; therefore, $o.CS \neq \emptyset$ must hold. Moreover, all the classes in $o.CS$ are most specific with respect to the others, that is $ms(o.CS) = o.CS$.

Example 2.14 *An example of object is the following, provided that $i_1 \in \mathcal{OI}$ and* `person` $\in \mathcal{CI}$.

$(i_1,$ (`name: ´john smith´`, `vatCode: ´666FF´`,
`birthday: (day:8, month:10, year:1969)`,
`income: 5000`, `age: 28`), { `person` })

◇

Note that the state v of an object is a value of a record value type. That is, v has the form $(a_1 : v_1, \ldots, a_n : v_n)$ with $a_1, \ldots, a_n \in \mathcal{AN}$. The function $\nu : \mathcal{OI} \to \mathcal{V}$, for each $i \in \mathcal{OI}$ simply returns $v \in \mathcal{V}$ such that $o = (i, v, CS)$ is an object in the database. Referring to the example above

$\nu(i_1) =$ (`name: ´john smith´`, `vatCode: ´666FF´`,
`birthday: (day:8, month:10, year:1969)`,
`income: 5000`, `age: 28`)

2.5.1 Object Consistency

Each object must be a consistent instance of all the classes to which it belongs. Therefore an object $o = (i, v, CS)$ must be a consistent instance of classes in CS[14]. We distinguish two kinds of consistency:

- *structural consistency* if the instance respects the structure of the class in which it is defined, that is, if the object state is a (record) value whose type is the type of the structural component of the class;
- *constraint consistency* if the instance satisfies the constraints of the class in which it is defined.

We first consider structural consistency. We determine for an object with multiple most specific classes, the state of the object, that is, which are the attributes of the object, and which are the proper domains for these attributes.

Roughly speaking, the state of an object belonging to several most specific classes is the union of the attributes in those classes. However, the sets of attributes in those classes may not be disjoint, that is, *name conflicts* may arise. To handle these situations we introduce the notion of *source* of an attribute. Intuitively, if an attribute belongs to the intersection of the attribute sets of two classes and it has in both classes the same source, that is, it is inherited by a common superclass, then the attribute is semantically unique, and thus the object must have a unique value for this attribute. If, by contrast, the attribute has different sources then the two attributes in the two classes have accidentally the same name, but represent different information, that must be kept in separate ways. Thus, the object may have two different values for the two attributes (a renaming policy is applied).

Example 2.15 *As an example of the first situation, consider the inheritance hierarchy of Figure 1.1 and suppose that an attribute* `spouse` *is present in class* `person`*. This attribute is inherited by all the subclasses of* `person`*. Consider now an object which is both a noble and a female, then the attribute* `spouse` *belongs to the intersection of the attribute sets of* `noble` *and* `female`*, and it has in both classes the same source (*`person`*), and thus the object has a single attribute* `spouse` *(this corresponds to the intuition that the information carried out by the two attributes in the two different classes is the same).*

As an example of the second situation, consider the inheritance hierarchy composed by a class `person` *and two subclasses* `student` *and* `employee`*. Suppose that in class* `student` *a new attribute* `code` *is defined, representing the number of the student in the university. Suppose moreover, that in class* `employee` *a new attribute* `code` *is defined, representing*

[14]This ensures also the consistency with respect to all the superclasses of classes in CS.

the number of the employee in the welfare office. Consider now an object which is both an employee and a student, then the attribute `code` *belongs to the intersection of the attribute sets of* `employee` *and* `student`*, but it has different sources in the two classes: it has source* `employee` *in class* `employee`*, while it has source* `student` *in class* `student`*. Thus the object have two different attributes, say* `student-code` *and* `employee-code` *(this corresponds to the intuition that the information carried out by the two attributes in the two different classes is different).* $\diamond$

We now formalize the above notions. We recall that, given a class c, $A(c)$ denotes the set of attributes of that class. Moreover, $A^{+}(c) = \{a \mid a \in A(c) \wedge \nexists\ c', c \neq c', c' \in ISA^{*}(c)$ such that $a \in A(c')\}$ denotes the set of proper (that is, not inherited) attributes of class c.

First, we define the notion of *source* of an attribute. Intuitively, the source of an attribute a in a class c is the most general superclass of c in which the attribute a is defined. Thus, it is the class from which c has inherited attribute a. However, because of multiple inheritance, there may be two different superclasses of c, not related by the ISA relationship, in which attribute a is defined. In such a situation, we impose that in the definition of class c an explicit specification (through a *qualification mechanism*) is given of the class from which the attribute is inherited. Thus, each class inherits each attribute from one of its superclasses, the most general in the path specified by the qualification mechanism. This unique class is called source of the attribute in the class.

A function $inh^{a} : \mathcal{CI} \to \mathcal{CI}$ is defined for each class c, $c \in \mathcal{CI}$, such that $a \in A(c) \setminus A^{+}(c)$. Given a class c, function inh^{a} returns the direct superclass of c from which c inherits attribute a. The definition of this function is as follows:

- if a unique class c' exists such that $c' \in ISA(c), a \in A(c')$ then $inh^{a}(c) = c'$;
- otherwise (if more than one class c' exists such that $c' \in ISA(c)$ and $a \in A(c')$) the resolution of name conflicts for multiple inheritance requires that in c definition an explicit qualification mechanism is exploited to state from which class attribute a is inherited; if c^{q} is the qualified class, then $inh^{a}(c) = c^{q}$.

We remark that, according to its definition, inh^{a} is a monotonic (strictly increasing) function, in that $\forall c \in \mathcal{CI}$ if $inh^{a}(c)$ is defined $inh^{a}(c) \in ISA(c)$ and thus $c <_{T} inh^{a}(c)$ holds.

Let $inh^{a^{*}} \subseteq \mathcal{CI} \times \mathcal{CI}$ denote the relation corresponding to the transitive closure of the relation associated with function inh^{a}[15]. If $inh^{a^{*}}$ contains two pairs (c, c') and (c, c'') then either $c' \in ISA^{*}(c'')$ or $c'' \in ISA^{*}(c')$ holds. Thus, $inh^{a^{*}}$ contains ascending paths of the inheritance hierarchy. Moreover, given a set of classes CS, let $mg(CS)$ denote the set $\{c \mid c \in CS \wedge \nexists\ c' \in CS, c' \neq c$ such that $c' \in ISA^{*}(c)\}$, that is the set of *most general* classes in C.

[15]The relation associated to function inh^{a} is the set $\{(c, c') \mid c, c' \in \mathcal{C} \wedge inh^{a}(c) = c'\}$.

Definition 2.18 *(Source). The source of an attribute* $a \in \mathcal{AN}$ *in a class* $c \in \mathcal{CI}$*, denoted as* $source(a, c)$*, is class* c *if* $a \in A^+(c)$*. It is a class* $c' \in \mathcal{CI}$*, where* $\{c'\} = mg(\{c^* \mid (c, c^*) \in inh^{a^*}\})$*, otherwise. The source is undefined if* $a \notin A(c)$*.* □

Note that for each class $c, c \in \mathcal{CI}$, $\{c^* \mid (c, c^*) \in inh^{a^*}\}$ is a totally ordered set with respect to the order $<_T$. Therefore, if $mg(\{c^* \mid (c, c^*) \in inh^{a^*}\})$ is not empty, it always contains a single element, that is the maximum of the set. As a consequence, given a class c and an attribute a such that $a \in A(c)$, then $source(a, c)$ is always defined and it is unique.

We now define which is the set of attributes of an object belonging to two most specific classes.

Definition 2.19 *Let* o *be an object such that* $o.CS = \{c_1, c_2\}$*. The set of* o *attributes is* $A(c_1) \uplus A(c_2)$*, where the operation* $\uplus$ *is defined as follows:*

$$\begin{aligned} A(c_1) \uplus A(c_2) = &\{a \mid a \in A(c_1) \cup A(c_2) \wedge a \notin A(c_1) \cap A(c_2)\} \cup \\ &\{a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) = source(a, c_2)\} \cup \\ &\{c_1\text{-}a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) \neq source(a, c_2)\} \cup \\ &\{c_2\text{-}a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) \neq source(a, c_2)\} \end{aligned}$$

□

Example 2.16 *Consider the inheritance hierarchy composed by class* `person` *with two subclasses* `student` *and* `employee`*. Suppose that*

$A(\texttt{person}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}\}$,

$A^+(\texttt{student}) = \{\texttt{code}, \texttt{average\_score}, \texttt{year}\}$,

$A^+(\texttt{employee}) = \{\texttt{code}, \texttt{salary}, \texttt{role}\}$

and therefore

$A(\texttt{student}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}, \texttt{code}, \texttt{average\_score}, \texttt{year}\}$,

$A(\texttt{employee}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}, \texttt{code}, \texttt{salary}, \texttt{role}\}$.

Then,

$A(\texttt{student}) \uplus A(\texttt{employee}) = \{$ `name, address, spouse, year, role,`
`employee-code, student-code, average_score, salary`$\}$. ◇

Operator $\uplus$ is associative. Thus Definition 2.19 can be easily generalized to arbitrary sets of classes.

We now consider the domain (that is, the set of legal values) for attributes. For the attributes not belonging to the intersection of the attribute sets of the most specific classes of the object, and for the ones that have been renamed because of different sources, the domain is simply the one specified in the class to which the attribute belongs. By contrast, for the attributes in the intersection of the attribute sets of two classes and with the same source, a new domain must be determined. Intuitively, the legal values for an attribute must be legal values for both the attributes in the two classes, that is, they must belong to both the domain.

Definition 2.20 *Let $dom(a, c_1) = T_1$ and $dom(a, c_2) = T_2$. Let o be an object such that $o.CS = \{c_1, c_2\}$. Then, value v is a legal value for attribute a in o, if $v \in [\![T_1]\!] \cap [\![T_2]\!]$.* $\square$

Note that the domain of an attribute for an object belonging to several most specific classes cannot be always characterized in terms of an existing type. In particular, if a greatest lower bound of type T_1 and T_2 exists, then the type of the attribute for the object is $T_1 \sqcap T_2$, but a greatest lower bound of two types in the hierarchy does not always exist. Moreover $[\![T_1 \sqcap T_2]\!] \subseteq [\![T_1]\!] \cap [\![T_2]\!]$ holds for any two types T_1 and T_2, while the converse $[\![T_1]\!] \cap [\![T_2]\!] \subseteq [\![T_1 \sqcap T_2]\!]$ holds only if $T_1 \leq_T T_2$ (or vice-versa). In this case, indeed, $T_1 \sqcap T_2 = T_1$ and $[\![T_1]\!] \cap [\![T_2]\!] = [\![T_1]\!]$ in that $[\![T_1]\!] \subseteq [\![T_2]\!]$. If $T_1 \not\leq_T T_2$ and $T_2 \not\leq_T T_1$, then $T_1 \sqcap T_2$ can be defined only if T_1 and T_2 have a common subclass (by multiple inheritance). However if T is a subclass of both T_1 and T_2 ($T_1, T_2 \in ISA(T)$), an object o may exist such that $o \in [\![T_1]\!] \wedge o \in [\![T_2]\!]$ but $o \notin [\![T]\!]$. Indeed, in such a way o can exhibit a context-dependent behavior, while it would not if $o \in [\![T]\!]$.

Example 2.17 *Consider the inheritance hierarchy of Figure 1.1. Suppose that an attribute* $\texttt{spouse} \in A(\texttt{person})$ *and that* $dom(\texttt{spouse}, \texttt{person}) = \texttt{person}$*. Then,*

$\texttt{spouse} \in A(\texttt{student}), A(\texttt{noble}), A(\texttt{male}), A(\texttt{female})$

and suppose that its domain is refined in the following way:

$dom(\texttt{spouse}, \texttt{student}) = \texttt{person}$

$dom(\texttt{spouse}, \texttt{noble}) = \texttt{noble}$

$dom(\texttt{spouse}, \texttt{male}) = \texttt{female}$

$dom(\texttt{spouse}, \texttt{female}) = \texttt{male}$.

In this case, if an object is a student and a noble, then the domain of the attribute $\texttt{spouse}$ *for this object can be characterized in terms of an existing type, that is* $\texttt{noble} = \texttt{noble} \sqcap \texttt{person} = dom(\texttt{spouse}, \texttt{noble}) \sqcap dom(\texttt{spouse}, \texttt{student})$.

By contrast, if an object is a noble female, since

$dom(\texttt{spouse}, \texttt{noble}) \sqcap dom(\texttt{spouse}, \texttt{female}) = \texttt{noble} \sqcap \texttt{male}$

is not defined, the domain of the attribute `spouse` *for this object cannot be characterized in terms of an existing type, but we can only say that the legal values for this attribute are those in* $[\![$`noble`$]\!] \cap [\![$`male`$]\!]$ *(that is, to objects belonging to the two most specific classes* `noble` *and* `male`*).* $\diamond$

Thus, the attributes for which a new domain is determined are the ones which are present in two different classes with a common source. This means that the attribute was defined in a class c (the source) with a domain, say T. Classes c_1 and c_2 have both inherited the attribute, possibly refining its domain into T_1 and T_2, respectively. Due to the refinement condition, both $T_1 \leq_T T$ and $T_2 \leq_T T$ hold. Thus a "compatibility" between types T_1 and T_2 exists, in that they are both subtypes of a common supertype. However, the existence of a common supertype does not ensure neither that $T_1 \sqcap T_2$ is defined nor that a legal value for the attribute exists (that is, $[\![T_1]\!] \cap [\![T_2]\!] \neq \emptyset$). Indeed, referring to the hierarchy of Figure 1.1, though `male` and `female` have a common supertype (`person`), `male` $\sqcap$ `female` is undefined and $[\![$`male`$]\!] \cap [\![$`female`$]\!] = \emptyset$.

The state of an object is then characterized by the set of most specific classes of the object as follows.

Definition 2.21 *(Structural Consistency). Let o be an object, such that $o.v = (a_1 : v_1, \ldots, a_n : v_n)$. Then o is structurally consistent if the following conditions are satisfied:*

- $\biguplus_{c \in o.CS} A(c) = \{a_1, \ldots, a_n\}$;
- $\forall i, 1 \leq i \leq n, v_i \in \bigcap_{c \in o.CS} [\![dom(a_i, c)]\!]$. $\square$

The above definition states that the state of each object must contain a value for each attribute of the classes to which it belongs, and that this value must be of the correct type, that is, it must meet the domain specification.

Definition 2.22 *(Constraint Consistency). An object $o = (i, v, CS)$ is constraint consistent if o falsifies all the bodies of rules implementing the constraints in the constr component of classes in CS.* $\square$

Definition 2.23 *(Consistency). An object $o = (i, v, CS)$ is consistent if o is both a structural and constraint consistent instance of all classes in CS.* $\square$

Example 2.18 *Consider the object of Example 2.14 and class* `person` *of Example 2.9.*

$\nu(i_1) = ($ `name: ´john smith´`, `vatCode: ´666FF´`,
`birthday: (day:8, month:10, year:1969)`,
`income: 5000`, `age: 28` $)$

is a legal value of stype(`person`) $=$

(`name` : `string`, `vatCode` : `string`, `birthday` : `date`, `income` : `integer`, `age` : `integer`),

thus structural consistency holds, since the object only belongs to class `person`.

Furthermore, if the implementation for the constraint `tooLowIncome` *of class* `person` *is specified by the following rule*

$$\texttt{tooLowIncome(N)} \leftarrow \texttt{Self.income} < \texttt{5000}, \texttt{N} = \texttt{Self.name}$$

the atom `Self.income` $<$ `5000` *is falsified by the object (since* `Self.income` $=$ `5000`*), thus constraint consistency holds as well.* $\diamond$

Besides being all consistent, objects in a database must be related in a consistent way. An object (i, v, CS) is said to *refer to* an object (i', v', CS') if i' appears in v. An object (i, v, CS) is said to *be composed of* an object (i', v', CS') if i' appears in v as the value of a composite attribute. A finite set of objects OBJ is consistent if the set is closed under the "refer to" and "composed of" relations, that is, if for each object in the set all objects referred by it belong to the set. This property is also known as *referential integrity* of the set. Intuitively speaking, this notion states that because identifiers are pointers to objects there must be no *dangling pointers* in the set. In addition, for a set of objects to be consistent, the property of oid-uniqueness must be ensured, as well as exclusiveness constraints related to composite attributes. The following definition formalizes these concepts.

Given an object o, $ref(o)$ denotes the set of object identifiers in $\mathcal{OI}$ appearing in $o.v$, whereas $comp(o)$ denotes the set of object identifiers in $\mathcal{OI}$ appearing in $o.v$ as the value for a composite attribute. Given a set of classes $S \subseteq \mathcal{CI}$, $\pi(S)$ denotes the set of objects belonging to a class in the set, that is, $\pi(S) = \bigcup_{c \in S} \pi(c)$.

Definition 2.24 *(Consistent Set of Objects). A (finite) set of consistent objects OBJ is consistent if and only if*

1. OID-UNIQUENESS

   $\forall o_1, o_2 \in OBJ$, *if* $o_1.i = o_2.i$, *then* $o_1.v = o_2.v$ *and* $o_1.CS = o_2.CS$;

2. REFERENTIAL INTEGRITY

(a) *weak (simple) references:*
$\forall o \in OBJ,\ ref(o) \subseteq \{\hat{o}.i \mid \hat{o} \in OBJ\}$;

(b) *composite references:*
$\forall o \in OBJ,\ comp(o) \subseteq \{\hat{o}.i \mid \hat{o} \in OBJ\}$;

3. EXCLUSIVESS

$\forall\ o \in OBJ$, $\forall a_i$ *attribute of* o, *if* a_i *is exclusive for* o *with respect to a set of classes* S, *then*

- $\nexists\ o' \in OBJ,\ o' \neq o,\ o' \in \pi(S)$ *such that* $o.v.a_i \in comp(o')$;
- $\nexists\ a_j,\ a_j$ *attribute of* o, $a_j \neq a_i$, *such that* $o.v.a_i = o.v.a_j$;

4. ESCLUSIVENESS ON THE ATTRIBUTE

$\forall\ o \in OBJ$, $\forall a_i$ *attribute of* o, *if* a_i *is exclusive on the attribute for* o *with respect to a set of classes* S, *then*

- $\nexists\ o' \in OBJ,\ o' \neq o,\ o' \in \pi(S)$ *such that* $o.v.a_i = o.v.a_j$. □

2.5.2 Object Equality

We consider now the notion of object equality. In value-based systems there is no need to distinguish between identical objects and equal objects, since the two notions of equality are the same. By contrast, object-oriented systems need to distinguish them as there is a sharp distinction between values and objects. Classically [92], there are two fundamentally different notions of equality that are supported by object-oriented systems: *equality by identity* (meaning that the two denoted objects are the same object) and *equality by value* (meaning that the two denoted objects have the same attribute values, but not necessary the same identifier). The latter form of equality can be further refined in two ways: *shallow equality* considers the equality of all direct attributes (possibly represented by means of object identifiers) of an object, while *deep equality* also considers the equality of the attributes of objects which are recursively reached by means of oid references. The following definitions formally state these concepts.

Definition 2.25 *(Equality by Identity). Two objects* $o_1 = (i_1, v_1, CS_1)$ *and* $o_2 = (i_2, v_2, CS_2)$ *are equal by identity (denoted by* $o_1 = o_2$*) if and only if* $i_1 = i_2$ *(and obviously* $v_1 = v_2$*, and* $CS_1 = CS_2$ *because of oid-uniqueness).* □

Definition 2.26 *(Shallow Value Equality). Two objects* $o_1 = (i_1, v_1, CS_1)$ *and* $o_2 = (i_2, v_2, CS_2)$ *are equal by value under shallow equality (denoted by* $o_1 == o_2$*) if and only if* $v_1 = v_2$*. This implies both the equality of the attribute values and the equality of the attribute names.* □

Note that shallow value equality does not even require that the two objects be instances of classes related in the inheritance hierarchy, provided that the objects have the same attributes.

Example 2.19 *Consider a database containing a class* `person` *and a class* `company`*, obviously not related in the inheritance hierarchy. Let*

stype(`person`) = *stype*(`company`) = `record-of(name:string, address:string)`.

Consider the objects

- $o_1 = (i_1,$ `(name:`Ford´, address: `Austin´)`, {`person`});
- $o_2 = (i_2,$ `(name:`Ford´, address: `Austin´)`, {`company`}).

o_1 *and* o_2 *are equal by value under shallow equality.* ◇

Before introducing deep value equality, we introduce the notion of span graph of an object. Intuitively the span graph of an object is the graph obtained by recursively replacing each oid-reference in the object value with the state of the object referenced by it. Note that this operation results in a graph, instead of a tree, because of the possibility of circular references among objects.

We introduce an operator $\hat{}: \mathcal{V} \rightarrow \mathcal{V}$, which given a value cuts off the fields of the value which are oids [16]. The operator is inductively defined as follows. Given a value $v \in \mathcal{V}$, $\hat{v}$ is

- $\widehat{null} = null$;
- if $v \in dom(D_i)$, with $D_i \in \mathcal{BVT}$, that is, v is a basic value, then $\hat{v} = v$;
- if $v = \{v_1, \ldots, v_n\}$ then $\hat{v} = \{\hat{v_{i_1}}, \ldots, \hat{v_{i_m}}\}$ where $m \leq n$, and $\{v_{i_1}, \ldots, v_{i_m}\} \subseteq \{v_1, \ldots, v_n\}$ is obtained from $\{v_1, \ldots, v_n\}$ by removing those v_i's for which $\hat{v_i}$ is undefined;

[16]This is due to the fact that when considering deep value equality, oids are not considered in the test, rather they are expanded into their corresponding values.

- if $v = (a_1 : v_1, \ldots, a_n : v_n)$ then $\hat{v} = (a_{i_1} : \hat{v_{i_1}}, \ldots, a_{i_m} : \hat{v_{i_m}})$ where $m \leq n$, and $\{(a_{i_1}, v_{i_1}), \ldots, (a_{i_m}, v_{i_m})\} \subseteq \{(a_1, v_1), \ldots, (a_n, v_n)\}$ is obtained from $\{(a_1, v_1), \ldots, (a_n, v_n)\}$ by removing those pairs (a_i, v_i) for which $\hat{v}_i$ is undefined[17];

- $\hat{v}$ is undefined otherwise.

Note that according to this definition $\hat{\imath}$, $i \in \mathcal{OI}$, is undefined.

Definition 2.27 *(Span Graph)*[18]. *The span graph of an object $o = (i, v, CS)$ (denoted as $span\_graph(o)$), is a directed graph (VE, E) such that*

- *vertices are values in $\mathcal{V}$, not containing identifiers in $\mathcal{OI}$ as components;*

- *edges are labeled by attribute names in $\mathcal{AN}$.*

The graph is defined as follows:

- *$\hat{v}$ is a vertex in VE;*

- *if a vertex $\hat{v}^*$ exists in VE, $\hat{v}^*$ has been obtained from a value v^*, and $i^* \in \mathcal{OI}$ appears in v^* as the value for an attribute a^*[19], then an edge from $\hat{v}^*$ to $\hat{ve}$ with label a^* is added to the graph. $\hat{ve}$ is obtained from the value ve, where ve is $o^*.v$ and $o^*.i = i^*$. If such a vertex is not in VE, then the vertex is added to VE.* □

Example 2.20 *Consider the following objects:*

- $o_1 = (i_1,$ `(name:`Alan Ford´, children:` $\{i_3, i_4\}), \{$`person`$\})$*;*

- $o_2 = (i_2,$ `(name:`computer science´, chief:`$i_1), \{$`department`$\})$*;*

- $o_3 = (i_3,$ `(name:`Susan Ford´, children:` $\{\ \}), \{$`person`$\})$*;*

- $o_4 = (i_4,$ `(name:`Bill Ford´, children:` $\{\ \},$ `department`$:i_2), \{$`student`$\})$.

The span graph of object o_4 is shown in Figure 2.6. ◇

We are now able to give the definition of deep value equality.

[17]The case $v = [v_1, \ldots, v_n]$ is handled similarly, regarding v as $(1 : v_1, \ldots, n : v_n)$, that is, as a record value with natural labels.

[18]This definition assumes that list values are seen as record values with natural labels.

[19]Or as a component of the set which is the value for a^*.

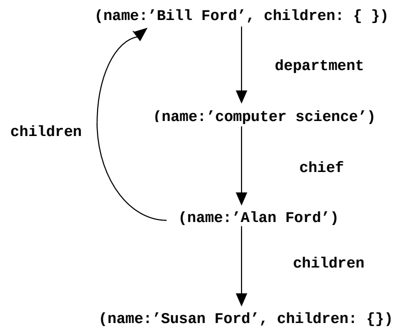

Figure 2.6: Span graph of object o_4 of Example 2.20

Definition 2.28 *(Deep Value Equality). Two objects $o_1 = (i_1, v_1, CS_1)$ and $o_2 = (i_2, v_2, CS_2)$ are equal by value under deep equality (denoted by $o_1 ==_d o_2$) if and only if $span\_graph(o_1) = span\_graph(o_2)$, where the equality of two graphs is the equality of their vertices and the equality of their edges.* □

Clearly equality by identity implies equality by value, and, among equalities by value, shallow equality implies deep equality. Indeed, shallow equality requires that all the objects referred by the two compared objects are exactly the same. Deep equality, by contrast, (recursively) requires the equality of the states of the objects referred by the two compared objects. Thus, because of oid uniqueness, shallow equality implies deep equality.

2.6 Inheritance

Inheritance relationships among classes are described by an ISA hierarchy established by the user. This ISA hierarchy represents which classes are subclasses of (inherit from) other classes. As we have anticipated in Section 2.2.3, this information is expressed as a function $ISA : \mathcal{CI} \rightarrow 2^{\mathcal{CI}}$ such that for each $c \in \mathcal{CI}$, $ISA(c)$ is the set of the (direct) superclasses of c.

A set of conditions must be satisfied by two classes related by the ISA relationship. These conditions are related to the fact that each subclass must contain all attributes, operations, constraints (both on the class as well on the instance level) of all its superclasses. Apart from the inherited concepts, additional features can be introduced in a subclass. Inherited concepts may be redefined (overwritten) in a subclass definition under a number of restrictions. Indeed, the redefinition of the signature of an attribute is possible by specializing, that is, refining, the domain of the attribute. The redefinition of the signature of an operation must verify the *covariance rule* for result parameters and the *contravariance rule* for the input ones. Therefore, result parameter domains may be specialized, whereas input parameter domains may be generalized, in the signature of the operation defined in the subclass. The implementation of an attribute or an operation may be redefined as well, introducing a different implementation of the respective concept, which "overrides" the inherited definition. The redefinition of derived and extensional attributes is not allowed if a derived attribute becomes extensional or vice-versa. We also require that the extent of a subclass is a subset of the extent of all its superclasses.

To formally define notions such as *domain refinement* we rely on the subtype relationship defined in Section 2.2.3.

2.6.1 Signature Refinement

An ordering among signatures is now imposed as follows. Note that we consider covariance for output parameter and contravariance for input ones. Specialization of domains of output parameters of operations can therefore be done by replacing the domain with a proper subtype (*covariance rule*). By contrast, input parameters of operations can be refined by replacing the domain with a proper supertype (*contravariance rule*).

Definition 2.29 *(Signature Refinement). Let* $s = T_1 \times \ldots \times T_j \rightarrow T_{j+1} \times \ldots \times T_n$ *and* $s' = T'_1 \times \ldots \times T'_j \rightarrow T'_{j+1} \times \ldots \times T'_n$ *be two signatures. We say that* s *is a refinement of* s'*, denoted by* $s \leq_S s'$*, if and only if for each* i*,* $1 \leq i \leq j$*,* $T'_i \leq_T T_i$ *and for each* i*,* $j+1 \leq i \leq n$*,* $T_i \leq_T T'_i$*.* □

Example 2.21 *Referring to the ordering of Example 2.7, given* s_1 *and* s_2 *defined as follows*

$s_1 =$ `person` $\times$ `employee` $\times$ `postalcode` $\rightarrow$ `summerdate`
$s_2 =$ `graduate` $\times$ `stud-emp` $\times$ `realpostalcode` $\rightarrow$ `date`

then $s_1 \leq_S s_2$ *holds.* ◇

2.6.2 Subclasses

Based on the ordering on types, we define an ordering on classes to ensure that the user-defined ISA hierarchy meets the compatibility conditions. Firstly, we define an ordering among metaclasses, imposing compatibility conditions in refinement of class features.

Definition 2.30 *Given* $MC_1 = (id_1, struct_1, beh_1, constr_1)$*,* $MC_2 = (id_2, struct_2, beh_2, constr_2)$ *metaclasses,* MC_2 *precedes* MC_1 *(denoted as* $MC_2 \leq_{MC} MC_1$*) if and only if all the following conditions hold:*

- $struct_1 = \{(a'_1, T'_1, at'_1), \ldots, (a'_{k_1}, T'_{k_1}, at'_{k_1})\}$
  $struct_2 = \{(a''_1, T''_1, at''_1), \ldots, (a''_{k_2}, T''_{k_2}, at''_{k_2})\}$
  and for each $i, 1 \leq i \leq k_1$, (a''_j, T''_j, at''_j) *exists,* $1 \leq j \leq k_2$, *such that*
  - $a'_i = a''_j$,
  - $T''_j \leq_T T'_i$,*that is, the domain of the attribute may be refined in a subtype of the domain in the superclass,*
  - $at'_i = at''_j$, *that is, the type of the attribute may not be changed from extensional to derived nor vice-versa;*
- $beh_1 = \{(m'_1, s'_1), \ldots, (m'_{h_1}, s'_{h_1})\}$
  $beh_2 = \{(m''_1, s''_1), \ldots, (m''_{h_2}, s''_{h_2})\}$
  and for each $i, 1 \leq i \leq h_1$ (m''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*
  - $m'_i = m''_j$,
  - $s''_j \leq_S s'_i$, *that is, the signature of the operation is a refinement of the signature in the superclass;*
- $constr_1 = \{(con'_1, s'_1), \ldots, (con'_{h_1}, s'_{h_1})\}$
  $constr_2 = \{(con''_1, s''_1), \ldots, (con''_{h_2}, s''_{h_2})\}$
  and for each $i, 1 \leq i \leq h_1$ (con''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*
  - $con'_i = con''_j$,
  - $s'_i = s''_j$, *that is, we do not consider constraint refinement.* □

We are now able to introduce the ordering among classes. First of all we introduce two different orderings on classes, one considering the time-invariant component of a class (that is, its definitional component) and the other considering the time-varying one (that is, its state and extension). To express compatibility conditions we take into account both orderings.

Definition 2.31 *(Intensional Ordering). Given classes* C_1 *and* C_2 *such that* $C_1.sign = (struct_1, beh_1, constr_1, mc_1)$, *and* $C_2.sign = (struct_2, beh_2, constr_2, mc_2)$, C_2 *precedes* C_1 *in the intensional ordering (denoted as* $C_2 \leq^i_C C_1$*) if and only if all the following conditions hold:*

- $struct_1 = \{(a'_1, T'_1, at'_1, e'_1, dep'_1, del'_1), \ldots, (a'_{k_1}, T'_{k_1}, at'_{k_1}, e'_{k_1}, dep'_{k_1}, del'_{k_1})\}$
  $struct_2 = \{(a''_1, T''_1, at''_1, e''_1, dep''_1, del''_1), \ldots, (a''_{k_2}, T''_{k_2}, at''_{k_2}, e''_{k_2}, dep''_{k_2}, del''_{k_2})\}$
  and for each i, $1 \leq i \leq k_1$, $(a''_j, T''_j, at''_j, e''_j, dep''_j, del''_j)$ *exists,* $1 \leq j \leq k_2$, *such that*
  - $a'_i = a''_j$,

- $T''_j \leq_T T'_i$, *that is, the domain of the attribute may be refined in a subtype of the domain in the superclass,*
- $at'_i = at''_j$, $e'_i = e''_j$, $dep'_i = dep''_j$, $del'_i = del''_j$, *that is, the type of the attribute, and the constraints associated with it may not be changed;*

- $beh_1 = \{(m'_1, s'_1), \ldots, (m'_{h_1}, s'_{h_1})\}$
  $beh_2 = \{(m''_1, s''_1), \ldots, (m''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (m''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*
  - $m'_i = m''_j$,
  - $s''_j \leq_S s'_i$, *that is, the signature of the operation is a refinement of the signature in the superclass;*

- $constr_1 = \{(con'_1, s'_1), \ldots, (con'_{h_1}, s'_{h_1})\}$
  $constr_2 = \{(con''_1, s''_1), \ldots, (con''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (con''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*
  - $con'_i = con''_j$,
  - $s'_i = s''_j$,*that is, we do not consider constraint refinement*

- $mc_2 \leq_{MC} mc_1$, *that is, we impose the same rules for the refinement of class features, as for refinement of instance features.* □

Example 2.22 *Consider class* `person` *defined as follows*[20].

id = `person`
pop = *ext*
struct = { (`name, string`, *ext*), (`birthday, date`, *ext*),
(`spouse, person`, *ext*, (*true*, {`person, employee`}), *false*, *Restr*),
(`income, integer`, *ext*), (`age, integer`, *der*) }
beh = { (`changeIncome : integer` → `person`) }
constr = { `tooLowIncome : string`) }
mc = `m-person`

and the following class `employee`

id = `employee`
pop = *der*

[20]In the example, we omit the *a_excl*, *a_dep* and *a_del* for attributes whose domains are basic value types, since those fields are not meaningful.

$struct = \{$ ($\texttt{name, string}$, *ext*), ($\texttt{birthday, date}$, *ext*),
($\texttt{spouse, employee}$, *ext*, (*true*, $\{\texttt{person, employee}\}$), *false*, *Restr*)),
($\texttt{income, integer}$, *ext*), ($\texttt{age, integer}$, *der*), ($\texttt{empnr, integer}$, *ext*),
($\texttt{manager, employee}$, *ext*, (*false*, $\emptyset$), *false*, *Restr*),
($\texttt{dependents, set-of(employee)}$, *ext*, (*false*, $\emptyset$), *false*, *Restr*) $\}$
$beh = \{$ ($\texttt{changeIncome} : \texttt{integer} \rightarrow \texttt{employee}$) $\}$
$constr = \{$ $\texttt{tooLowIncome} : \texttt{string}$
$\texttt{exceedMgrSalary} : \texttt{integer}$ $\}$
$mc = \texttt{m-employee}$

Then, provided that $\texttt{m-employee} \leq_{MC} \texttt{m-person}$, $\texttt{employee} \leq_C^i \texttt{person}$. ◇

Definition 2.32 *(Extensional Ordering). Given* $C_1 = (id_1, sign_1, impl_1, state_1)$*, and* $C_2 = (id_2, sign_2, impl_2, state_2)$ *classes,* C_2 *precedes* C_1 *in the extensional ordering (denoted as* $C_2 \leq_C^e C_1$*) if and only if* $state_2.extent \subseteq state_1.extent$*, that is, the set objects members of the subclass is included in the set of the members of the superclass.* □

Note that, as we will see in Section 2.7, in each consistent database the ISA hierarchy must be consistent with the orderings $\leq_C^i$ and $\leq_C^e$. We thus require that the ISA specification (the subclass relationship) does not contradict the defined orderings. This consistency must be ensured at two different levels: the intensional and the extensional ones. At the intensional level (schema level) the ordering on classes imposed by the ISA hierarchy is required to be consistent with the intensional ordering on classes. Therefore, each subclass must contain all the features of the superclasses, possibly refined. At the extensional level (instance level), the ordering on classes imposed by the ISA hierarchy is required to be consistent with the extensional ordering on classes. Intuitively an ISA relationship is well-defined if it is consistent with the compatibility conditions stated above.

In what follows, for simplicity of notation, we apply function ISA^* to classes, instead of to class identifiers. Let $\mathcal{C}$ denote the set of all classes. Then, $ISA^* : \mathcal{C} \rightarrow \mathcal{C}$ is simply defined as

$ISA^*(C) = \{C' \mid C'.id \in ISA^*(C.id)\}$.

This mapping is very intuitive due to the one-to-one correspondence between classes and class names.

Definition 2.33 *(Well-defined Inheritance Relationship). An inheritance relationship ISA is said to be intensionally-well-defined if* $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq_C^i C_2$*, that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *in the intensional ordering.*

An inheritance relationship ISA is said to be extensionally-well-defined if $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq^e_C C_2$*, that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *in the extensional ordering.*

An inheritance relationship ISA is said to be well-defined if it is both intensionally-well-defined and extensionally-well-defined, namely, if $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq^i_C C_2 \wedge C_1 \leq^e_C C_2$ *that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *both in the extensional and in the extensional ordering.* □

By contrast, two classes $C_1, C_2 \in \mathcal{C}$ may exist such that $C_1 \leq^i_C C_2$ and $C_1 \leq^e_C C_2$, but $C_1 \notin ISA^*(C_2)$. Consider the case of a class `person` with two subclasses `male` and `student` which simply inherit structure and behavior (both at the instance and at the class level) without adding nor refining anything. The two classes have therefore identical structure and behavior and thus `student` $\leq^i_C$ `male` (and also the vice-versa). Suppose that all students are male, then we have `student`.*extent* $\subseteq$ `male`.*extent* and therefore `student` $\leq^e_C$ `male`. In this case `student` $\leq^i_C$ `male` and `student` $\leq^e_C$ `male`, while `male` $\notin ISA^*($`student`$)$.

2.6.3 Properties of the Inheritance Hierarchy

As we have anticipated in Section 2.2.3 the relation $\leq_T$ defines a partial order. Therefore, the type system $(\mathcal{T}, \leq_T)$ can be represented as a directed acyclic graph (DAG). In this DAG there is a node for each type in $\mathcal{T}$, and there is a path from node T_2 to node T_1 if and only if $T_2 \leq_T T_1$. The existence of a common root of the entire type hierarchy is not imposed, that is, it is not imposed the existence of a common supertype of all types in $\mathcal{T}$. Rather, the hierarchy is partitioned into multiple strongly connected components. Each strongly connected component of the DAG is characterized by a single node without outgoing edges, corresponding to a class without superclasses, which we call *root* of the strongly connected component.

Definition 2.34 *(Root Class). A class* C *is a root if* $ISA(C) = \emptyset$*.* □

Example 2.23 *Referring to the inheritance hierarchy of Example 2.6 the only root class is* `person`*.* ◇

In our model multiple inheritance is supported. However the constraint is imposed that for multiple inheritance a common ancestor must exist. Therefore a class C can be defined as a subclass of classes C_1 and C_2 only if $ISA^*(C_1) \cap ISA^*(C_2) \neq \emptyset$.

Note that an object can belong to several most specific classes only if the classes belong to the same strongly connected component in the DAG, that is, if they have a common ancestor. The sets of oids in different strongly connected components are therefore disjoint. That is, the extents of root classes, which are the extents of the entire connected components, are disjoint. Note that we may think at $\mathcal{C}$ as partitioned in $\mathcal{H}_1, \ldots, \mathcal{H}_m$,that is, into m distinct hierarchies.

2.7 Schema and Database

We are now able to integrate the notions of values, types, objects, classes and inheritance we have dealt with in the previous sections. Our model, as most object-oriented data models, distinguishes between the schema level (the time-invariant component) and the instance level (the time-varying one).

Definition 2.35 *(Schema). Given a set $\mathcal{AN}$ of attribute names, a set $\mathcal{TN}$ of type names, a set $\mathcal{MN}$ of method names, a set $\mathcal{CI}$ of class identifiers and a collection of basic domains $D_1, \ldots, D_n$, a schema is a tuple*

$$(VT, Cl, MCl, type, ISA)$$

where

- $VT \subseteq \mathcal{VT}$[21] *is a (finite) set of value types;*
- Cl *is a finite set of definitional components of classes;*
- $MCl \subseteq \mathcal{MC}$ *is a finite set of metaclasses;*
- $type : \mathcal{TN} \rightarrow \mathcal{VT}$ *is a total function on* $\mathcal{TN}$*;*
- $ISA : Cl \rightarrow 2^{Cl}$ *is a total function on* Cl *for which the following holds:*
  - *a)* ISA *is a DAG;*
  - *b)* ISA *is int-well-defined.*

All class and metaclass names are distinct and for each class the corresponding metaclass must exist. □

Definition 2.36 *(Database). Let S be a schema. A database over S is a tuple*

[21] $\mathcal{VT}$ is a denumerable infinite set of value types, implicitly defined starting from primitive domains, class names and attribute names according to Definition 2.3.

$(OBJ, \pi, cval)$

where

- OBJ *is a consistent set of objects;*
- π *is a pair of functions*

  $\pi_V : \mathcal{ET} \rightarrow 2^{\mathcal{V}}$, *value assignment*[22],
  $\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$, *oid assignment*

  which handle extents of extended types and classes;
- $cval$ *is a total function* $cval : Cl \rightarrow \mathcal{V}$ *such that* $\forall C \in Cl \;\; cval(C)$ *is a legal value for* $stype(C.mc)$, *that is, the function* $cval$ *assigns values to class attributes of* C*;*

such that:

(i) $\forall ET \in VT$, *extended value type,*
$\pi^*(ET) \subseteq [\![struct(ET)]\!]_\pi$;

(ii) $\forall o = (i, v, CS) \in OBJ$, $\forall c \in CS$, c *belongs to* $Cl.id$*;*

(iii) $\forall o = (i, v, CS) \in OBJ$ o *is a consistent instance of all classes in* CS*;*

(iv) $\forall C \in Cl$, $c = C.id$,
$cval(C).extent = \pi^*(c)$ *and*
$cval(C).proper\text{-}extent = \pi(c)$;

(v) $\forall o = (i, v, CS) \in OBJ$, $\forall c \in CS$, $i \in \pi(c)$;

(vi) $\forall C \in Cl$, $\forall i \in cval(C).proper\text{-}extent$, $o = (i, v, CS) \in OBJ$ *and* $C.id \in CS$*;*

(vii) ISA *is ext-well-defined;*

(viii) if $C_1, \ldots, C_n \in Cl$ *are the roots of the inheritance hierarchy, then*
$cval(C_i).extent \cap cval(C_j).extent = \emptyset$, $\forall i, j$, $i \neq j$, $1 \leq i, j \leq m$. $\square$

[22] $\mathcal{V}$ is the set of values defined starting from basic values and object identifiers, according to Definition 2.8.

2.8 Concluding Remarks

In this chapter we have presented a formal definition of our reference object-oriented data model. Table 2.1 provides a comparison among formal models for object-oriented databases so far proposed. In particular, we have compared the formal models under the following aspects: whether complex objects are supported; whether the model supports single or multiple inheritance; whether the model supports both the notions of type and class[23]; whether relations are included in the model (thus coupling an object model with a value-oriented one); whether typing issues are addressed. Most models originate from the work of Beeri [25], which has been the first to clarify several issues on object-oriented data models. Furthermore, some proposals formalize a core object-oriented data model as a basis for addressing other issues, such as particular forms of integrity constraints in [152], a behavior specification in [106], an algebra and communication issues in [126]. Also the model developed in [169] may be seen as a basis for the development of an algebra [168]. Thus, those formal models do not adequately cover all features of a real object-oriented data model.

Other formal models for object-oriented databases have been developed that we have not considered in the comparison. However, most of them are concerned with a specific aspect and only consider a partial view. In [13] type hierarchies and subtyping problems are considered, but other aspects are not addressed. In [165] a simple object model is formally specified as a basis for addressing query issues. [93] and [144] are mainly concerned with the specification of object behavior.

[23]We remark that in models supporting both the notion of type and that of class, the two notions seldom have the same meaning in different models.

	[4, 124]	[152]	[106]	[169]	[126]	Our model
Complex Objects	Partially	YES	Partially	YES	YES	YES
Inheritance	Single	Multiple	No	Single	Multiple	Multiple
Classes & Types	Types only	Both	Sorts only	Both	Both	Both
Relations	NO	NO	YES	NO	NO	NO
Schema - Instance Levels	YES	YES	YES	YES	YES	YES
Feature Refinement	YES	NO	NO	Attributes only	YES	YES
Type System	NO	NO	NO	NO	NO	YES
Class Features	NO	NO	NO	NO	NO	YES
Named Types	NO	NO	NO	NO	NO	YES

Legenda:

Partially denotes the non-complete orthogonality in applying constructors to arbitrary values.

Table 2.1: Comparison with other formal models for OO databases

Chapter 3

Method Definition Language

In this chapter we introduce a set-oriented rule-based method definition language for object-oriented databases. Most existing object-oriented database systems exploit a general purpose imperative object-oriented programming language as method definition language. Because methods are written in a general-purpose imperative language, it is difficult to analyze their properties and to optimize them. Optimization is important when dealing with large amount of objects as in databases. We therefore believe that the use of an ad-hoc, set-oriented language can offer some advantages, at least at a specification level. In particular, such a language can offer an appropriate framework to reason about method properties.

In the chapter, besides defining the syntax of the method definition language, we formally define its semantics, addressing the problems of inconsistency and non-determinism of set-oriented updates. Moreover, we address typing issues and we characterize some relevant properties of methods, such as conflicts among method specifications in sibling classes and behavioral refinement in subclasses, in the framework of the proposed language.

The chapter is organized as follows. In Section 3.1 we introduce the syntax of the method definition language. Section 3.2 is devoted to typing aspects. The semantics of the language is presented in Section 3.3, while method properties are discussed in Section 3.4. Finally, Section 3.5 concludes the chapter.

3.1 Syntax

In this section we specify the syntax of the method definition language, by first introducing declarative expressions, then update primitives, and finally methods.

3.1.1 Declarative Expressions

In what follows we introduce deductive rules, which are a mean to specify declarative expressions on a database. They are used to express constraints and to specify the implementation of different class features. We consider a set of variables Var_T for each type $T \in \mathcal{T}$ of the language, to which the special variables `Self` and `Class` belong. These special variables denote, respectively, the object/the class on which the rule is being executed.

Definition 3.1 *(Terms). The set of terms $Term$ is inductively defined as follows:*

- *each variable $x \in Var_T$ is a (atomic) term;*
- *each basic value (constant) in $dom(D_1), \ldots, dom(D_n)$, and the null value are (atomic) terms;*
- *let $t_1, \ldots, t_n$, $n \geq 0$ be terms, then $\{t_1, \ldots, t_n\}$ is a (complex) term;*
- *let $t_1, \ldots, t_n$, $n \geq 0$ be terms, then $[t_1, \ldots, t_n]$ is a (complex) term;*
- *let $t_1, \ldots, t_n$, $n \geq 0$ be terms, and $a_1, \ldots, a_n \in \mathcal{AN}$ be distinct labels, then $(a_1 : t_1, \ldots, a_n : t_n)$ is a (complex) term;*
- *let t be a term, $a \in \mathcal{AN}$ a label, then $t.a$ is a (complex) term;*
- *let t be a term, n a natural number, then $t.n$ is a (complex) term;*
- *let $c \in \mathcal{CI}$ be a class name, $a \in \mathcal{AN}$ a label, then $c.a$ is a (complex) term.* $\square$

In addition to terms introduced by the definition above, a set of standard predefined operators that can be used to build terms is also provided. These predefined operators include arithmetic operators, set operators and list operators.

Example 3.1 *The following are examples of terms:*

- X;
- { `‘bob’`, `‘john’`, `‘sue’` };
- $[true, X]$;
- $(a : X, b : true, c : 1627)$;
- $X.name$;

- $[true, X].1$;
- `person.averageAge`

where $X \in Var$, `person` $\in \mathcal{CI}$, $a, b, c, name, averageAge \in \mathcal{AN}$. $\diamond$

According to the above definition, values (except oids) are terms, path expressions (built by making use of the dot notation) are terms. In addition we consider a number of terms obtained using classical predefined operators for integers, reals, lists, sets. Of course there are type constraints for the applicability of such operators; we deal with typing of terms, making explicit the type of terms to which operators may be applied, in Section 3.2. Obviously, not all the terms obtained applying the operators on any two terms are correct terms.

We remark that oids, though values, are not included in terms. This is due to the fact that we do not allow oids to be manipulated explicitly by the user. In a user expression (such as a formula) the user may bind a variable to an oid, and then "access" the object denoted by this variable. We have indeed included in terms a typed set of variables, containing also variables of object types, that is variables denoting objects.

Like terms, formulas are either atomic or complex. Atomic formulas are composed of a predicate symbol and a list of parameter terms. Predicate symbols include a number of special predefined relations including comparison predicates, such as $<$ or $==$, and membership predicates (in). Class and type names, seen as unary predicate symbols, are predicate symbol, too. Finally, predicate symbols contain the set $\mathcal{PN}$ of constraint names.

Definition 3.2 *(Atomic Formulas). Atomic formulas are defined as follows:*

- COMPARISON FORMULAS

  if t_1, t_2 *are terms and* $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$ *is a predefined predicate, then* $t_1 \ op \ t_2$ *is a comparison atomic formula;*

- MEMBERSHIP FORMULAS

  if t_1, t_2 *are terms, then* $t_1 \ in \ t_2$ *is a membership atomic formula;*

  if t *is a term and* $c \in \mathcal{CI}$ *is a class name, or* $c \in \mathcal{ET}$ *is an extended value type name, then* $t \ in \ c$ *is a membership atomic formula;*

- CLASS AND TYPE FORMULAS

  if t *is a term and* c *is a class (or type) name, then* $c(t)$ *is a class formula*[1]*;*

[1] We will see in detail later how these formulas are used.

- FUNCTION CALLS

  if m is the name of a side-effects free method, O is an object-denoting variable and $t_1, \ldots, t_n$ are terms, then $O.m(t_1, \ldots, t_n)$ is an atomic formula;

- CONSTRAINT FORMULAS

  if $t_1, \ldots, t_n$ are terms and con is a constraint symbol in $\mathcal{PN}$, then $con(t_1 \ldots, t_n)$ is a constraint formula. □

Side-effects free methods can be invoked in declarative expressions for two reasons:

1. computing values that can be used inside the condition itself; for example: the method invocation `X.year_salary (Y)` which invoked on an employee `X` multiplies his salary by twelve and binds `Y` to the amount `X` gets during one year;

2. computing boolean predicates; for example: the method invocation `X.is_mgr` which invoked on an employee `X` evaluates to *true* if such employee is a manager.

Example 3.2 *The following are examples of formulas.*

- *comparison formulas*

  $X = Y$
  $X == Z$[2]
  `Self.age` $>$ `18`
  `X.name = ´john´`

- *membership formulas*

  `X in Self.dependents`
  $\{$`1,2`$\}$ `in X.dates`
  $Y.manager$ *in* `employee`

- *class formulas*

  `person(X)`
  `date((day:8, month:10, year:1969))`

- *constraint formulas*

  `tooLowIncome(X)`
  `invalidLifeExpentancy(Y).` ◇

[2]In this formula we test for shallow value equality, while in the previous one we test for identity.

Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. All variables are assumed to be implicitly quantified as in Datalog.

Definition 3.3 *(Formulas). Formulas are inductively defined as follows:*

- *all atomic formulas are formulas;*
- *if F is an atomic comparison, membership or constraint formula*[3]*, then $\neg F$ is a (complex) formula;*
- *if F_1 and F_2 are formulas, then F_1, F_2 is a (complex) formula, where the symbol "," denotes the $\wedge$ logical connective.* □

Example 3.3 *Examples of formulas are the following:*

- $\texttt{person(X)}, \texttt{X.profession} =' \texttt{engineer}'$;
- $\texttt{person(X)}, \neg \texttt{X in student}, \texttt{X.income} < 6000$. ◇

We require that each formula contains exactly one class formula for each variable. This requirement is motivated in Section 3.2.1. In addition, we require formulas to be range restricted [98], to avoid formulas that are satisfied by an infinite set of instances.

The formulas we consider are thus similar to first order formulas extended with some notions typical of the object-oriented paradigm and, in particular, with function calls.

Deductive rules, as specified by the following definition, are means for defining constraints, derived attributes and views (see Chapter 5).

Definition 3.4 *(Deductive Rule). A deductive rule is an expression of the form*

$$Head \leftarrow Body$$

where

- *Head is an atomic formula;*
- *Body is an arbitrary formula;*

[3]Class formulas cannot be negated.

- *each variable in Head occurs in Body.* □

Example 3.4 *Examples of deductive rules are the following:*

- `Self.salary = 20000` $\leftarrow$ `engineer(Self), Self.age` $<$ `35`
  defining the implementation of a derived attribute;
- `Y in Self.children` $\leftarrow$ `person(Self), person(Y), Y.father=Self`
  defining the implementation of a derived set attribute;
- `improperDate(Date)` $\leftarrow$ `Date.day` $<$ `0`
  defining a constraint;
- `tooLowIncome(N)` $\leftarrow$ `Self.income`$<$`50000, N = Self.name`
  defining a constraint. ◇

Deductive rules must in addition satisfy certain stratification and safety conditions [98].

3.1.2 Update Primitives

Our language provides a number of predefined operations for expressing fundamentals data manipulations on the object database. Supported update primitives are: object creation and deletion; object migration from a class to another, modifications to the value of an object attribute. In addition, method invocations are (non-atomic) update operations, too.

The following definition defines update operations of our language.

Definition 3.5 *(Update). Let $c, c' \in \mathcal{CI}$ be class names, $a \in \mathcal{AN}$ be an attribute name, O be an object-denoting variable, m be a method name, and $t, t_1, \ldots, t_n$ be terms. An update has one of the following forms:*

- `create`(c, t, O)*;*
- `delete`(c, O)*;*
- `specialize`(c, c', O, t)*;*
- `generalize`(c, c', O)*;*
- `modify`$(c.a, O, t)$*;*

- $O.m(t_1, \ldots, t_n)$. □

A `create`(c, t, O) operation creates a new object of class c, returning in O the oid of the newly created object. t is a record term, containing the values for initializing the object attrivutes. Attributes of class c for which no value is specified in t are initialized to the *null* value. A `delete`(c, O) operation deletes the object denoted by the oid to which O is bound from the class c to which it belongs (and also from any other class of the database). As we will discuss in specifying the semantics of the deletion operation, this could cause the deletion of other objects, because of referential integrity and dependency constraints.

A `specialize`(c, c', O, t) operation migrates the object denoted by O from class c to class c', which must be a subclass of c. t is a record term, containing the values for initializing additional attributes of class c'. The new state of the object denoted by O is obtained by appending t to its old state. Obviously, the object denoted by O continues to belong to c as a member. A `generalize`(c, c', O) operation migrates the object denoted by O from class c to class c', which must be a superclass of c. The object is thus deleted from class c, and it becomes an instance of class c', of which it was already member. All the values for proper attributes of class c are removed from the object state. Note that object migration operations do not affect object identity.

A `modify`$(c.a, O, t)$ operation, set the value of attribute a, which must be an attribute of class c, of the object denoted by O, to the value to which t evaluates. The object denoted by O must belong to class C. Finally, a $O.m(t_1, \ldots, t_n)$ operation invokes the operation m on the object denoted by O, with parameters $t_1, \ldots, t_n$.

Example 3.5 *Given class* `person` *and its subclass* `employee` *as defined in Example 2.22, given a variable* `X`*, the following are examples of actions:*

- `create(person, (name:`john smith´, birthday:(day:8,month:10,year:1969), spouse:null, income:5000), X)`*;*
- `delete(person,X)`*;*
- `specialize(person,employee,X,(empnbr:14038,manager:null,dependents:{}))`*;*
- `generalize(employee, person, X)`*;*
- `modify(person.income, X, 6000)`*;*
- `X.changeIncome(1000)`*.* ◇

3.1.3 Methods

The update primitives we have introduced in the previous section are the constructs which can appear in the action part of a method. An action list is a sequence of update primitives.

Definition 3.6 *(Action List). Let u_i, $i = 1, \ldots, n$, be updates defined according to Definition 3.5, then $u_1; \ldots; u_n$ is an action list.* □

An update rule is a rule of the form *condition* $\rightarrow$ *action*, where *condition* is a formula expressing a declarative control upon *action* execution. That is, the action is executed only if the condition is verified, and the evaluation of the condition determines the set of bindings on which the action is executed.

Definition 3.7 *(Update Rule). An update rule has the form condition $\rightarrow u_1; \ldots; u_n$ where condition is a formula (according to Definition 3.3) and $u_1; \ldots; u_n$ is an action list (according to Definition 3.6).* □

A method implementation is specified by a set of update rules. Rules in a method implementation are executed sequentially, according to the order they appear in the implementation.

Definition 3.8 *(Method Implementation). A method implementation has the form:*

$$m(P_1, \ldots, P_n) : condition_1 \rightarrow action_1 || \ldots || condition_m \rightarrow action_m$$

where:

m is a method name;

$P_1, \ldots, P_n$ are the formal method parameters, that is, variables;

$condition_i \rightarrow action_i$, $i = 1, \ldots, m$, is an update rule (according to Definition 3.7). □

Example 3.6 *The following is an example of method implementation.*

```
increase_salary(Amount): Self.profession="clerk" →
                 modify(employee.salary, Self, (Self.salary+Amount))||
                 Self.profession="manager" →
                 modify(employee.salary, Self, (Self.salary+(2*Amount)))
```

Method `increase_salary` *increases the salary of the employee of an amount which depends on the employee job.* `Self` *denotes the object on which the method is invoked.* ◇

3.2 Typing Issues

In this section we deal with the typing of our language constructs. We first present rules for typing terms and formulas, making explicit the type requirements we impose on such constructs. Subtyping is based on the well-known ideas of Cardelli concerning object-oriented type systems [22, 53, 54, 55]. This kind of typing is performed assuming that for each variable a type is known, thus we explicitly investigate this issue. We then consider typing rules for update primitives and discuss type correctness of method implementations. Finally, we examine the class implementation and establish some criteria for the consistency of an implementation with respect to a signature.

3.2.1 Typing of Terms and Formulas

Figure 3.1 presents the term typing rules. Typing rules for terms built with predefined operators are not presented here due to space limitations. They can be found in [98]. Figure 3.2 specifies some conditions that must be satisfied for the correct application of predicates to terms. Therefore, these conditions exactly specify legal formulas. As far as class formulas are concerned we do not specify any type constraint for the application of class/type predicates because class formulas are the mean to specify the type of a variable. We only require that class formulas be applied to variables.

The above typing rules are based on the existence of a basis, that is, of a set of statements of the form $X \in Var_T$, where X is a variable and T is a type. Other typing rules use this basis. In the following we discuss how these bases are obtained.

When writing a formula, we must explicitly state the type of each variable used in the expression. This is accomplished in our language by using class formulas. Indeed, we use class formulas as a typing mechanism. A class formula has the form $c(X)$ where $c \in \mathcal{CI}$ is a class name and X is a variable. The meaning of this formula is to assign the type corresponding to c to variable X and to state that the objects, to which X may be instantiated, must be members of the class identified by c. This formula may be seen as a shorthand for the expression $X : c \wedge X$ `in` $\pi^*(c)$ where $X : c$ is a type declaration, and `in` is the membership predicate. Class formulas are built from class or type names representing unary predicate symbols. Thus, to state that a variable X is of type T (that is $X \in Var_T$) we add to our expression the class formula $T(X)$. Note that this has also the effect to state that $X \in [\![T]\!]$, thus providing a domain for the evaluation of the expression.

For each type declaration $c(X)$ we add $X \in Var_c$ to the basis, and, to properly type the pseudovariable `Self`, if a rule appears in the context of a class c we add `Self` $\in Var_c$ to the basis.

$$\frac{}{null : T} \quad T \in \mathcal{T}$$

$$\frac{X \in Var_T}{X : T}$$

$$\frac{t \in dom(D_i)}{t : D_i} \quad D_i \in \mathcal{BVT}$$

$$\frac{t_i : T_i \quad (1 \leq i \leq n)}{\{t_1, \ldots, t_n\} : set\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{[t_1, \ldots, t_n] : list\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{t_i : T_i \quad (1 \leq i \leq n)}{(a_1 : t_1, \ldots, a_n : t_n) : record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)} \quad a_1, \ldots, a_n \in \mathcal{AN}$$

$$\frac{t : record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{t : list\text{-}of(T)}{t.n : T} \quad n \in \mathbb{N}$$

$$\frac{t : Tn, Tn \in \mathcal{TN}, type(Tn) = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{t : Tn, Tn \in \mathcal{TN}, type(Tn) = list\text{-}of(T)}{t.n : T} \quad n \in \mathbb{N}$$

$$\frac{t : c, \quad c \in \mathcal{CI}, \quad stype(c) = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{c \in \mathcal{CI},\ C \in \mathcal{C},\ C.id = c,\ C.mc = c',\ stype(c') = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)}{c.a_i : T_i} \quad i = 1 \ldots n$$

Figure 3.1: Term typing rules

$$\frac{t_1, t_2 : T}{t_1\ op\ t_2}\quad T \in \{\texttt{integer}, \texttt{real}, \texttt{character}, \texttt{string}\},\ op \in \{<, >, \geq, \leq\}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 = t_2}\quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 == t_2}\quad T_1, T_2 \in \mathcal{OT},\ T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 ==_d t_2}\quad T_1, T_2 \in \mathcal{OT},\ T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1, t_2 : \mathit{set\text{-}of}(T_2)}{t_1\ \mathit{in}\ t_2}\quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1, t_2 : \mathit{list\text{-}of}(T_2)}{t_1\ \mathit{in}\ t_2}\quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : \mathit{set\text{-}of}(T_1), t_2 : \mathit{set\text{-}of}(T_2)}{t_1\ \mathit{in}\ t_2}\quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : \mathit{list\text{-}of}(T_1), t_2 : \mathit{list\text{-}of}(T_2)}{t_1\ \mathit{in}\ t_2}\quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t : T,\quad c \in \mathcal{CI}}{t\ \mathit{in}\ c}\quad T \in \mathcal{OT}, T \sqcup c \text{ is defined}$$

$$\frac{t : T,\quad ET \in \mathcal{ET}}{t\ \mathit{in}\ ET}\quad T \in \mathcal{VT}, T \sqcup ET \text{ is defined}$$

$$\frac{X \in Var,\quad T \in \mathcal{T}}{T(X)}\quad Var = \bigcup_T Var_T$$

Figure 3.2: Formula typing rules

The scope of a type assignment for a variable is that of an expression (a formula). Each formula must contain a class formula for each variable. Thus, each variable is associated with a unique type, with respect to which type checking is performed. Declaring a variable X of type T influences type checking in that the only features available for X are those of type T. That is, type checking of the expression, in which X appears, is done regarding X as a term of type T. We say that T is the *static type* of variable X, because it is the type with respect to which the static type checking of the expression containing X is done.

At execution time variable X is instantiated with a value of type T. Note that when T is a class name, the variable is instantiated with the identifier of an object member of the class identified by T. This means that the variable may also be instantiated with the identifier of an object instance of a class T', subclass of T. T' is said the *dynamic type* of variable X. However, since we consider objects possibly belonging to several most specific classes, the dynamic type of a variable denoting an object is, in general, not unique. Thus, a variable can be characterized by a set of dynamic types. The relationships between the static and dynamic types of variables will be discussed in Chapter 6.

Another way to type variables in declarative expressions is offered by the possibility of invoking, in declarative expressions, side-effect free methods. A side-effect free method is characterized by some input parameters (for which a type must be declared in the formula in which the method invocation appears) and by some output parameters (for which the type can be determined by the method invocation itself). Thus, there are two different ways for assigning a type to a variable in a formula:

1. explicitly, through a class or type formula;
2. implicitly, if the variable is an output parameter of a (side-effect free) method, invoked in the formula.

Thus, the basis from which we start to evaluating types in conditions consists of:

1. $X \in Var_c$ and $X \in Var_T$ for each class and type formula,
2. $P_i \in Var_{T_i}$ for each P_i method parameter, being T_i the type of the i-th parameter in the method signature.

Figure 3.3 illustrates rules for side-effect free method invocations. A generic method invocation has the form $O.m(t_1, \ldots, t_n)$ where t_i, for $i = 1, \ldots, j$, are input parameters, whereas t_i, for $i = j + 1, \ldots, n$, are output parameters. While input parameters can be arbitrary terms, output parameters must be variables. In the rules, we have assuned that, if no type has been specified for an output parameter, the corresponding variable has an undefined type, which we have denoted as $\perp$. Thus, $T \leq_T \perp$ for any $T \in \mathcal{T}$.

$$\frac{O : c,\quad c \in \mathcal{CI},\quad t_i : T'_i\ (1 \leq i \leq j),\quad t_i : \bar{T}_i\ (j+1 \leq i \leq n)}{O.m(t_1, \ldots, t_n)}\ Cond$$

$Cond$:

$$\begin{array}{l} sign(m, c) = T_1 \times \ldots \times T_j \rightarrow T_{j+1} \times \ldots \times T_n, \\ T'_i \leq_T T_i,\ T'_i \in \mathcal{T}, \quad (1 \leq i \leq j) \\ T_i \leq_T \bar{T}_i,\ \bar{T}_i \in \mathcal{T} \cup \{\perp\}, \quad (j+1 \leq i \leq n) \end{array}$$

$$\frac{O.m(t_1, \ldots, t_n)\quad t_i : \perp \quad (j+1 \leq i \leq n)}{t_i : T_i}\ Cond'$$

$Cond'$:

$$sign(m, c) = T_1 \times \ldots \times T_j \rightarrow T_{j+1} \times \ldots \times T_n,$$

Figure 3.3: Typing rules for method invocations

The typing rules points out that, while types of the method input parameters must be known before the invocation, and they must be compatible with those specified in the method signature, types of the method output parameters, by contrast, may or may not be already known. In the former case they must be compatible with those specified in the method signature, whereas in the latter the types of output parameters are deduced from the types specified for them in the method signature.

3.2.2 Typing of Atomic Updates

Figure 3.4 illustrates the typing rules for atomic updates. Note that the only atomic update which can be executed on a non-typed variable is the `create` operation, which, if executed on a non-typed variable, sets its type to the class on which the creation is executed.

In the rule for the `specialize` operation, the notation $T_{|c_2}$ is employed. This notation denotes the restriction of the record type T to class c_2. That is, given a record type T, $T_{|c_2}$ denotes T containing only those fields whose labels are proper attributes of class c_2. Thus, if $stype(c_2) = (a_1 : T_1, ..., a_n : T_n)$ and $stype(c_1) = (a'_1 : T'_1, ..., a'_k : T'_k)$ then $T' = (a''_1 : T''_1, ..., a''_n : T''_n)$ such that $a''_i \in \{a_1, ..., a_n\} \setminus \{a'_1, ..., a'_k\}$ and $a''_i = a_j$ and $T''_i \leq_T T_j$.

Note moreover that the `generalize` operation (upward migration) changes the type of the variable on which it is executed. Updates include method invocations, too. The typing rules for method invocations are given in Figure 3.3, and they apply both to side-effect and side-effect free method invocations.

$$\frac{c \in \mathcal{CI},\ stype(c) = T,\ t : T',\ O : \bar{c}}{\texttt{create}(c, t, O)} \quad T' \leq_T T,\ c \leq_T \bar{c},\ \bar{c} \in \mathcal{CI} \cup \{\perp\}$$

$$\frac{\texttt{create}(c, t, O),\ O :\perp}{O : c}$$

$$\frac{c \in \mathcal{CI},\ O : c}{\texttt{delete}(c, O)}$$

$$\frac{c \in \mathcal{CI},\ dom(a_i, c) = T_i,\ O : c,\ e : T'}{\texttt{modify}(c.a_i, O, e)} \quad T' \leq_T T_i,\ i = 1, \ldots, n$$

$$\frac{c_1\ e\ c_2 \in \mathcal{CI},\ O : c_1,\ stype(c_2) = T,\ t : T',\ c_2 \leq_T c_1}{\texttt{specialize}(c_1, c_2, O, t)} \quad T' \leq_T T_{|c_2}$$

$$\frac{\texttt{specialize}(c_1, c_2, O, t)}{O : c_2}$$

$$\frac{c_1\ e\ c_2 \in \mathcal{CI},\ O : c_1,\ c_1 \leq_T c_2}{\texttt{generalize}(c_1, c_2, O)}$$

$$\frac{\texttt{generalize}(c_1, c_2, O)}{O : c_2}$$

Figure 3.4: Update operation typing rules

3.2.3 Type Correctness of Method Implementations

In this section we discuss type checking of method implementations. We refer to a generic method implementation of the form:

$$m(P_1, ..., P_n) : R_1 || \ldots || R_m$$

where $P_1, ..., P_j$ are input parameters and $P_{j+1}, ..., P_n$ are output parameters. Suppose moreover that the signature of the method is:

$$sign(m, c) = T_1 \times \ldots \times T_j \rightarrow T_{j+1} \times \ldots \times T_n.$$

Note that variables can be assigned types in different ways within the body of a method. Variables in a method implementations can be categorized in two types: parameters and local variables. The *scope* of a type assignment in a method implementation is the whole implementation (that is, rules $R_1, \ldots, R_m$) for method parameters, whereas it is the single update rule (that is, R_i) for local variables.

For what concerns parameters, their type is determined by the method signature. Thus, the basis from which we start in type checking method implementations, is, besides $V \in Var$ for any variable V, the following, establishing the type of method parameters:

$$\frac{}{P_i : T_i} \quad 1 \leq i \leq n.$$

For what concerns local variables, by contrast, their type can be established in one of the following ways:

1. a class or type formula in the update rule condition explicitly specifies a type for the variable;

2. the variable is the output parameter of a method invocation contained in the condition or action of the update rule; in such a case its type is determined by the signature of the invoked method;

3. the variable is the output parameter of a `create` operation invoked in the action of the update rule; in such a case, its type is determined by the class on which the creation is executed.

Starting from one of the above criteria, a unique type is assigned to each variable, with respect to which the type checking of each update rule constituting the method implementation is performed.

An update rule is type correct if both its condition and its action are type correct. A method implementation is type correct if all the update rules constituting it are type correct.

The well-formed expressions of our language are all the method implementations that verify a number of static constraints. Those expressions are the ones for which we define the semantics. Intuitively, static constraints are the constraints a method implementation must satisfy to be correct, and that can be checked at compile time, rather than at execution time. Restricting the set of well-formed expressions of a language makes easier defining the semantics. Indeed, a number of error situations need not to be considered in the semantic evaluation, because they are excluded by static constraints. Typing constraints are not the only constraints that a method implementation must satisfy to be correct. In particular, the following additional static constraints must be satisfied by a method implementation for being correct:

1. The method implementations can be of two types:

   (a) each rule is without action part, then the method has no side-effects;

   (b) each rule has an action part, then the method has side-effects.

   Moreover, if a method has no side-effects, each redefinition of such method must be without side-effects. Likewise for methods with side-effects.

2. If a method is invoked in a condition it must be without side-effects. If a method is invoked in a method body it must be with side-effects.

3. A method invocation always conforms to one of the signatures of the method[4].

Finally, we remark that in what follows we will give the semantics of non-recursive method implementations. Thus, the graph, having as nodes methods names and an edge $m_i \rightarrow m_j$ if and only if m_j is invoked in the body of m_i, must be acyclic. For example, we do not consider methods whose corresponding graph has the form $m \rightarrow m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_n \rightarrow m$.

3.2.4 Type Correctness of a Class Implementation

We now analyze with a greater detail the rules that constitute a class implementation.

An attribute implementation is a set of rules of one of the following forms

$$Self.a = term \leftarrow body$$
$$term\ in\ Self.a \leftarrow body$$

where $Self$ is the pseudovariable used for denoting the object on which the rule is being executed.

A constraint implementation is a set of rules of the form

$$constraint\_formula \leftarrow body$$

where $constraint\_formula$ is a constraint name applied to a list of parameters, that is $con(t_1, \ldots, t_n)$.

We have already discussed the syntax and the typing constraints for operation implementation in the previous section.

[4]Several signatures may correspond to the same method.

Definition 3.9 *(Class Implementation - Revised). A class implementation is a set of deductive rules, each having one of the following forms*

1. *equality_formula* $\leftarrow$ *body*

   expressing the implementation of a derived attribute;

2. *membership_formula* $\leftarrow$ *body*

   expressing the implementation of a derived set valued attribute;

3. *constraint_formula* $\leftarrow$ *body*

   expressing a constraint

plus a set of operation implementations, namely update rules of the form

4. *op-name* : *condition* $\rightarrow$ *action.* □

Example 3.7 *The following is an example of class implementation.*

```
Self.age = X ← X = 1997 - Self.birthday.year
tooLowIncome(N) ← Self.income < 5000, N = Self.name
changeIncome(Amount): integer(New), New = Self.income + Amount
                       → modify(person.income,Self,New)
```
◇

As already stated, a number of conditions must be verified by a set of rules of this form to be a correct implementation for a class C. These conditions are as follows. First of all, the implementation must contain at least a rule for each class feature. Moreover,

1. *equality_formula* must be of the form

   $$Self.a = term$$

   with a derived attribute for the class whose implementation is being defined;

2. *membership_formula* must be of the form

   $$term\ in\ Self.a$$

   with a derived attribute for the class whose implementation is being defined;

3. *constraint_formula* must be constructed with a constraint of the class, applied to a number of parameters corresponding to that declared in the signature.

For operation implementation (4.) we require that *op-name* corresponds to an operation of the class, applied to a number of parameters corresponding to those declared in the signature.

Conditions 1—4 above, however, are not sufficient to guarantee that the set of rules provides a correct implementation for a class because they do not take into account type compatibility. Thus we have the following additional conditions related to typing of rules (typing rules for method implementations have already been considered in the previous section):

1. If the head of the rule is $Self.a = term$ and the type deduced for $term$, starting from the type declaration in rule body, is T, then $T \leq_T T^a$ must hold, where T^a is the domain specified for attribute a in the class signature;

2. if the head of the rule is $term\ in\ Self.a$ and the type deduced for $term$, starting from the type declaration in rule body, is T, then $set\text{-}of(T) \leq_T T^a$ must hold, where T^a is the domain specified for attribute a in the class signature;

3. if the head of the rule is $con(t_1, \ldots, t_n)$ and the type deduced for each t_i, starting from the type declaration in rule body, is T_i, $i = 1, \ldots, n$, then $T_i \leq_T T_i^c$ must hold, where the signature for constraint con in the class signature is $c \rightarrow T_1^c \times \ldots \times T_n^c$.

Example 3.8 *Consider the class signature of Example 2.10 and the class implementation of Example 3.7. Since pop = ext, no population implementation is provided. The only derived attribute in struct is (*`age, integer`*, der), whose implementation is specified by the rule*

`Self.age = X` $\leftarrow$ `X = 1997 - Self.birthday.year`.
Since Self denotes the object on which the rule is executed, it is implicitly typed to the class to which the rule belongs, in that case class `person`. *Thus, according to* `person` *signature,* `Self.birthday` *has type* `date`, *and, since type*(`date`)=`record-of (day:integer, month:integer, year:integer)`, *the term* `Self.birthday.year` *has type* `integer`. *Since also the value* `1997` *has type* `integer`, *the term* `1997 - Self.birthday.year` *has type* `integer` *too. Thus, variable* `X` *has type* `integer`, *which is exactly the type declared for attribute* `age`.

As far as operation implementation is concerned, operation `changeIncome:integer` $\rightarrow$ `person` *is implemented by the expression*

```
changeIncome(Amount): integer(New), New=Self.income + Amount →
                      modify(person.income,Self,New)
```

where the implicit output parameter is $Self$, *of type* `person`, *while the condition is well-typed for the input parameter* `Amount` *of type* `integer`.

Finally, consider the constraint `tooLowIncome:person` $\rightarrow$ `string`, *implemented by the rule*

`tooLowIncome(N)` $\leftarrow$ `Self.income <5000, N=Self.name`.
Since $Self$ *has type* `person`, `Self.income` *has type* `integer`, *while* `Self.name` *has type* `string`. *Thus, the formula in the rule body is well typed and* `N` *has type* `string`, *coherently with the constraint signature.* $\diamond$

3.3 Semantics

In this section we define the language semantics. We start by providing an intuitive idea, then we introduce some preliminary notions. Then, in giving the semantics we distinguish between two kinds of updates: atomic and non-atomic. An update is atomic when it cannot be decomposed into simpler updates. The atomic updates of our language are `create`, `delete`, `specialize`, `generalize`, `modify`, while non-atomic updates are method invocations. Atomic operations transform the database from one state to another state without intermediate states, whereas non-atomic updates require several intermediate states. Therefore, the semantics of atomic update operations is simpler than for the non-atomic operations.

3.3.1 Intuitive Idea

As we have seen, a method implementation in our language is an expression of the form:

$$\begin{aligned} m(P_1, \ldots, P_n) : \ & condition_1 \rightarrow action_1 || \\ & \vdots \\ & condition_m \rightarrow action_m \end{aligned}$$

where $condition_i$, $i = 1, \ldots, m$, is a declarative formula expressing control upon method execution, and each $action_i$, $i = 1, \ldots, m$, is an action list of the form $u_1^i; \ldots; u_k^i$, where each u_j^i, $j = 1, \ldots, k$ is an update operation. An update operation can be an update primitive of our language, or a method call. An expression $condition_i \rightarrow action_i$ in the above method implementation is called *update rule*. Moreover, $condition_i$ is called the *condition part* of the rule, whereas $action_i$ is called the *action part* of the rule. A method invocation (or method call) is an expression of the form:

$$O.m(t_1, \ldots, t_n)$$

where O is an object-denoting variable. For the sake of simplicity, we assume that the first parameters $P_1, \ldots, P_j$ $(j \leq 0)$ in a method implementation are the formal input parameters, and last $P_{j+1}, \ldots, P_n$ $(j \leq n)$ are the formal output parameters[5]. The actual parameters, $t_1, \ldots, t_n$, are terms. In particular, the actual output parameters, $t_{j+1}, \ldots, t_n$ are variables.

As it is usual in database systems, a method is invoked with respect to a database state, denoted by S_i, and a set B_i of bindings (that is, assignments of values to variables). S_i and B_i represent the state with respect to which the method is invoked. In particular, B_i represents the state of the variables of the program within which the method is invoked. This state together with the method behavior completely defines the semantics of the method execution[6].

The semantics of the execution of method m is informally defined as follows. When the method is invoked, first of all, the actual parameters $t_1, \ldots, t_j$ are evaluated and a new set of bindings B_{i_P} is generated. The generation of this set of bindings establishes the correspondence between the formal input parameters $P_1, \ldots, P_j$ and the values resulting from the evaluation of $t_1, \ldots, t_j$. Note that no bindings for the formal output parameters $P_{j+1}, \ldots, P_n$ are included in B_{i_P}. Then, each condition is evaluated against state S_i starting from the set of bindings B_{i_P}. We get as result the sets of bindings $B^1, \ldots, B^m$ which can be considered as extensions of the initial set B_{i_P}. Starting from the first rule, updates $u_1^1; \ldots; u_n^1$ are performed on state S_i with the set of bindings B^1, after the execution of u_n^1 the database state will be S_n^1. Likewise the second rule body is executed in the state S_n^1, with set of bindings B^2 and so on. After all rules are executed, the database state is in the final state, S_f which is returned as result of the method invocation. In the following we will specify also the final set of bindings B_f resulting from method invocation. We notice that, if an update operation u_i is a method invocation, its evaluation makes the database go through a sequence of intermediate states and gives as result a final set of bindings and a final state.

The semantics can thus be summarized as follows:

- Rules in a method implementation are executed sequentially, according to the order in which they appear in the implementation; rule R_{i+1} is then executed after rule R_i has completed its execution.
- All update operations in the same rule R_i are executed with respect to the same state. Such state is the state with respect to which R_i has been invoked. Note, therefore,

[5]Note that a method can have no input parameters, or no output parameters.

[6]We assume that concurrency control is used to synchronize accesses on the underlying database. Therefore, there is no interferences between concurrent executions.

that update operations within the same rule do not see each other modifications on the database state.

3.3.2 Preliminaries

As we have already mentioned, the semantics of a method invocation trasforms a database state and a set of bindings in a new database state and a new set of bindings. Thus, we have first of all to formally define the notion of database state and of set of bindings. We have already introduced the notion of database state in Chapter 1 (see Definition 2.36); however, since the update operations we consider only manipulate class extents and object states, for the sake of simplicitly we see the database state only as a pair $\langle \pi, \nu \rangle$ where π is the function assigning to each class its proper extent, whereas ν is the function assigning to each object its state.

For what concerns the notion of set of bindings, informally, we can state that a set of bindings B is a set of substitutions. This set is the mean by which the bindings for variables obtained by the evaluation of the condition are passed to the action part of the rule. Condition and action parts, indeed, may share some variables, in which case the action must be executed for every binding generated by the condition on the shared variables. We now formally define the notion of binding we will use in the subsequent discussion. First of all, we introduce the definition of substitution which is similar to the one generally adopted [62], revisited according to our needs. We recall that Var denotes the set of possible variables, while $\mathcal{V}$ denotes the set of possible values in our language.

Definition 3.10 *(Substitution and Set of Bindings). A substitution θ is a partial function from Var to $\mathcal{V}$; $\theta : Var \rightarrow \mathcal{V}$. A set of bindings B is a set of substitutions $\{\theta_1, \ldots, \theta_m\}$.* □

We notice that according to the previous definition, a substitution is a partial function. The domain of function θ is the set of possible variables of the language. However, given a formula, only a subset of the set of all variables Var is used in the formula. Therefore, for the variables appearing in the formula, function θ returns a value; for the remaining ones it is undefined ($\perp$). In the following, we often refer to substitutions as a subset of the cartesian product $Var \times \mathcal{V}$; such subset include only pairs for which θ is defined.

Intuitively, the set of bindings $B = \{\theta_1, \ldots, \theta_m\}$ satisfying a condition C, is the set of substitutions such that the application of each θ_i $(i = 1, \ldots, m)$ to C, denoted $C\theta_i$, is a ground formula which is true according to first order logic. Finally, we give the following definitions, which will be used in the remainder of the chapter.

Definition 3.11 *(Variable Interpretation with respect to a Set of Bindings). Given a variable X and a set of bindings B, X_B, denotes the interpretation of X in B, that is,* *the set of values or oids to which X is bound in B. Given $B = \{\theta_1, \ldots, \theta_m\}$, $X_B = \bigcup_{i=1}^{m} \{\theta_i(X)\}$.* □

Definition 3.12 *(Substitution Domain). Given a substitution θ, the substitution domain, denoted by $dom(\theta)$, is defined as the set $dom(\theta) = \{X \mid X \in Var \ \wedge \ \theta(X) \neq \perp\}$.* □

Definition 3.13 *(Restriction of a Substitution to a Set of Variables). Given a substitution θ and a set of variables $V \subseteq dom(\theta)$, $\theta_{|_V}$ denotes the restriction of θ to V, that is:*

$$\theta_{|_V}(X) = \begin{cases} \theta(X) & \text{if } X \in V \\ \perp & \text{otherwise} \end{cases}$$ □

Definition 3.14 *(Union of Substitutions). Let θ_i and θ_j be two substitutions such that $dom(\theta_i) \cap dom(\theta_j) = \emptyset$. The union of θ_i with θ_j, denoted as $\theta_i \cup \theta_j$, is defined as the following substitution:*

$$\theta_i \cup \theta_j(X) = \begin{cases} \theta_i(X) & \text{if } X \in dom(\theta_i) \\ \theta_j(X) & \text{if } X \in dom(\theta_j) \\ \perp & \text{otherwise} \end{cases}$$ □

Example 3.9 *Given $X, Y, Z, W \in Var$ and a set of values $\mathcal{V}$ including integers, the following are examples of substitutions:*

$$\theta_1 = \{X/5, Y/7, Z/8\} \qquad \theta_2 = \{X/7, W/10\} \qquad \theta_3 = \{W/8\}$$

Given $B = \{\theta_1, \theta_2\}$, $X_B = \{5, 7\}$. Moreover, $dom(\theta_1) = \{X, Y, Z\}$, $dom(\theta_2) = \{X, W\}$, $dom(\theta_3) = \{W\}$. Finally, $\theta_{1|_{\{X,Z\}}} = \{X/5, Z/8\}$ whereas $\theta_1 \cup \theta_3 = \{X/5, Y/7, Z/8, W/8\}$. ◇

We now introduce semantic domains. In defining semantics domains we refer to well-formed syntactic objects, that is, objects which meet the given static constraints and for which a semantics can be defined.

Definition 3.15 *(Semantic Domains). The semantic domains we consider in giving the semantics are the following:*

- *$Bind$ = set of possible sets of bindings.*
- *$State$ = set of possible states of a schema.*

- *Term = set of the well-formed terms of the language.*
- $2^{\mathcal{V}}$ *= powerset of* $\mathcal{V}$*, set of possible values of the language.*
- *Cond = set of the well-formed declarative expressions of the language.*
- *Update = set of the well-formed update sequences of the language.* □

Definition 3.16 *(Semantics). The semantics of the method language is a family of functions defined as follows:*

$$\mathcal{E} : (Term \times State) \to ((Bind \times State) \to 2^{\mathcal{V}} \cup \{\langle \emptyset, State \rangle\})$$
$$\mathcal{C} : (Cond \times State) \to ((Bind \times State) \to Bind \cup \{\langle \emptyset, State \rangle\})$$
$$\mathcal{U} : (Update \times State) \to ((Bind \times State) \to (Bind \times State))$$
□

We notice that the functions receive as arguments pairs of form $\langle$*construct to be evaluated*, *state*$\rangle$. The state is necessary because in case of run-time errors the initial database state must be restored to maintain the database consistent. By contrast, when run-time errors arise, an empty set ($\emptyset$) of bindings is returned, and, thus, the variables are not bound to any value. Semantic functions, in turn, return functions as result.

In what follows, we specify semantic functions $\mathcal{C}$ and $\mathcal{U}$; due to space limitations, we do not present here the definition of $\mathcal{E}$ since it is the quite standard interpretation of a term, given a set of bindings. We refer the interested reader to [35] for a definition of $\mathcal{E}$.

In the following, we first define the semantics of atomic updates and then we deal with the semantics of method calls.

3.3.3 Semantics: Atomic Updates

Figures 3.5 and 3.6 specify the semantics for each atomic update operation.

Consider first of all the operation `create`($\bar{c}$, $\bar{t}$, $\bar{O}$) executed on a set of bindings $B = \{\theta_1, ..., \theta_m\}$ in a state S. The intuitive idea of the `create` operation is to create as many distinct objects as the different evaluations of the record $\bar{t}$ with respect to the set of variables appearing in $\bar{t}$. Each of such evaluations represents the state of the new, just created, objects.

Given the set V of the variables appearing in term $\bar{t}$, consider the set $B_V = \{\theta_{1_{|V}}, \ldots, \theta_{m_{|V}}\}$. It can happen that two restricted substitutions, say $\theta_{i_{|V}}$ and $\theta_{j_{|V}}$, are equal even if $\theta_i \neq \theta_j$. As a consequence, the set B_V includes only k substitutions, $k \leq m$, rather than m; that

$\mathcal{U}\,[\![\, \texttt{create(}\bar{c},\, \bar{t},\, \bar{O}\texttt{)}_{S_0} \,]\!]\, BS = \langle B', S' \rangle$

$B' = \{\theta_1 \cup \langle \bar{O}/oid_{f(1)} \rangle,\, \ldots,\, \theta_m \cup \langle \bar{O}/oid_{f(m)} \rangle\}$, where:

$f : \{1, \ldots, m\} \to \{1, \ldots, k\}$, $k \leq m$, is a function such that $f(i) = j$ if $\theta_{i_{|V}} = \sigma_j$, where $\{\sigma_1, \ldots, \sigma_k\} = \{\theta_{i_{|V}} \mid i = 1, \ldots, m\}$ and $\{oid_1, \ldots oid_k\} = OIDNEW$

$S' = \langle \pi', \nu' \rangle$, where:

$$
\begin{array}{ll}
\pi'(c) = \pi(c) & \forall\, c \in \mathcal{CI} \textit{ such that } c \neq \bar{c} \\
\pi'(c) = \pi(c) \cup OIDNEW & c = \bar{c} \\
 & \\
\nu'(oid) = \nu(oid) & \forall\, oid \notin OIDNEW \\
\nu'(oid_i) = \mathcal{E}\,[\![\, \bar{t} \,]\!]\, \sigma_i S & \forall\, oid_i \in OIDNEW,\ 1 \leq i \leq k
\end{array}
$$

$\mathcal{U}\,[\![\, \texttt{delete(}\bar{c},\, \bar{O}\texttt{)}_{S_0} \,]\!]\, BS = \langle B', S' \rangle$

$B' = B$ and $S' = \langle \pi', \nu' \rangle$, where:

$$
\begin{array}{ll}
\pi'(c) = \pi(c) & \forall\, c \in \mathcal{CI} \textit{ such that } c \neq \bar{c} \\
\pi'(c) = \pi(c) \setminus \bar{O}_B & c = \bar{c} \\
 & \\
\nu'(oid) = \nu(oid) & \forall\, oid \notin \bar{O}_B \\
\nu'(oid) = \perp & \forall\, oid \in \bar{O}_B
\end{array}
$$

$\mathcal{U}\,[\![\, \texttt{specialize}(c_1, c_2, \bar{O}, \bar{t})_{S_0} \,]\!]\, BS = \langle B', S' \rangle$

$B' = B$ and $S' = \langle \pi', \nu' \rangle$, where:

$$
\begin{array}{ll}
\pi'(c) = \pi(c) & \forall\, c \in \mathcal{CI} \textit{ such that } c \neq \bar{c}_1 \textit{ and } c \neq \bar{c}_2 \\
\pi'(c) = \pi(c) \setminus \bar{O}_B & c = c_1 \\
\pi'(c) = \pi(c) \cup \bar{O}_B & c = c_2 \\
 & \\
\nu'(oid) = \nu(oid) & \forall\, oid \notin \bar{O}_B \\
\nu'(oid) = \nu(oid) \bar{\cup} \mathcal{E}\,[\![\, \bar{t} \,]\!]\, \theta_i S & \forall\, oid \in \bar{O}_B \textit{ such that } \theta_i(\bar{O}) = oid
\end{array}
$$

Figure 3.5: Semantics of atomic update operations

$\mathcal{U}[\![\texttt{ generalize(}c_1, c_2, \bar{O})_{S_0}]\!] BS = \langle B', S' \rangle$

$B' = B$ and $S' = \langle \pi', \nu' \rangle$, where:

$$\begin{array}{ll} \pi'(c) = \pi(c) & \forall\ c \in \mathcal{CI}\ \textit{such that}\ c \neq \bar{c}_1\ \textit{and}\ c \neq \bar{c}_2 \\ \pi'(c) = \pi(c) \setminus \bar{O}_B & c = c_1 \\ \pi'(c) = \pi(c) \cup \bar{O}_B & c = c_2 \\ & \\ \nu'(oid) = \nu(oid) & \forall\ oid\ \notin\ \bar{O}_B \\ \nu'(oid) = \nu(oid)_{|_{c_2}} & \forall\ oid\ \in\ \bar{O}_B\ \textit{such that}\ \theta_i(\bar{O}) =\ oid \end{array}$$

$\mathcal{U}[\![\texttt{modify(}\bar{c}.a,\ \bar{O},\ \bar{t}\texttt{)}_{S_0}]\!] BS = \langle B', S' \rangle$

$B' = \emptyset$ and $S' = S_0$ if $\exists\ \theta_i, \theta_j \in B$ such that $\theta_i(\bar{O}) = \theta_j(\bar{O})$, but $\mathcal{E}[\![\bar{t}_{S_0}]\!] \theta_i S \neq \mathcal{E}[\![\bar{t}_{S_0}]\!] \theta_j S$

otherwise $B' = B$ and $S' = \langle \pi', \nu' \rangle$, where:

$$\begin{array}{ll} \pi'(c) = \pi(c) & \forall\ c \in \mathcal{CI} \\ & \\ \nu'(oid) = \nu(oid) & \forall\ oid\ \notin\ \bar{O}_B \\ \nu'(oid).a_k = \nu(oid).a_k & \forall\ oid\ \in\ \bar{O}_B\ \textit{and}\ a_k \neq a \\ \nu'(oid).a_k = v & \forall\ oid\ \in\ \bar{O}_B,\ a_k = a\ \textit{and if}\ \theta_i(\bar{O}) =\ oid \\ & \textit{then}\ \mathcal{E}[\![\bar{t}_{S_0}]\!] \theta_i S = \{v\} \end{array}$$

Figure 3.6: Semantics of atomic update operations (continued)

is, $B_V = \{\sigma_1, \ldots, \sigma_k\}$. Therefore, the evaluation of $\bar{t}$ in $\langle B_V, S\rangle$ produces k records, one for each substitution in B_V. Then, k distinct objects are created, each with state $t_i = \mathcal{E}[\![\bar{t}]\!]\sigma_i S$, $i = 1, \ldots, k$. Two distinct objects with the same state can, however, be created, when two t_i's ($i = 1, \ldots, k$) are equal, but these t_i's result from the evaluation of $\bar{t}$ with respect to two different substitutions in B_V. Therefore, these t_i's must correspond to different *oids*. Let $OIDNEW = \{oid_1, \ldots, oid_k\}$ be the set of the oids of the newly, just created, objects. The idea is to add to each $\theta_i \in B$ the pair $\langle \bar{O}/oid_i\rangle$ where oid_i is the object whose state is given by $\mathcal{E}[\![\bar{t}]\!]\theta_{i_{|V}} S$. To model this correspondence we introduce a function $f : \{1, \ldots, m\} \to \{1, \ldots, k\}$ such that $f(i) = j$ if $\theta_{i_{|V}} = \theta_j$, that is, $\mathcal{E}[\![\bar{t}]\!]\theta_{i_{|V}} S = t_j$.

We remark that in the state S which precedes the `create` operation, the state of the *oids* assigned to the new objects is undefined, that is, $\forall\, oid_i \in OIDNEW$, $i = 1, \ldots, k$, $\nu(oid_i) = \perp$. Moreover, each new object state t_i can be the result of the evaluation of record $\bar{t}$ with respect to different substitutions θ_i. In particular if each evaluation of $\bar{t}$ in B returns a different value, then the number of created objects is equal to the number of substitutions in B, i.e. m, and the state of the objects would be defined as $\nu'(oid_i) = \mathcal{E}[\![\bar{t}]\!]\theta_i S$. Finally, note that the `create` operation returns $\bar{O}$ as output parameter, bound to the *oids* of the created objects.

Consider now the operation `delete(`$\bar{c}$`,` $\bar{O}$`)` executed on a set of bindings B in a state S. The `delete` operation deletes all the objects which are bound to the variable $\bar{O}$ in B, that is $\bar{O}_B$. The set of bindings B remains unchanged, but the new state of the deleted objects is undefined, that is, $\nu'(oid) = \perp, \forall\, oid \in \bar{O}_B$. This is in agreement with the idea we have of B and S: if an update returns output variables (as in the `create` operation), both B and S change, otherwise only the state is modified (as in the `delete` operation). Moreover, in general, the *oids* such that $\nu(oid) = \perp$ are the ones that are not currently assigned to any object. They can thus be assigned to new objects. Hence, in our language, *oids* of deleted objects can be reassigned.

Consider now the operation `specialize` $(c_1, c_2, \bar{O}, \bar{t})$ executed on a set of bindings $B = \{\theta_1, ..., \theta_m\}$ in a state S. The intuitive idea is that objects to which variable $\bar{O}$ is bound migrates from class c_1, to which they belong, to class c_2. $\bar{t}$ is the record term providing the values for new attributes acquired by such objects. In the semantic clauses the symbol $\bar{\cup}$ denotes record concatenation. Note that the objects in the set $\bar{O}_B$ become instances of class c_2, but they are still members of class c_1. The new state of each object is obtained by adding to the old state the record $\mathcal{E}[\![\bar{t}]\!]\theta_i S$, containg the values for proper attributes of c_2.

Consider the operation `generalize` $(c_1, c_2, \bar{O})$ executed on a set of bindings $B = \{\theta_1, ..., \theta_m\}$ in a state S. The intuitive idea is that objects to which variable $\bar{O}$ is bound migrates from class c_1, to which they belong, to class c_2. In the semantic clauses $nu(oid)_{|_{c_2}}$ denotes the restriction of $\nu(oid)$ to attributes of class c_2. That is, $\nu'(oid).a = \nu(oid).a$ if $a \in A(c_2)$, otherwise $\nu'(oid).a = \perp$. Note that the objects in the set $\bar{O}_B$ become instances of class c_2, while they are removed from the extent of class c_1.

Finally, consider the operation `modify(`$\bar{c}.a$`,` $\bar{O}$`,` $\bar{t}$`)` executed on a set of bindings $B = \{\theta_1, ..., \theta_m\}$ in a state S. Two cases can be distinguished. If term $\bar{t}$ evaluates to different values corresponding to two different substitutions $\theta_i, \theta_j \in B$ which assign the same value to variable $\bar{O}$, then an error situation occurs and the initial database state is restored. In that case, indeed, variable $\bar{O}$ has the same value, say object o, in the two substitions, whereas $\bar{t}$ has different values, say v_1 and v_2. Thus it is not possible to decide which value, between v_1 or v_2, to assign to attribute a of object o. If no problems of this kind arise, the value resulting from the evaluation of $\bar{t}$ is assigned to attribute a of the object to which $\bar{O}$ evaluates. Thus, this semantics model intrinsic ambiguity problems and clearly states that the evaluation of the term $\bar{t}$ must be in the state preceding the operation execution.

3.3.3.1 Semantics of Atomic Updates under Referential Integrity, Exclusivess and Dependency Constraints

In what follows we consider how the semantics of atomic update operations is influenced by referential integrity, exclusiveness and dependency constraints. We first consider creation and modification, and then deletion. Note that, for what discussed in this section, a specialization can be seen as a partial creation, whereas a generalization can be seen as a partial deletion,

Creation and Modification The creation of a new object may cause the violation of composite exclusiveness constraints. The creation is thus allowed if for any composite reference originating from the newly created object, exclusiveness is not violated. Moreover, the creation may violate referential integrity if the object specified as values for an object attribute does not exist in the database. If any of these constraints is violated, the creation is disallowed, that is, it is backed out. In those cases, the state resulting from the execution of the operation is state S_0, The following rule states the conditions upon which creation is allowed.

Rule 3.1 *(Creation Rule). The creation of object o is allowed provided that:*

- Referential Integrity*: for each object* $o' \in ref(o)$, $o' \in OBJ$ *must hold, that is, each object referenced by o must exist;*
- Exclusiveness*: for each object* $o' \in comp(o)$*:*

- *if o' is the value for a composite attribute a of o, exclusive with respect to a set of classes S, then no object o^* in $\pi^*(S)$ must exist such that $o' \in comp(o^*)$ and no attribute a' must exist such that $\nu(o).a' = o'$;*
- *if o' is the value for a composite attribute a of o, exclusive on the attribute with respect to a set of classes S, then no object o^* in $\pi^*(S)$ must exist such that $\nu(o^*).a = o'$;*
- *if o' is the value for a composite non-exclusive attribute a of o, then no other object o^* must exist such that $o' \in comp(o^*)$, o' is the value for attribute a^* of o^* exclusive for o^* with respect to a set classes S such that $o \in \pi^*(S)$;*
- *if o' is the value for a composite non-exclusive attribute a of o, then no other object o^* must exist such that $o' \in comp(o^*)$, o' is the value for attribute a of o^* exclusive on the attribute for o^* with respect to a set classes S such that $o \in \pi^*(S)$.* □

The modification of an object can be dealt with as the creation of a new object. We recall that the *OID* of an object cannot be modified and that the modification of an object cannot result in object deletion. Therefore, the only constraints potentially violated by a modification are those related to exclusiveness and referential integrity. If those constraints are violated, the modification is simply disallowed (backed out).

Deletion Object deletion must be carefully handled in presence of composite objects, because semantic ambiguities due to the different options may arise. In formulating the strategies for solving those ambiguities we follow the basic idea of preserving the information contained in the database from non explicitly requested deletion, and thus, in ambiguous cases, we have chosen not to delete an object.

Let us first show, by means of examples, why deletion may result in ambiguities, and then present the semantics we have defined to capture the desired behavior.

Example 3.10 *Consider the objects in Figure 3.7, such that a_del of attribute a is* cascade *whereas a_del of attribute a' is* restrict*. When object o is deleted, the* cascade *option specified for a requires the deletion of o_1. By contrast, the* restrict *option specified for a' states that o_1 cannot be deleted as long as it is the value of attribute a' of an object (o_2). Thus, an ambiguity arises. We adopt a conservative approach and thus we choose not to delete object o_1. Thus, the* restrict *option of a' prevails on the* cascade *option of a.* ◇

Example 3.11 *Consider the objects in Figure 3.8, such that a_del of attributes a, a_1, a_3 is* cascade*, whereas a_del of attribute a_2 is* restrict*. When object o is deleted, two different possible final states can be obtained depending on the order in which attributes whose value is o are considered.*

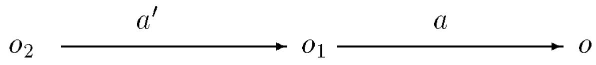

Figure 3.7: Objects of Example 3.10

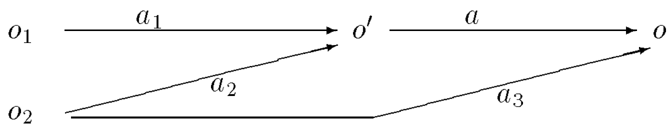

Figure 3.8: Objects of Example 3.11

- *If a_3 is considered first, o_2 is deleted, since a* cascade *option is specified for a_3 (note that, as a consequence, the reference from o_2 to o' is also removed). Then, a is considered and o' and o_1 are also deleted, because of the* cascade *option specified for a and a_1. Thus, all the four objects are deleted.*

- *If a is considered first, o' is not deleted because it is the value for an attribute (a_2) for which a* restrict *option is specified. Then, a_3 is considered and o_2 is deleted because of the* cascade *option specified foir a_3. This deletion does not cause the deletion of any other object. Thus, the deletion of o only causes the deletion of o_2.*

Note that the final state which more closely reflects the intuitive semantics is the first one, because the object preventing the deletion of o', namely o_2, is in turn deleted as a consequence of the deletion of o. $\diamond$

A similar situation may arise because of dependent declarations, as shown by the following example.

Example 3.12 *Consider the objects in Figure 3.9, such that the a_dep fields of a and a_1 are* true*, whereas the a_dep field of a_2 is* false*. When object o is deleted, two different possible final states can be obtained depending on the order in which o attributes are considered.*

- *If a_1 is considered first, o'' is deleted, because attribute a_1 is dependent, and no other reference to o'' exists (note that, as a consequence, the reference from o'' to o' is also removed). Then, a is considered and o' is also deleted, because a is dependent and no more references to o' exist. Thus, all three objects are deleted.*

- *If a is considered first, o' is not deleted because it is the value of attribute a_2. Then, a_1 is considered and o'' is deleted. Thus, the deletion of o only causes the deletion of o''.*

$$\begin{tikzcd} & o'' \arrow[dr, "a_2"] & \\ o \arrow[ur, "a_1"] \arrow[rr, "a"'] & & o' \end{tikzcd}$$

Figure 3.9: Objects of Example 3.12

Note that the final state which more closely reflects the intuitive semantics is the first one, because the object to which o' belongs as a component (i.e. o'') is in turn deleted as a consequence of the deletion of o. $\diamond$

As shown by the above examples, it is important to establish a well-founded semantics for delete operations on composite objects. Such semantics is defined in what follows. We start from the idea that, given an object o, we want to determine the set, denoted as Set^d, of the objects that have to be deleted as a consequence of the deletion of o. Obviously, before determining Set^d, it must be verified whether object o itself can be deleted, that is, whether no references to o[7] exists, through an attribute a such that a_del of a is **restrict**.

The set Set^d initially contains only the object to be deleted, that is, $Set^d = \{o\}$. Other objects are then inserted into this set, because of **dependent** declarations or **delete** declarations, according to what follows.

- **dependent** declarations:

  Each object o' referenced through a composite attribute by an object o in Set^d such that all the references leading to o' originate from objects in Set^d or from object o' itself, and at least one of these references has a **dependent** declaration, is added to Set^d.

- **delete** declarations:

  Each object o' referenced by an object o in Set^d through an attribute a such that a_del of a is **cascade**, and such that for all other references through attribute a' to o' either a_del of a' is not **restrict** or a' originates from some object in Set^d or from object o' itself, is added to Set^d.

Now we introduce two operators formalizing the above concepts.

Definition 3.17 ($\vdash^d$). *Let o, $o' \in OBJ$ be two objects. o' is said to be deleted for dependence from o, denoted as $o \vdash^d o'$, if $\nu(o).a = o'$, a_dep of a is* **true** *and for any other attribute a' such that $o' = \nu(o^*).a'$ either $o^* = o'$ or $o^* \in Set^d$.* $\square$

[7]This means that an object o' exists such that $\nu(o').a = o$.

Definition 3.18 ($\vdash^r$). *Let o, $o' \in OBJ$ be two objects. o' is said to be deleted for referential integrity from o, denoted as $o \vdash^r o'$, if $\nu(o').a = o$, a_del of a is* cascade *and for any other attribute a' such that $o' = \nu(o^*).a'$ and a_del of a' is* restrict, *either $o^* = o'$ or $o^* \in Set^d$.* □

We now introduce an operator which is the union of $\vdash^d$ and $\vdash^r$.

Definition 3.19 (DEL). *Operator* DEL : $2^{OBJ} \to 2^{OBJ}$ *denotes the operator which returns the objects to be deleted (either for dependence or for referential integrity) from a given set of objects. Given a set of objects $O \subseteq 2^{OBJ}$*

$$\mathsf{DEL}(O) = O \cup \{o' \mid \exists\, o \in O \quad o \vdash^d o'\} \cup \{o' \mid \exists\, o \in O \quad o \vdash^c o'\}. \qquad \square$$

We are now able to formally specify the semantics of the delete operation, relying on the notion of *fixpoint*. According to [44], given a poset $(A, \leq)$, where A is a set and $\leq$ is a partial order on A, and given a monotonic transformation T on the complete poset $(A, \leq)$, T has a least fixpoint. In our context, $(A, \leq)$ is $(\mathcal{P}(OBJ)^o, \subseteq)$, where $\mathcal{P}(OBJ)^o$ is the set of all possible sets of objects containing object o[8], "$\subseteq$" is set inclusion on $\mathcal{P}(OBJ)^o$ and the monotonic transformation T is DEL. $(\mathcal{P}(OBJ)^o, \subseteq)$ is a complete poset since for each subset O of $\mathcal{P}(OBJ)^o$ both the least upper bound (lub(O)) and the greatest lower bound (glb(O)) exist. Finally, DEL is a monotonic operator on $(\mathcal{P}(OBJ)^o, \subseteq)$. Indeed, for each $I, J \in \mathcal{P}(OBJ)^o$, $I \subseteq J$, $\mathsf{DEL}(I) \subseteq \mathsf{DEL}(J)$, because the DEL operator applied to set J (where J contains all the objects in I and eventually some other objects) returns all the objects in $\mathsf{DEL}(I)$ and eventually some more objects that can be deleted starting from J. Thus, the monotonic operator DEL has a least fixpoint, denoted as $lfp(\mathsf{DEL}) = Set^d$. An element $O \in \mathcal{P}(OBJ)^o$ is said a *fixpoint* of DEL if $\mathsf{DEL}(O) = O$.

Definition 3.20 *Let $o \in OBJ$ be an object, the effect of the deletion of o in our object model is the set $Set^d \in \mathcal{P}(OBJ)^o$ such that $Set^d = lfp(\mathsf{DEL})$.* □

The algorithm to compute the solution of the equation $Set^d = \mathsf{DEL}(Set^d)$ is given in Figure 3.10. The proposed algorithm is a *naive fixpoint*, that is, it naively computes the fixpoint of operator DEL.

The following example shows how the defined semantics correctly handles the "ambiguous" situation of Example 3.11.

Example 3.13 *Referring to objects and relationships of Example 3.11, the various iterations of the algorithm lead to the following sets:*

[8] $\mathcal{P}(OBJ)^o$ consists of the sets of objects O belonging to 2^{OBJ} such that $o \in O$.

Input: o, object to be deleted.
Output: Set^d, objects to be deleted from o.
Method: $Set^d := \{o\}$;
$Set^d_N := \mathsf{DEL}(Set^d)$;
while ($Set^d \subset Set^d_N$) **do**
begin
$Set^d := Set^d_N$;
$Set^d_N := \mathsf{DEL}(Set^d)$
end

Figure 3.10: Deletion algorithm

$\mathrm{Set}^d = \{o\}$	
$\mathrm{Set}^d = \{o, o_2\}$	*for* $o \vdash^r o_2$
$\mathrm{Set}^d = \{o, o_2, o'\}$	*for* $o \vdash^r o'$
$\mathrm{Set}^d = \{o, o_2, o', o_1\}$	*for* $o' \vdash^r o_1$

The set $\{o, o_2, o', o_1\}$ *is the effect of the deletion of* o, *because* $\mathsf{DEL}(\{o, o_2, o', o_1\}) = \{o, o_2, o', o_1\}$. *In the fixpoint computation, note that at the second iteration,* o' *has not been inserted in* Set^d *because of a reference to it with delete* restrict *(i.e.* r_2*), whereas in the third iteration* o' *is inserted because the origin of that reference (i.e.* o_2*) belongs to* Set^d. ◇

The following example illustrates the semantics of deletion in a more complex example.

Example 3.14 *Consider the objects in Figure 3.11. For the sake of clarity, instead of the name of the attributes we have depicted their* delete *and* dependent *options. Suppose that object o is deleted (o can be deleted because there are no references leading to it with delete* restrict*). The various iterations of the algorithm lead to the following sets:*

$Set^d = \{o\}$	
$Set^d = \{o, o_1\}$	*for* $o \vdash^r o_1$
$Set^d = \{o, o_1, o_2, o_3\}$	*for* $o_1 \vdash^r o_3$ *and* $o_1 \vdash^d o_2$
$Set^d = \{o, o_1, o_2, o_3, o_6\}$	*for* $o_3 \vdash^r o_6$

The set $\{o, o_1, o_2, o_3, o_6\}$ *is the effect of the deletion of* o, *because* $\mathsf{DEL}(\{o, o_1, o_2, o_3, o_6\}) = \{o, o_1, o_2, o_3, o_6\}$.

This correctly models the semantics specified for attributes. Indeed:

- *object o can be deleted, because there is no reference with delete* restrict *to it;*

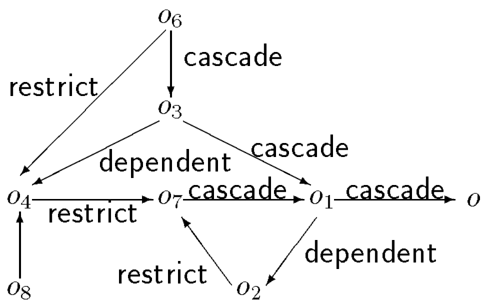

Figure 3.11: Objects of Example 3.14

- *as a consequence of the deletion of object o, object o_1 is deleted, because of a reference with delete* cascade *leading to o, and because there are no references with delete* restrict *to o_1;*
- *as a consequence of the deletion of object o_1, objects o_3 and o_7 (due to a reference with delete* cascade *leading to o_1) and object o_2 (due to a composite dependent reference from o_1) could be deleted;*
  - *o_2 is deleted, because there are no more references to it;*
  - *o_3 is deleted, because there are no references to it with delete* restrict*;*
  - *o_7 is not deleted, because there is a reference to it (from o_4) with delete* restrict*;*
- *the deletion of o_2 does not cause the deletion of any other object, whereas the deletion of o_3 could cause the deletion of o_6 (due to a reference with delete* cascade *to o_3) and of o_4 (due to a composite dependent reference from o_3);*
  - *o_6 is deleted, because there are no references to it with delete* restrict*;*
  - *o_4 is not deleted, because there is a reference to it (from object o_8);*
- *the deletion of o_6 does not cause the deletion of any other object.* $\diamond$

3.3.4 Semantics: Method Calls

Method calls are the unique non-atomic updates of our language. As we said before, a method invocation has the form $O.m(t_1, \ldots, t_n)$ where $t_1,\ldots,t_n$ are terms. Moreover, the former j parameters are assumed to be input parameters and the remaining parameters are assumed to be output parameters. $t_1, \ldots, t_j$ are thus terms which represent the values associated with the formal input parameters; $t_{j+1}, \ldots, t_n$ are, by contrast, variables.

A method call is implemented by several rules, where each rule has the form: *condition* $\rightarrow$ $u_1; \ldots; u_n$. Hence we first formalize the semantics of update concatenation. Then, to fully formalize the semantics of method calls, we define the semantic function of conditions $\mathcal{C}$ and the semantics of each rule which implements the method body and specify how such rules interact among each other.

3.3.4.1 Semantics: Concatenation

The semantics of concatenation is defined inductively on the number of updates in the sequence as follows.

Definition 3.21 *(Semantics of Concatenation). Let $u_1, \ldots, u_n$ be updates according to Definition 3.5, let B be a set of bindings and S be a database state.*

$$\mathcal{U}\,[\![\, u_1; u_2; \ldots; u_n \,]\!]\, BS = \begin{cases} \langle \emptyset, S_0 \rangle & \textit{if } \mathcal{U}\,[\![\,(u_1)_{S_0}\,]\!]\, BS = \langle \emptyset, S_0 \rangle \\ \mathcal{U}\,[\![\,(u_2; \ldots; u_n)_{S_0}\,]\!]\,(\mathcal{U}\,[\![\,(u_1)_{S_0}\,]\!]\, BS) & \textit{otherwise} \end{cases}$$

$\square$

The above definition is an inductive definition where the basis ($n = 1$) consists of sequences composed by a single update. Each update in turn can be an atomic update, whose semantics is specified according to Figure 3.5, or a method invocation, whose semantics is specified according to Definition 3.26 (defined later on).

3.3.4.2 Semantics: Conditions

The semantic domain of function $\mathcal{C}$ is the set of well-formed formulas of our language. Intuitively the evaluation of a condition on a set of bindings B and in a state S, returns the subset of the cartesian product of the variables appearing in the formula and satisfying the formula itself, as specified by the following definition.

Definition 3.22 *(Semantics of Conditions). Let C be a formula according to Definition 3.3, let $B = \{\theta_1, ..., \theta_m\}$ be a set of bindings and S be a database state. The evaluation of condition C in $\langle B, S \rangle$ is defined as*

$$\mathcal{C}\,[\![\, C \,]\!]\, BS = \begin{cases} \langle \emptyset, S_0 \rangle & \textit{if run-time errors occur in the evaluation of } C \\ \bigcup_{i=1}^{m} \mathcal{C}\,[\![\, C \,]\!]\, \theta_i S & \textit{otherwise} \end{cases}$$

where $\mathcal{C}\,[\![\, C \,]\!]\, \theta_i S = \{\theta_1^i, \ldots, \theta_k^i\}$ *and* $\forall j = 1, \ldots, k$*:*

- $\theta_i \subseteq \theta^i_j$, $i = 1, \ldots, m$, *that is, the substitutions* θ^i_j *coincides with* θ_i *on the variables for which* θ_i *is defined,*
- $C\theta^i_j$, $i = 1, \ldots, m$, *is a ground formula which evaluates to true in the state* S. $\square$

Note that the condition evaluation returns $\langle \emptyset, S_0 \rangle$ when a run-time error (as a division by zero) occurs in the condition evaluation. Otherwise, the result of the evaluation of the condition C is a set of substitutions, each grounding for C and such that the instantiation of C with respect to each of them, evaluates to true in state S according to first order logic. Each substitution $\theta_i \in B$ partially instantiates the formula C, while substitutions $\theta^i_1, \ldots, \theta^i_k$ are extensions of θ_i specifying the bindings for the variables in C non-bound by θ_i. More formally, if $Var(C)$ denotes variables appearing in condition C and $dom(\theta)$ denotes the domain of substitution θ, according to Definition 3.12, then $dom(\theta^i_j) = Var(C) \cup dom(\theta_i)$.

Example 3.15 *Consider the simple class formula* $c(X)$. $\mathcal{C} [\![c(X)_{S_0}]\!] BS = B'$ *where* $B = \{\theta_1, ..., \theta_m\}$ *and* X *does not appear in* B. $B' = \{\theta_1 \cup (X/oid_1), \ldots, \theta_1 \cup (X/oid_n), \ldots, \theta_m \cup (X/oid_1), \ldots, \theta_m \cup (X/oid_n)\}$ *where* $S.\pi^*(c) = \{oid_1, \ldots, oid_n\}$. *In this case, since* $Var(C) \cap dom(\theta_i) = \emptyset$, $i = 1, \ldots, m$, *the set of substitutions generated for each* $\theta_i \in B$ *is the same, i.e.* $\{\langle X/oid_1 \rangle, \ldots, \langle X/oid_n \rangle\}$. *For each substitution* θ'_i *belonging to* B' $(c(X))\theta'_i = c(\theta'_i(X)) = c(oid) = true$ *since* $oid \in S.\pi(c)$ *for definition of* B'. $\diamond$

Based on first order logic, the semantics of conditions can be easily formalized. Due to space limitations, we do not introduce here all the clauses, rather we focus on the semantics of other constructs of our language for which a formalism so well-known as first order logic does not exist.

3.3.4.3 Semantics: Method Calls

In formalizing the semantics of method calls, we can distinguish four phases:

1. Selection of method implementation.
2. Generation of the set of bindings in which the rule conditions are evaluated.
3. Evaluation of the method implementation.
4. Generation of the set of bindings to return as result.

Phase 3 above is different for side-effect free methods and for methods with side-effects, whereas the other phases are not affected by such difference. In the following, we discuss in detail the various phases.

Selection of Method Implementation The first problem arising in the definition of the semantics of method calls is to establish which method has to be invoked for each *oid* bound to the variable, on which the method is invoked. Our language supports *late binding*, that is, at run-time, for each object the method related to one of its most specific classes is invoked. In what follows d denotes the dispatching function, whose definition is: given the object O and the method m, $d(O, m){=}c$ where c is the class whose implementation for m is selected for execution. This dispatching function will be specified in Chapter 6.

Late binding, however, complicates the semantics of set-oriented method calls, especially for methods with side-effects, as we discuss in what follows. Suppose to invoke method m on object O, with respect to a set of bindings B, then the set $O_B = \{oid_1, \ldots, oid_m\}$ can be partitioned into several disjoint subsets of *oids* according to the method implementation selected for each object. Therefore, each subset contains the *oids* of objects for which the same method implementation is considered, as stated by the following definition.

Definition 3.23 *(Partitioning). Let $B = \{\theta_1, ..., \theta_m\}$ be a set of bindings. The dispatching function d induces a partitioning $P(B) = \{B_1 \ldots, B_k\}$ of B such that, for $j = 1, \ldots, k$, $B_j = \{\theta_i \mid \theta_i \in B \land d(\theta_i(O), m) = c_j\}$*[9]. □

According to the set-oriented approach, given the invocation $O.m(t_1, \ldots, t_n)$ to be executed in the state S with respect to the set of bindings B, the body of method m should be executed in the state S on each object O and on each parameter evaluation corresponding to a substitution in B. Note that each of these body executions transform the database from one state to another. Therefore, when different method implementations must be executed for different sets of objects, the database transformation into a new state is not atomic. Each implementation may perform completely different updates on the database. In Section 3.4 we discuss which conditions must be imposed to ensure absence of conflicts.

Generation of the Set of Bindings in which the Rule Conditions are Evaluated Let m be a method name, $B = \{\theta_1, ..., \theta_m\}$ be a set of bindings and S be a database state. We define the semantics of $O.m(t_1, \ldots, t_n)$ where $t_1, \ldots, t_j$ are the actual input parameters and $t_{j+1}, \ldots, t_n$ are the actual output parameters. Since we assume that the database schema meets all the static constraints, the formal parameters of method m are $P_1, ..., P_n$, and a correspondence between the actual parameters $t_1, \ldots, t_n$, and the formal parameters $P_1, ..., P_n$ must be established. For each substitution θ_i we generate a substitution, $\theta_{i_{/P}}$, local to the method. A local substitution establishes the correspondence between each formal input parameter P_i, $i = 1, \ldots, j$, and the relative actual input parameter t_i. No binding is established for the output parameters, since the substitutions obtained from the

[9]Note that one of the most specific implementations is selected for each object, that is, if $d(\theta_i(O), m) = c_j$ then $\theta_i(O) \in S.\pi(c_j)$.

evaluation of the conditions implementing the method establish such bindings. We recall that the variable `Self` is implicitly included in the input parameters and, locally to the method, it is bound to the objects corresponding to variable O. The set of bindings local to the method is, thus, obtained as specified by the following definition.

Definition 3.24 *(Local Substitution). Let $O.m(t_1, \ldots, t_n)$ be a method invocation. Let $P_1, \ldots, P_j$ be the formal input parameters of method m and $P = \{P_1, \ldots, P_j\}$. Let θ_i be a substitution, the substitution local to m $\theta_{i_{/P}}$ is:*

$\theta_{i_{/P}} = \{\langle P_1/\mathcal{E}\,[\![\, t_1 \,]\!]\, \theta_i S, \ldots, P_j/\mathcal{E}\,[\![\, t_j \,]\!]\, \theta_i S, \mathtt{Self}/\mathcal{E}\,[\![\, O \,]\!]\, \theta_i S\rangle\}.$ □

Given a set of bindings $B = \{\theta_1, ..., \theta_m\}$, $B_P = \{\theta_{i_{/P}} \mid i = 1, \ldots, m\}$.

Evaluation of Method Implementation: Side-Effect Free Methods The semantics of a side-effect free method invocation, which is an atomic formula, given a database state S and a set bindings $B = \{\theta_1, ..., \theta_m\}$, is defined as follows.

Definition 3.25 *(Semantics of Side-Effect Free Method Calls). Let $O.m(t_1, \ldots, t_n)$ be a side-effect free method invocation, let S be a database state and $B = \{\theta_1, ..., \theta_m\}$ be a set of bindings. Then:*

$$\begin{aligned}&\mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, BS = \mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, B_{/P} S = \\ &\mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, \theta_{1_{/P}} S \cup \ldots \cup \mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, \theta_{m_{/P}} S = B'_{/P}\end{aligned}$$

where $\mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, \theta_{i_{/P}} S$, for $i = 1, \ldots, m$ is defined as follows.

Let $d(\mathcal{E}\,[\![\, O_{S_0} \,]\!]\, \theta_{i_{/P}} S,\ m)$=c and let the implementation for method m in class c be $m(P_1, \ldots, P_n) : condition_1 || \ldots || condition_k$. Then:

$$\mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, \theta_{i_{/P}} S = (\mathcal{C}\,[\![\, (condition_1)_{S_0} \,]\!]\, \theta_{i_{/P}} S) \cup \ldots \cup (\mathcal{C}\,[\![\, (condition_k)_{S_0} \,]\!]\, \theta_{i_{/P}} S)$$

□

Note that, because of the definition of condition evaluation, the above set of substitutions $\mathcal{C}\,[\![\, O.m(t_1, \ldots, t_n)_{S_0} \,]\!]\, \theta_{i_{/P}} S$ is the union of the sets of substitutions each one making true one of the conditions.

Evaluation of Method Implementation: Methods with Side-Effects The semantics of a method invocation, for methods with side-effects is defined as follows.

Definition 3.26 *(Semantics of Method Calls with Side-Effects). Let $O.m(t_1, \ldots, t_n)$ be a method invocation with side-effects, let S be a database state and B be a set of bindings. Let $d(O_B, m) = c$*[10] *and let the implementation for method m in class c be $m(P_1, \ldots, P_n) : condition_1 \rightarrow u^1_1; \ldots; u^1_n || \ldots || condition_m \rightarrow u^m_1; \ldots; u^m_k$. Then:*

$$\mathcal{U} [\![O.m(t_1, \ldots, t_n)_{S_0}]\!] BS = \mathcal{U} [\![O.m(t_1, \ldots, t_n)_{S_0}]\!] B_{/P}S =$$
$$\mathcal{U} [\![(condition_1 \rightarrow u^1_1; \ldots; u^1_n || \ldots || condition_m \rightarrow u^m_1; \ldots; u^m_k)_{S_0})]\!] B_P S = \langle B'_{/P}, S' \rangle$$

where $B'_{/P} = \bigcup_{i=1}^m (\pi_1(\mathcal{U} [\![(u^i_1; \ldots; u^i_h)_{S_0}]\!] B_{i_{/P}} \setminus (B_{1_{/P}} \cup \ldots \cup B_{i-1_{/P}})S) = \bigcup_{i=1}^m (\pi_1 \langle B^i_f, S^i_f \rangle)$

and $S' = S_m$ is defined by the following recurrence

$$\begin{cases} S_0 = S \\ S_i = \pi_2(\mathcal{U} [\![(u^i_1; \ldots; u^i_h)_{S_0}]\!] B_{i_{/P}} \setminus (B_{1_{/P}} \cup \ldots \cup B_{i-1_{/P}})S_{i-1}) \end{cases}$$

where π_i, $i = 1, 2$, denotes the projection of the i-th component of a pair $\langle B, S \rangle$. □

As we stated earlier, the evaluation of the condition for each rule results into a set of bindings. For the i-th condition, given B_P as defined in Definition 3.24, we have that $\mathcal{C} [\![(condition_i)_{S_0}]\!] B_P S = B_{i_{/P}}$. Starting from this set of bindings and from the state S, the update sequence of the body of the i-th rule is evaluated in $\mathcal{U} [\![(u^i_1; \ldots; u^i_n)_{S_0}]\!] B_{i_{/P}} \setminus (B_{1_{/P}} \cup \ldots \cup B_{i-1_{/P}})S$. Note that the set of bindings in which the update sequence is evaluated is not $B_{i_{/P}}$, rather it is $B_{i_{/P}} \setminus (B_{1_{/P}} \cup \ldots \cup B_{i-1_{/P}})$. One of the main issues in giving the semantics of a method implemented by several rules, is that the sets of bindings returned by the condition evaluations are not, in general, disjoint. Intuitively we want that a method performs different operations corresponding to sets of different objects. If, in the same method, two substitutions return the same set of bindings B, on such set the operations related to the first condition are executed first, followed by ones related to the second condition. Therefore, it is as if the concatenation of the update sequences had been executed on a single set of bindings.

We could impose that the set of bindings resulting from the evaluation of the conditions of different rules be disjoint, that is, $B_1 \cap \ldots \cap B_m = \emptyset$[11], and return an error whenever such restriction is not fulfilled. In our opinion such restriction is too strict. Suppose that

[10] For the sake of simplicity we consider the case in which this class is unique. The extension to more than one class will be presented in Section 3.4.

[11] We omit the restriction to P in this context, because it is irrelevant and makes the notation heavier.

two B_i's have in common a substitution $\bar{\theta}$. Because of such a substitution, the method call would fail, even when all the others substitutions would raise no such error. Thus, we have choosen to establish a sort of implicit order among the rules in the implementation of a method. If a substitution $\bar{\theta}$ is shared between two B_i's, only updates related to the *first* rule, whose condition evaluation includes $\bar{\theta}$, are performed on such substitution. This is the reason why the evaluation of the body of the i-th rule takes place in $B_i \setminus (B_1 \cup \ldots \cup B_{i-1})$, that is, all B_i's from which all the substitutions already computed in the previous rules are eliminated, if they are present.

This semantics does not prevent the same object to be modified by two different rules. As an example, θ' and θ'' belonging to two different B_i's, such that, for a variable X, $\theta'(X) = \theta''(X)$ holds. A correct handling of those cases is left to the programmer who must ensure that the variable X is used only as input parameter or that, if subsequent changes are performed, they must be coherent.

The first assumption we have made in dealing with the issues related to B_i, is that, intuitively, we want that a method performs different updates for different sets of objects. In a certain sense, we require rules to be mutually exclusive. Since it is difficult to establish whether the rules in a method always modify disjoint sets of objects for every possible database state, our semantics solves this problem by making the B_i's disjoint, if they are not, using the implicit order of the rules. If the B_i's are already disjoint, mutual exclusion is automatic.

$B'_{/_P}$ is the union of all B''^{i}_{f} resulting from the evaluation of the different rules. The substitutions in $B'_{/_P}$ obtained as result are bound to the local set of bindings restricted to the formal parameters. Therefore, this set of bindings must be joined to the set B of bindings on which the method has been invoked, and a correspondence must be established between the formal and the actual output parameters. We address this issue in following paragraph.

For what concerns the state, the semantics of the body of the i-th rule is computed in the state S_{i-1} resulting from the evaluation of the body of the previous rule, according to the implicit order.

Generation of the Set of Bindings to Return as Result It is necessary to distinguish two cases:

1. the method has output parameters,
2. the method has no output parameters.

If the method has output parameters, before defining the set B', we have to establish the link between the formal output parameters P_x $(x = j+1, \ldots, n)$ and the actual output parameters t_x ($x = j+1, \ldots, n$). Let B_{output} be the set defined as follows. Intuitively, the set B_{output} is the set $B'_{/P}$ corresponding to the actual output parameters.

Definition 3.27 *(B_{output}). Let $B'_{/P} = \{\theta'_{1_{/P}}, \ldots, \theta'_{k_{/P}}\}$ as above, for each substitution $\theta'_{i_{/P}} \in B'_{/P}$ let $\theta_{i_{output}}$ defined as follows:*

1. *if $\theta'_{i_{/P}}(P_x) = v$ then $\theta_{i_{output}}(t_x) = v \quad x = j+1, \ldots, n$*
2. *if $\theta'_{i_{/P}}(P_x) = \perp$ then $\theta_{i_{output}}(t_x) = \texttt{null} \quad x = j+1, \ldots, n.$*

Then $B_{output} = \{\theta_{1_{output}}, \ldots, \theta_{h_{output}}\}$. □

Let $f : \{1, \ldots, k\} \to \{1, \ldots, m\}$, $m \leq k$, be a function such that, given $B = \{\theta_1, ..., \theta_m\}$ and $B'_{/P} = \{\theta'_{1_{/P}}, \ldots, \theta'_{k_{/P}}\}$ as above, $f(i) = j$ if $\theta'_{i_{/P}}(P_x) = \mathcal{E}[\![t_x]\!] \theta_j S$, where P_x is a formal input parameter and t_x is an actual input parameter.

To define B' we have to join each substitution $\theta_{i_{output}}$, which binds the actual output parameters, with the substitution $\theta_{f(i)}$, with respect to which the corresponding actual input parameters have been evaluated, as stated in the following definition.

Definition 3.28 *(Resulting Set of Bindings for Method with Output Parameters). Let f be the function defined above, then:*

$$B' = \bigcup_{i=1}^{m} \{\theta_{f(i)} \cup \theta_{i_{output}}\}$$

where $\theta_{f(i)} \in B$ and $\theta_{i_{output}} \in B_{output}$. □

In the second case, that is, if the method has no output parameters, we have to distinguish two subcases: the method is side-effect free; the method has side-effects. B' is defined as follows.

Definition 3.29 *(Resulting Set of Bindings for Method without Output Parameters). Let f be the previously defined function, then:*

$$B' = \begin{cases} \bigcup_{j=1}^{m} \{\theta_j \in B \ s.t. \ \exists \ i, \ 1 \leq i \leq k, \ f(i) = j\} & \textit{for side-effect free methods} \\ B & \textit{otherwise} \end{cases}$$

□

Intuitively, methods implementing boolean predicates, have no output parameters. Therefore, for each substitution $\theta_i \in B$ they compute the relative $\theta_{i_{/P}}$. If such a substitution makes true one of the conditions implementing the method, then the corresponding substitution $\theta_i \in B$ is given as result in B'.

We notice that the above semantics formalizes for side-effect free methods the cases 1 and 2 introduced at the beginning of the section. In case 1, the returned bindings set is completed by including the output values of the method corresponding to each substitution of input parameters which satisfies the conditions. In case 2, the returned bindings consists of the substitutions in B verifying one of the conditions. If the method has side-effects but no output parameters, it means that the method aims at updating the database, thus the database state changes whereas the bindings set remains invariant, that is, $B' = B$.

3.4 Properties

In this section we show how some relevant semantic properties of methods can be reasoned about in our language. We consider behavioral refinement of methods in subclasses and conflicts among method implementations in sibling classes. Both these properties can be formally characterized in terms of the semantics of our language. Note that these properties depend on the specific database state, and are thus undecidable in general. We have however devised some sufficient static conditions ensuring the verification of those properties.

3.4.1 Behavioral Refinement

In object-oriented languages, polymorphic code can be written. However, this makes reasoning about methods difficult. There might be several method implementations that could be executed upon a method invocation. In most object-oriented programming languages, moreover, method overriding is always uncontrolled. The only restrictions imposed on method redefinitions in subclasses concern method signatures, but no restriction is imposed on method implementations to ensure that semantics is preserved. Intuitively, if a class c' is a subclass of a class c, every object instance of c' must behave *like* some object of class c. This notion of semantic method refinement is known as *behavioral subtyping* [123]. Consider as an example an *init* method initializing the attributes of an object. The implementation of such method in each class initializes to the proper values the attributes of that class. An *init* implementation in class c' that only extends the *init* implementation of class c with a new piece of code to initialize the additional attributes of c' is an example of behavior refinement. To obtain a correct behavioral subtyping, *super* calls or the *inner* mechanism could be exploited.

A method m in a class c' (denoted by $m^{c'}$) is a behavioral refinement of method m in a class c (denoted by m^c) if the portion of state manipulated by $m^{c'}$ includes the portion of state manipulated by m^c and if the portion of state modified by both is modified in the same way[12]. To formally define this notion, we must first model the changes made by a method execution. Given a method m, let $\mu(m)$ be the set of classes manipulated by m (through any of the primitive update operations)[13]. Given a method m and a class c, let $\delta_m(c)$ be the set of objects deleted from class c and $\iota_m(c)$ the set of objects inserted in class c as a consequence of the execution of method m; moreover, given an oid i and an attribute name a, let $\nu_m(i).a$ be defined if and only if the execution of method m has modified the value of attribute a of the object identified by i, and, if defined, let it contain the new value of the attribute.

Definition 3.30 *(Portion of State Manipulated by a Method). Given a method implementation m, such that $\mu(m)$ is the set of classes manipulated by m, a database state S and a set of bindings B such that $\mathcal{U}\,[\![\, m \,]\!]\, BS = \langle B', S' \rangle$, consider the following functions:*

- $\forall\ c \in \mu(m) : \delta_m(c) = S.\pi(c) \setminus S'.\pi(c)$
- $\forall\ c \in \mu(m) : \iota_m(c) = S'.\pi(c) \setminus S.\pi(c)$
- $\forall\ c \in \mu(m),\ \forall\ i \in S'.\pi(c) : \nu_m(i).a = v$ *if and only if the execution of method m has modified the value of the attribute a of object i setting it equal to v.*

The triple $\langle \delta_m, \iota_m, \nu_m \rangle$ models the portion of state manipulated by the method invocation. □

Definition 3.31 *(Behavioral Refinement). Given a class c' subclass of a class c, method $m^{c'}$ is a behavioral refinement of method m^c if $\mu(m^c) \subseteq \mu(m^{c'})$ and if, for each database state, and for each set of bindings, the execution of m^c and $m^{c'}$ restricted to the objects in c' satisfies the following conditions:*

- $\forall \bar{c} \in \mu(m^c)$*:* $\delta_{m^c}(\bar{c}) \subseteq \delta_{m^{c'}}(\bar{c})$ *and* $\iota_{m^c}(\bar{c}) \subseteq \iota_{m^{c'}}(\bar{c})$*;*
- $\forall \bar{c} \in \mu(m^c), \forall a$ *attribute of* $\bar{c}$*,* $\forall$ *oid* i *instance of* $\bar{c}$*: if* $\nu_{m^c}(i).a$ *is defined, then* $\nu_{m^{c'}}(i).a$ *is defined and* $\nu_{m^c}(i).a = \nu_{m^{c'}}(i).a$*.* □

[12] Note that a method invoked on an object (set of objects) instance(s) of a class may manipulate objects of other classes.

[13] They can be deduced syntactically.

Unfortunately, given two method definitions, behavioral refinement is undecidable.

We have however devised some sufficient static conditions ensuring that a method $m^{c'}$ is a behavioral refinement of a method m^c. These conditions can be checked at method definition time, so that the overriding of a method in a subclass can be disallowed if the overriding method is not a behavioral refinement of the overridden one. These conditions (referred to as *refinement conditions*) requires that for each rule R in m^c a rule R' in $m^{c'}$ must exist refining R[14]. The implementation for $m^{c'}$ may also contain additional rules. An update rule R' refines an update rule R if the condition of R' subsumes the condition of R, restricted to the class of R', and the action of R' refines the action of R. Containment among conditions can be tested by exploiting algorithms for deciding subsumption among object-oriented queries [50, 69][15]. Algorithms for testing containment among conditions expressed in our language have been developed [31]. Action refinement basically means that the action of the refined rule must do at least what the action of the original rule does. This means that, for each update in the sequence constituting the action of the original rule there must be a corresponding update in the refined action. However, since rule actions are update sequences, the corresponding update could be discarded by some complementary update executed after it in the sequence. Consider as an example the case of a rule creating an object in its action, overridden by a rule whose action firstly creates a corresponding object and then deletes it. We consider, therefore, the net effect of the updates in the sequence and we state that for each update in the original rule there must be a corresponding update in the net effect of the refined rule. Note that the notion of net effect employed here is purely syntactical and relies only on complementary database operations. Net effect computation consists of composing the effects of those actions whose effect was compensated by a subsequent action on the same object.

In what follows, we formalize these notions. The bindings produced by the evaluation of the condition are represented as a set of ground substitutions, as defined in Definition 3.10. A substitution ρ is a *renaming* of variables if $\rho = \{X_1/t_1, \ldots, X_n/t_n\}$, and, for each $i, 1 \leq i \leq n$, t_i is a distinct variable. Moreover, we introduce the following notations. Given an update rule R, let $BVar(R)$ denote the set of variables appearing in the condition part of R and in the action part of R (that is, the variables employed for passing bindings). Moreover, given a renaming of variables ρ, let -^{ρ} denote subset (non strict) inclusion modulo renaming ρ. That is, let B and B' be sets of substitutions (sets of bindings) or set of variables, $B \;\text{-}^{\rho}\; B'$ means $B \subseteq B'\rho$; let $\mathtt{w}^{\rho}$ denote equality modulo renaming ρ, $B \;\mathtt{w}^{\rho}\; B'$ means $B = B'\rho$. Equality modulo renaming ρ also holds between terms. Let t and t' be terms, $t \;\mathtt{w}^{\rho}\; t'$ means $t = t'\rho$ where $t'\rho$ is the term t' where each variable has been substituted

[14] Actually, a rule could also be refined by two or more rules, the disjunction of whose conditions must subsume the condition of the original rule. However, we do not consider this case here.

[15] Actually, containment must be tested modulo variable renaming, since the methods could exploit different parameter names and different local variables. Moreover, the condition of the refined rule may also share some more variables with its actions.

according to renaming ρ. Given a formula F and two class identifiers c_1 and c_2, let $F[c_1/c_2]$ denote the formula obtained from F by substituting each class formula $c_1(X)$ with a class formula $c_2(X)$. Finally, given a record term t and a class c, let $t_{|c}$ denote the restriction of the term to the field whose labels are attributes of c.

Definition 3.32 *(Basic Update Refinement). An atomic update u is a refinement of an atomic update u' under the renaming of variables ρ (denoted as $u \leq_b^\rho u'$) if and only if one of the following conditions holds:*

- $u = \texttt{create}(c, t, O)$, $u' = \texttt{create}(c', t', O')$ *and* $c \leq_T c'$, $O\ \mathtt{w}^\rho\ O'$, $t_{|c'}\ \mathtt{w}^\rho\ t'$;
- $u = \texttt{delete}(c, O)$, $u' = \texttt{delete}(c', O')$ *and* $c \leq_T c'$, $O\ \mathtt{w}^\rho\ O'$;
- $u = \texttt{generalize}(c, c_1, O)$, $u' = \texttt{generalize}(c', c'_1, O')$ *and* $c \leq_T c'$, $c'_1 \leq_T c_1$, $O\ \mathtt{w}^\rho\ O'$;
- $u = \texttt{specialize}(c, c_1, O, t)$, $u' = \texttt{specialize}(c', c'_1, O', t')$ *and* $c \leq_T c'$, $c_1 \leq_T c'_1$, $O\ \mathtt{w}^\rho\ O'$, $t_{|c'_1}\ \mathtt{w}^\rho\ t'$;
- $u = \texttt{modify}(c.a, O, t)$, $u' = \texttt{modify}(c'.a, O', t')$ *and* $c \leq_T c'$, $t\ \mathtt{w}^\rho\ t'$, $O\ \mathtt{w}^\rho\ O'$;
- $u = O.m(t_1, \ldots, t_n)$, $u' = O'.m(t'_1, \ldots, t'_n)$ *and for each* $i, 1 \leq i \leq n$, $t_i\ \mathtt{w}^\rho\ t'_i$, $O\ \mathtt{w}^\rho\ O'$.

□

Net effect computation consists of composing the effects of those actions whose effect was compensated by a subsequent action on the same object. Classical compensations [75, 172] are performed as follows:

- a sequence of `create` and `delete` primitives on the same object, possibly with an arbitrary number of intermediate `modify` primitives on that object, has a null net effect;
- a sequence of `create` and several `modify` primitives on the same object has the net effect of a single create operation;
- a sequence of several `modify` and a `delete` primitive on the same object has the net effect of a single delete operation on that object;
- a sequence of several `modify` primitives on the same object has the net effect of a single modify operation on the old object which modifies it in the newest.

In addition to those classical compensations, we consider also compensations involving object migrations along the hierarchy. For the sake of brevity, we omit all rules for computing the net effect of a sequence of actions. Given an action list A, let $Net(A)$ denote the net effect of the sequence. The net effect of the sequence is performed at a syntactic level, by considering compensating actions on the same object-denoting term, contained in the sequence.

Definition 3.33 *(Action Refinement). An action list A is a refinement of an action list A' (denoted as $A \leq_a^\rho A'$) if the following conditions hold:*

- $Net(A') = u'_1; \ldots; u'_n$, $Net(A) = u_1; \ldots; u_m$ *and* $m \geq n$*;*
- *for each* u'_i, $1 \leq i \leq n$, *in* $Net(A')$, u_j, $1 \leq j \leq m$, *in* $Net(A)$ *exists, such that* $u_j \leq_b^\rho u'_i$, *that is,* u_j *is a refinement of* u'_i *according to Definition 3.32; let function* $\xi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$, *such that* $\xi(i) = j$, *model this correspondence;*
- *if* u'_i *precedes* u'_k *in* $Net(A')$ $u_{\xi(i)}$ *precedes* $u_{\xi(k)}$ *in* $Net(A)$. □

We remark that, since both basic action refinement and the computation of net effect only rely on syntactical properties of the action list, action refinement is decidable.

Example 3.16 *Suppose* `employee_log` $\leq_T$ `person_log` *and* `employee` $\leq_T$ `person`*, then*

- $A =$ `create(employee_log, (who:X, age:X.age, salary:X.salary), 0)`

  is a refinement of

  $A' =$ `create(person_log, (who:X, age:X.age), 0)`

  under renaming ϵ*, and*

- $A =$ `modify(department.nbr_of_employees, Y, Y.nbr_of_employees-1);`
  `delete(employee,X)`

  is a refinement of

  $A' =$ `delete(person,X)`

  under renaming ϵ*.* ◇

We are now able to give the definition of update rule refinement.

Definition 3.34 *(Update Rule Refinement). An update rule $R = C \rightarrow A$, belonging to the implementation of a class c, is a refinement of an update rule $R' = C' \rightarrow A'$, belonging to the implementation of a superclass c' of c, if ρ, renaming of variables, exists such that the following conditions are satisfied:*

- *$BVar(R') \dashv^{\rho} BVar(R)$, and*
- *$\forall S$ database state, $\forall B$ set of bindings,*
  $\mathcal{C} [\![R'[c'/c]]\!] SB \dashv^{\rho} (\mathcal{C} [\![R]\!] SB)_{|BVar(R')}$
- *$A \leq^{\rho}_{a} A'$, that is, A is a refinement of A' according to Definition 3.33.* □

Finally, we can state when a method implementation refines a superclass method implementation.

Definition 3.35 *(Syntactic Refinement). Given the definition of method $m^{c'}$, consisting of the rules $R'_1, \ldots, R'_n$, and the definition of method m^c, consisting of the rules $R_1, \ldots, R_k$, $m^{c'}$ is a syntactic refinement of method m^c if for each i, $1 \leq i \leq k$, j exists, $1 \leq j \leq n$, such that, R'_j is a refinement of R_i, according to Definition 3.34.* □

The following result holds.

Proposition 3.1 *Given two methods m^c and $m^{c'}$:*

- *we can decide whether $m^{c'}$ is a syntactic refinement of method m^c;*
- *if $m^{c'}$ is a syntactic refinement of method m^c, then $m^{c'}$ is a behavioral refinement of m^c.* □

Proof Sketch: The proof of the first item is based on the fact that both condition subsumption and action refinement are decidable.

The proof of the second item is more complex. $\mu(m^c) \subseteq \mu(m^{c'})$ follows from the fact that for each update rule R_i in m^c implementation a rule R'_j exists in $m^{c'}$ implementation, such that R'_j refines R_i. Since this means that for any upate u^* in the net effect of R'_j action a corresponding update u^{**} in the net effect of R_i action exists, $\mu(m^c) \subseteq \mu(m^{c'})$.

Let us consider now the first condition of Definition 3.31, that is $\forall \bar{c} \in \mu(m^c)$: $\delta_{m^c}(\bar{c}) \subseteq \delta_{m^{c'}}(\bar{c})$. Again, this follows from the fact that for each update rule R_i in m^c implementation a rule R'_j exists in $m^{c'}$ implementation, such that R'_j refines R_i. This means that the set of objects selected by R'_j condition contains the set of objects selected by R_i condition (restricted to instances of c'), and for each delete operation in the net effect of R'_j action, a corresponding deletion is in the net effect of R_i action.

The same arguments can be applied for the other conditions of Definition 3.31, namely those concerned with insertions and modifications. □

3.4.2 Conflicting Method Definitions

In our model, objects can belong to several most specific classes. In Chapter 6 we will address the problem of dispatching in object systems supporting multiple class direct membership. The approach is based on selecting the method implementation from *one* of the most specific classes of the object. This approach is, however, not always adequate. When different implementations for a method are defined in the most specific classes of an object, there are cases in which the most intuitive approach is executing all of them. Consider the *init* method above, initializing the attributes of an object. Suppose that different classes provide different implementations for the *init* method. Remember that the implementation of the *init* method in each class initializes to the proper values the attributes of that class. If, when the *init* method is invoked on an object, all the *init* implementations of all the classes of which the object is a direct member, are executed, the desired result is obtained, that is, all the attributes of the object are properly initialized. Problems due to different assignments to the same attribute may, however, arise. This approach causes the problem that some attributes can be assigned different values by different implementations of the method. To solve these issues we introduce the notion of non-conflicting method implementations. Different method implementations are non-conflicting if each method modifies a portion of data distinct from the one modified by the others and if data modified by two methods is modified consistently.

Definition 3.36 *(Non-Conflicting Method Implementations). Let m_1 and m_2 be method implementations. m_1 and m_2 are said to be non-conflicting if for each state S and each set of bindings B, the following conditions are satisfied:*

- $\forall\ c \in \mu(m_1) \cap \mu(m_2)$:
  - $\forall i \in \delta_{m_1}(c) \cup \delta_{m_2}(c) :\ i \in \delta_{m_1}(c) \cap \delta_{m_2}(c)$;
  - $\forall i \in \iota_{m_1}(c) \cup \iota_{m_2}(c) :\ i \in \iota_{m_1}(c) \cap \iota_{m_2}(c)$;
- $\forall\ c \in \mu(m_1) \cap \mu(m_2)$, $\forall$ *a attribute of c*, $\forall$ *i instance of c: if* $\nu_{m_1}(i).a \neq \perp$ *and* $\nu_{m_1}(i).a \neq \perp$ *then* $\nu_{m_1}(i).a = \nu_{m_2}(i).a$. □

The notion of non-conflicting method definitions is undecidable. However some static conditions ensuring the absence of conflicts among two method implementations can be devised. A first static condition ensuring that two method implementations are non-conflicting is that the actions of the rules of the two method implementations do not contain updates on the same attributes or on the same classes. This condition can be checked through a simple syntactic analysis of the actions of the rules of which the method implementation consists[16]. Though this condition ensures that two method implementations are non-conflicting, it is too restrictive. Indeed, also methods modifying the same attributes of different objects are non-conflicting. Some sufficient conditions ensuring that two rule conditions are *exclusive*, that is, they will never be satisfied by the same objects, can be developed. Then, only the actions of rules with non-exclusive conditions must be analyzed for the presence of conflicts. For these rules, first the net effect of those actions is computed and the updates corresponding to a common more general action (if any) are extracted. Then, if the remaining updates modify different portions of the schema, that is, different attributes or classes, the methods are non-conflicting. This means that a *syntactic* notion of non-conflicting method definitions can be devised, which is decidable and is a sufficient condition for (semantic) absence of conflicts among method executions.

Given two non-conflicting method implementations m_1 and m_2, moreover, we can execute the two method implementation "in parallel", on the same database state S, obtaining two different final states S'_1 and S'_2, and then we can recombine (that is, *merge*) those states so that the effect of both the method executions are taken into account. Since the two method implementations are non-conflicties, their resulting states can be unambigously merged.

Definition 3.37 *(Function merge). Let m_1 and m_2 be two non-conflicting method implementations. The state $S' = merge(S'_1, S'_2)$ obtained from the merging of states S'_1 and S'_2, obtained from the executions of m_1 and m_2 in the database state S with set of bindings B, is defined as follows:*

$$\forall c \in \mathcal{CI} : S'.\pi(c) = S.\pi(c) \setminus (\delta_{m_1}(c) \cup \delta_{m_2}(c)) \bigcup (\iota_{m_1}(c) \cup \iota_{m_2}(c))$$

$$\forall i \in \mathcal{OI}, \forall a \in \mathcal{AN} : S'.\nu(i).a = \begin{cases} S.\nu(i).a & \textit{if } \nu_{m_1}(i).a = \perp \textit{ and } \nu_{m_2}(i).a = \perp \\ S'_j.\nu(i).a & \textit{if } \nu_{m_j}(i).a \neq \perp, j \in \{1, 2\} \end{cases}$$

□

Function *merge* is easily generalized to an arbitrary number of states by saying $merge(S_1, S_2, S_3, \ldots, S_n) = merge(\ldots merge(merge(S_1, S_2), S_3), \ldots S_n)$.

To conclude this section, we are now able to extend the semantics of method calls with side-effects (given in Definition 3.26) to the case in which the method implementations selected for oids in O_B are different, say $d(O_B, m) = \{c_1, \ldots, c_k\}$. Then,

[16] This kind of analysis brings some similarities with the one proposed by [27] for detecting whether a method may potentially violate an integrity constraint.

$$\mathcal{U}\,[\![\, O.m(t_1, \ldots, t_n)_{S_i} \,]\!]\, BS = \langle B', S' \rangle$$

where two cases can be distinguished:

1. $\langle B', S' \rangle = \langle \emptyset, S_i \rangle$
   if m_1, m_2 exist in $\{m^{c_1}, \ldots, m^{c_k}\}$ such that m_1 and m_2 are conflicting,

2. $B' = \bigcup_{j=1}^{k} \Pi_1(\mathcal{U}\,[\![\, O.m(t_1, \ldots, t_n)_{S_i} \,]\!]\, B_j S)$
   $S' = merge_{j=1}^{k} \Pi_2(\mathcal{U}\,[\![\, O.m(t_1, \ldots, t_n)_{S_i} \,]\!]\, B_j S)$

where, given a n-tuple $\langle el_1, \ldots, el_n \rangle$, Π_i, $1 \leq i \leq n$, denotes the projection on the i-th component el_i of the tuple.

3.5 Concluding Remarks

In this chapter we have defined a set-oriented method definition language for object databases. As we have pointed out in the introduction, the use of an ad-hoc set-oriented language has several advantages compared to the use of general purpose imperative programming languages. By contrast, one could argue that the use of a general purpose imperative programming language is the best approach to overcome the impedance mismatch. Though our language is not computationally complete (because the language is restricted to non-recursive method definitions), it is expressive enough to express sequences of statements and if-then-else control constructs. Moreover, we believe that our framework could be easily extended to accomodate recursive method definitions. Note, moreover, that the object-oriented approach as opposed to other traditional approaches (such as embedding SQL into a programming language) improves the problem of impedance mismatch. Under the object-oriented approach, application programs in most cases can simply invoke methods, which encapsulate update commands. Methods are written through rules and need not to be interfaced with other languages.

Other approaches dealing with "how" to perform update sequences on a database have been proposed. In the following we compare our approach with two other approaches. We consider an update sequence of the form u_1;...;u_n to be performed on a set of variable bindings B, a set of objects, in a particular database state, where each u_i is an update operation (either atomic or a method call).

The first approach we consider is the deductive database approach [37] where the update sequence is considered as a set of updates and each single operation is executed according to the set-oriented semantics, that is, on the set of bindings B. Because updates u_1;...;u_n are considered as a set, the execution order must be irrelevant. This approach thus needs a notion of consistent updates. Consistency is important because it prevents a set of updates containing both an insertion and a deletion of the same object to be executed. One disadvantage of this approach is that not all syntactically correct update sequences are semantically correct. Moreover the sharp distinction between the phase in which bindings are collected and the update phase is not convenient for our language in which updates can be method calls.

The second approach is that of object-oriented update sequences [122]. In this approach u_1;...;u_n is considered as a sequence, in the real meaning of the word (one update after the other). Such sequence must be executed, in the established order, on each single object in B. In this case the order in which the objects in B are chosen to perform the updates must be irrelevant, otherwise the semantics is non-deterministic. To make the semantics deterministic, only update sequences which commute on objects in B are accepted as valid. Like the deductive approach, not all syntactically correct update sequences are semantically correct. This semantics eliminate the sequences which cause problems, by considering them invalid, without solving the problem of non-determinism. Moreover, several problems concerning aggregate functions and data sharing arise [35].

Our approach combines the principles of the previous approaches. The updates u_1;...;u_n are seen as a sequence; each update is performed on the set of bindings B. Therefore, each u_i is performed as in the set-oriented approach. Our approach seems preferable to the other ones because it better models the intuitive semantics. Moreover it does not impose conditions restricting the set of valid sequences, such as consistency and commutativity, but our semantics detects at execution time programming errors. With our approach problems related to aggregate functions do not arise and the semantics is deterministic.

The following example shows how the three approaches differ and how our approach solves problems arising with the others.

Example 3.17 *Consider a class* `employee` *containing two attributes:* `salary` *(with domain* `integer`*) and* `manager` *(with domain* `employee`*), representing the salary and the manager of an employee, respectively. Consider a method* `raise_salary` *of class* `employee` *that, when invoked on an employee sets its salary to the salary of his manager and increases the salary of his manager of an amount specified as input. Thus, signature of method* `raise_salary` *is* `employee` $\times$ `integer` $\rightarrow$*, while its implementation is specified by the following rule:*

```
raise_salary(Amount):M = Self.manager →
                     modify(employee.salary, Self, M.salary);
                     modify(employee.salary, M, M.salary + Amount)
```

Consider the invocation of method `raise_salary` *with respect to the following set of bindings* $B = \{\theta_1, \theta_2\}$ *with* $\theta_1 = \{\texttt{Self}/o_1, \texttt{Amount}/100\}$ *and* $\theta_2 = \{\texttt{Self}/o_2, \texttt{Amount}/100\}$*, and* *a database state S such that $S.\nu(o_1)$.*`manager` $= o_2$ *and $S.\nu(o_2)$.*`manager` $= o_2$ *(that is, o_2 is manager of himself).*

According to the deductive approach such a method invocation fails. Indeed, considering object o_2, the first `modify` *operation sets the salary of o_2 to his salary in S, while the second* `modify` *operation sets the salary of o_2 to his salary in S increased by 100. These two modifications are collected in a set and since they are not consistent (different values are assigned to the same attribute) the method invocation fails.*

According to the object-oriented update sequence approach, the semantics is non-deterministic, indeed, different database states are obtained depending on the order in which the two substitutions are considered. If substitution θ_1 is considered first, the salary of o_1 is set to the salary of o_2 in S, whereas the salary of o_2 is set to his salary in S increased by 200. By contrast, if substitution θ_2 is considered first, the salary of o_1 is set to the salary of o_2 in S increased by 100, whereas the salary of o_2 is set to his salary in S increased by 200. Therefore, o_1 could receive different salaries depending on the order in which the substitutions are considered. The semantics is thus non-deterministics since it depends on that order.

With our approach, by contrast, no problems arise and the salary of o_1 is set to the salary of o_2 in S, whereas the salary of o_2 is set to his salary in S increased by 100. $\diamond$

Chapter 4

Triggers

In this chapter we deal with the active component of our model. In our opinion, though several systems supporting active rules in an object-oriented data model have been proposed, several issues related to the integration of triggers with object-oriented modeling concepts have not been satisfactorily addressed and still need investigation. In this chapter, we extensively analyze issues related to trigger inheritance and refinement in the context of our active object-oriented data model. In particular, we define a semantics for an active object language that takes into account trigger inheritance and supports trigger overriding. We also state conditions on trigger overriding ensuring that trigger semantics is preserved in subclasses [36]. There are several cases in which rule overriding is useful. However, we believe that rules should not be overridden in a uncontrolled way, rather they should only be *refined* in subclasses, that is, they could be overridden provided that (*i*) the redefined rule is triggered each time the overridden one would be; (*ii*) the redefined rule does at least what the overridden one would do.

This chapter is organized as follows. Section 4.1 motivates the need for trigger overriding. Section 4.2 presents the reference active rule language. In Section 4.3 trigger semantics is formalized. Section 4.4 deals with trigger redefinition. In Section 4.5 we compare our approach with other existing approaches to trigger overriding and to the definition of a semantic framework for triggers. Finally, Section 4.6 concludes the chapter.

4.1 Trigger Overriding

The influence of inheritance on triggers has not been deeply investigated in existing object-oriented database systems. Under some proposals [26, 75, 132], triggers are always inherited and can never be overridden nor refined. Such an approach, that we refer to as *full trigger* *inheritance*, simply means that event types are propagated across the class inheritance hierarchy. Consider the event of a rule r, say $op(c')$, characterizing the operation op on a class c'. If c' has a subclass c, when an operation op occurs on an instance of c rule r is triggered, as well as any other rule having as event $op(c)$. Thus, inheritance of triggers is accomplished by applying a trigger to all the members of the class in which the trigger is defined, rather than only to the instances of this class.

Full trigger inheritance is, however, not appropriate in all situations, as shown by the examples below. There are situations in which trigger overriding is needed. Moreover the meaning of the ISA hierarchy is to define a class in terms of another class possibly refining its attributes, methods and triggers. This way of reasoning is one of the aspects which have made of the object-oriented approach a powerful paradigm. Thus, in our opinion, the possibility of redefining triggers in subclasses, instead of simply inheriting them, is needed in an active object-oriented system.

In a system supporting trigger inheritance, but not trigger overriding[1], as in the full inheritance case, the only way to refine the behavior of a trigger in a subclass is to define in the subclass a trigger on the same events which performs the refined action. However, for this addition to be effective, the trigger in the superclass must have priority over the trigger in the subclass. Thus, upon occurrence of the common triggering event on an object belonging to the subclass, both triggers are activated, but, since the trigger defined in the superclass is executed first, the action in the trigger defined in the subclass "prevails". However, it is not always possible to refine the behavior of a trigger in a subclass by adding a new trigger, even by specifying that the subclass trigger has lower priority than (thus, is executed after) the superclass one. Consider the following examples.

Example 4.1 *Suppose that a class* `employee` *and a subclass* `manager` *are defined. In the former class an attribute* `rank` *is defined. Consider a trigger r_1 increasing the salary of an employee each time the rank of that employee is increased. Moreover consider a corresponding trigger r_2 for the class* `manager` *increasing the salary of a manager of an amount greater than that of a simple employee. Notice that the trigger r_2 is exactly the refinement of r_1 for the class* `manager`. *If trigger overriding is not supported we have to define two different triggers as follows*[2]*:*

- *trigger r_1 on class* `employee`

Events:	`modify(rank)`
Condition:	`employee(X), occurred(modify(rank),X), Y=X.rank,` `old(Y>X.rank)`
Action:	`modify(employee.salary, X, X.salary + 50)`

[1]We remark that this is almost always the case for existing active object-oriented database systems.

[2]Note that the syntax used in the following example will be introduced in Section 4.2.

- *trigger* r_2 *on class* `manager`

  Events: `modify(rank)`
  Condition: `manager(X), occurred(modify(rank),X), Y=X.rank, old(Y>X.rank)`
  Action: `modify(manager.salary, X, X.salary + 100)`

In the previous example whenever the rank of an employee is increased, both r_1 *and* r_2 *are triggered and if* r_1 *has priority over* r_2 *(as under the default ordering we mentioned above)* r_1 *is executed first. The execution of* r_1 *increases the manager salary of* `50` *and the execution of* r_2 *increases again the salary of* `100`*, thus at the end the manager salary has been increased of* `150`*, instead of only* `100`*, which is not the desired result. This problem can be overcome by increasing the salary of a manager of the right amount in* r_2 *keeping into account that the salary has already been increased by* r_1*, but this is not intuitive and not object-oriented, it does not model a refinement. Suppose, moreover, to have a complex class hierarchy, these calculi become really complex and not always possible.* ◇

Example 4.2 *Consider a class* `person` *with a subclass* `employee`*. Suppose, moreover, that a third class* `department` *is defined, with an attribute* `nbr_of_employees` *which maintains the number of employees of the department. Suppose that a trigger* r_1 *is defined on class* `person` *such that, whenever the age of a person is greater than* 100*, an object of a class* `person_log` *whose state refers to the deleted object is created (the class* `person_log` *is used for monitoring purposes) and then the object is deleted. Suppose that a corresponding trigger* r_2 *is defined on class* `employee`*, such that, whenever the age of an employee is greater than 100, an object of a class* `employee_log` *is created, the value for attribute* `nbr_of_employees` *of the department in which the employee works is decremented and finally the employee is deleted. The triggers can be expressed as follows:*

- *trigger* r_1 *on class* `person`

  Events: `modify(age)`
  Condition: `person(X), occurred(modify(age),X), X.age > 100`
  Action: `create(person_log, (who:X, age:X.age), O); delete(person,X)`

- *trigger* r_2 *on class* `employee`

  Events: `modify(age)`
  Condition: `employee(X), occurred(modify(age),X), X.age > 100, department(Y), X.department = Y`

Action: `create(employee_log,(who:X,age:X.age,salary:X.salary),0);`
`modify(department.nbr_of_employees,Y,Y.nbr_of_employees-1);`
`delete(employee,X)`

Whenever the age of an employee is set to `101`*, both r_1 and r_2 are triggered and, if r_1 has priority over r_2 (as under the default ordering we mentioned above), r_1 is executed first. The execution of r_1 deletes the involved object, and then the execution of r_2 does not have any effect. Indeed, when r_2 is executed the object whose* `age` *attribute has been modified, has already been deleted. Therefore, it cannot be accessed any longer. As a consequence, the object does not satisfies trigger r_2 condition. Intuitively trigger r_2 is just the refinement of r_1 because it has a behavior similar to that of r_1 but refined for subclass* `employee`*. In a system supporting inheritance and trigger overriding, trigger r_2 would be the refinement of trigger r_1, thus for the objects proper instances of* `employee` *only trigger r_2 would be executed giving a correct result.* $\diamond$

As it is shown by the previous examples, the absence of trigger overriding does not allow triggers to manage in different ways the instances and the members of a class. The distinction between classes and subclasses, that is, between instances and members, is one of the main features of the object-oriented approach. Thus, we can say that a system only supporting full trigger inheritance and, in general, each system which does not allow triggers to be overridden in subclasses, is not truly an object-oriented system.

In the following we will propose an active object rule language supporting the possibility of overriding triggers in subclasses.

4.2 Active Rule Language: Syntax

In this section we introduce the syntax of the active rule language. The language supports *set-oriented* active rules [172]: rules react to sets of changes to the database and may perform sets of changes. This approach is consistent with the remainder of our model, which supports set-oriented, declarative query and update languages. In this respect, our language is different from most other active object-oriented database languages where rules are triggered by method activations, and are used to test pre and post conditions for method applications to individual objects.

Active rules in our language are called *triggers*. Each trigger is characterized by five components: a name, a class, a set of events, a condition and a reaction. The trigger name is simply a mean to refer to the trigger, and, as we will see, will be the basis on which trigger overriding is achieved. Each trigger is targeted to a class, that is, it monitors events occuring on objects of that class. In what follows we examine in details the other component of a trigger.

The set of events has a disjunction semantics, that is, the trigger becomes active if *any* of its triggering events occurs. Events are denoted by the name of the primitive operation and the schema element to which the operation is applied. Primitive operations are object creations, deletions, modifications and object migrations in the inheritance hierarchy. Modifications refer to specific attributes. In addition, active rules may monitor operation calls (methods), although rule execution remains set-oriented. As we have seen in Chapter 3, indeed, method execution itself is set-oriented.

Definition 4.1 *(Event). Let $c \in \mathcal{CI}$ be a class name, $a \in \mathcal{AN}$ be an attribute name and $m \in \mathcal{MN}$ be a method name. An event has one of the following forms:*

- `create`*;*
- `delete`*;*
- `specialize(c)`*;*
- `generalize(c)`*;*
- `modify(a)`*;*
- *m.* □

Note that all the events are intended to be applied on the class to which the corresponding trigger is targeted, and that in our language events are not parametric, that is, there is no parameter passing between the event and the condition and action parts of the trigger. When triggered, thus, a rule will be executed with no reference to the objects affected by the event that caused its triggering. As we will see soon, however, there is a syntactic mean to refer in the trigger condition to the object(s) affected by the event that caused rule triggering. This mechanism is extremely useful for efficiently testing the condition.

The condition, that is the formula monitoring the execution of the reaction part, is a conjunction of atomic formulas and it is interpreted as a predicate calculus expression over typed variables. Conditions may contain, in addition to conjunctions of atoms, *event formulas* and references to *old state*. *Event formulas* are particular formulas, built by means of the binary predicate `occurred`. This predicate is used to inspect the events that have occurred during a transaction. Syntactically, this predicate has two arguments: an event name and a variable ranging over the *oids* of the objects affected by the event, which becomes bound to *oids* of instances which are affected by the event. *References to past database states* are allowed in active rule conditions through the use of the special function `old`. Such a function applied to an atomic formula, indicates that the formula must be evaluated in the state at transaction start. This mechanism is useful, for example, for expressing comparisons among previous and current attribute values.

The atomic formulas of our language that can appear in a rule condition are then defined as follows. Note that the following definition is a slight revision of Definition 3.2, specifying the atomic formulas that can appear in constraint and method implementations.

Definition 4.2 *(Atomic Formulas). Atomic formulas are defined as follows:*

- COMPARISON FORMULAS

  if t_1, t_2 are terms and $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$ is a predefined predicate, then t_1 op t_2 is a comparison atomic formula;

- MEMBERSHIP FORMULAS

  if t_1, t_2 are terms, then t_1 in t_2 is a membership atomic formula;

  if t is a term and $c \in \mathcal{CI}$ is a class name, or $c \in \mathcal{ET}$ is an extended value type name, then t in c is a membership atomic formula;

- CLASS AND TYPE FORMULAS

  if t is a term and c is a class (or type) name, then $c(t)$ is a class formula;

- FUNCTION CALLS

  if m is the name of a side-effects free method, O is an object-denoting variable and $t_1, \ldots, t_n$ are terms, then $O.m(t_1, \ldots, t_n)$ is an atomic formula;

- EVENT FORMULAS*:*

  if X is a variable and e is an event according to Definition 4.1, then `occurred`(e, X) *is an atomic formula.* □

Atomic formulas may be applied to the state at transaction start through the *old* function. Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. Formulas that can be the condition of a trigger are then formally defined as follows. Note that the following definition is a slight revision of Definition 3.3, specifying the formulas that can appear in constraint and method implementation.

Definition 4.3 *(Formulas). Formulas are inductively defined as follows:*

- *all atomic formulas are formulas;*

- *if F is an atomic comparison or membership formula*[3], $\neg F$ *is a (complex) formula;*
- *if F is an atomic comparison or membership formula*[4], $\texttt{old}(F)$ *is a (complex) formula;*
- *if F_1 and F_2 are formulas, then F_1, F_2 is a (complex) formula.* □

As for formulas introduced in Chapter 3 we require that each formula contains exactly one class formula for each variable, specifying the type of the variable.

Example 4.3 *Examples of formulas are the following:*

- $\texttt{employee(X)}, \texttt{old(X.salary} > \texttt{40000)}$;
- $\texttt{person(X)}, \texttt{occurred(modify(salary)}, \texttt{X)}$. ◇

The reaction is a sequence of database operations, including update primitives (as specified by Definition 3.5) and the transactional command `rollback`, as specified by the following definition. Operations that constitute the reaction are executed in strict sequence, because each of them may have side effects. Condition and action of a rule may share some atomic variables, in which case the action must be executed for every binding produced by the condition on the shared variables.

Definition 4.4 *(Action). Let $c, c' \in \mathcal{CI}$ be class names, $a \in \mathcal{AN}$ be an attribute name and O be an object-denoting variable. Moreover, let $m \in \mathcal{MN}$ be an operation name and t, $t_1, \ldots, t_n$ be terms. An action has one of the following forms:*

- $\texttt{create}(c, t, O)$;
- $\texttt{delete}(c, O)$;
- $\texttt{generalize}(c, c', O)$;
- $\texttt{specialize}(c, c', O, t)$;
- $\texttt{modify}(c.a, O, t)$;
- $O.m(t_1, \ldots, t_n)$;
- `rollback`. □

[3] Class and event formulas cannot be negated.
[4] Class and event formulas cannot be tested on past database states.

We are now able to give the definition of trigger.

Definition 4.5 *(Trigger). A trigger is a 5-tuple*

$$(Name,\ Class,\ Events,\ Condition,\ Action)^5$$

where:

$Name$ *is the trigger identifier;*

$Class$ *is the class the trigger is targeted to;*

$Events$ *is the set of operations monitored by the active rule, each event in the set is as specified by Definition 4.1;*

$Condition$ *is a formula, as specified in Definition 4.3;*

$Action$ *is a sequence of actions, as specified in Definition 4.4;*

such that the following conditions are satisfied:

1. *each variable occurring as input parameter of an operation in the Action must appear in some positive atomic formulas of the Condition (safety condition);*
2. *for each event formula* $occurred(e, X)$ *in the* $Condition$*,* e *must appear in* $Events$*.* □

We remark that events are not parametric and there is no parameter passing between the event and other rule components. Thus, events only cause rules to be triggered, but rules are then considered and executed with no reference to the triggering events. However, the triggering events can be explicitly bound to variables in rule conditions by means of event formulas.

Example 4.4 *The following is an example trigger, defined on a class* `employee`*, whose effect is to prevent an employee from earning more than his manager. If an employee is assigned a salary higher than the salary of his manager, the employee salary is automatically overwritten by assigning it a salary equal to the manager salary.*

> *Events:* `create, modify(salary)`
> *Condition:* `employee(X), X.salary > X.mgr.salary`
> *Action:* `modify(employee.salary, X, X.mgr.salary)`

[5]In the following, given a trigger r, $r.Name$, $r.Class$, $r.Events$, $r.Condition$, $r.Action$ denote the respective components of trigger r.

Another example of trigger is the following, also defined on the class `employee`*, that has the effect of specializing each new employee earning more than* `40000` *by inserting the employee in the class* `specialEmpl`*.*

> *Events:* `create`
> *Condition:* `employee(X), occurred(create,X), X.salary > 40000`
> *Action:* `specialize(employee, specialEmpl, X).`

Moreover, triggers in Example 4.1 and in Example 4.2 are examples of triggers expressed in our language. $\diamond$

In order to provide a consistent behavior when multiple triggers are activated by the same events, it is important that a well-defined policy is established. In an object-oriented system, an important question is whether the triggers inherited by the superclasses of a class should have higher priority than triggers defined in the class, especially when dealing with action refinement in triggers.

A partial order $<_r$ is considered on the set of triggers to express trigger priorities. The meaning of the order is as follows: given two triggers r_1 and r_2, $r_1 <_r r_2$ means that when r_1 and r_2 are both triggered then r_1 is considered and executed before r_2. In our model the approach is to define the priority order on triggers by combining user-defined priorities among triggers belonging to the same class c (denoted by $<_r^c$), with the order induced by inheritance relationships among classes. Thus, local priorities, specified by the user for triggers in the same class, are combined by the system with the order induced by ISA relationships[6]. To privilege the most specific behavior, the reverse ISA ordering is considered as a default for relating triggers defined in different classes. That is, given two classes c_1 and c_2, such that c_2 is subclass of c_1, each trigger r_1 defined in c_1 has priority over r_2 defined in c_2 ($r_1 <_r r_2$), that is, it is executed first. Note that this policy is in accordance with the intuition discussed in Section 4.1 for simulating trigger overriding when it is not directly supported.

Definition 4.6 *(Priority Order on Triggers). A trigger r_1 has priority over a trigger r_2 (denoted as $r_1 <_r r_2$) if either*

- $\exists\, c \in \mathcal{CI}$ *such that* $r_1 <_r^c r_2$*, or*
- $r_2.Class \leq_{ISA} r_1.Class$*.* $\square$

If for each class c the local priority ordering $<_r^c$ is an order, the priority ordering $<_r$ defined in Definition 4.6 is a (partial) order. Note that the acyclicity of the local trigger ordering

[6] Given two classes c_1 and c_2, $c_1 \leq_{ISA} c_2$ denotes that c_1 is a subclass of c_2, that is, $c_2 \in ISA^*(c_1)$.

is checked upon trigger definition. The default priority ordering of triggers obtained as in Definition 4.6 could be modified by the user in the subclass definition.

Trigger overriding is accomplished by defining a new trigger in the subclass with the same name as the inherited trigger. When a trigger r_1 is overridden by a trigger r_2 such that $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, the occurrence of an event $e \in r_1.Event$ on class $r_2.Class$ does not trigger r_1. Notice that a trigger can be overridden in a subclass only by a trigger with the same name defined in the subclass and that, viceversa, a trigger in a subclass that has the same name of one in a superclass overrides it.

4.3 Active Rule Language: Semantics

In this section we present a formal semantics of triggers. We start by providing an intuitive idea of the active rule execution model, then we formalize the notion of reactive process and finally we present the semantics. The semantics defined in this section will be the basis for investigating trigger semantic refinement, that will be discussed in Section 4.4.

4.3.1 Intuitive Idea

When one of the events of an active rule occurs, the rule is said to be *triggered*; several rules may be triggered at the same time. Trigger processing consists of an iterative execution of rule processing steps, each of which in turn consists of four phases, called rule activation, selection, consideration and execution:

- *rule activation* consists of determining the triggered rules, that is, the ones for which any of the triggering events has occurred;
- *rule selection* consists of non deterministically choosing one of the triggered rules at highest priority;
- *rule consideration* consists of evaluating the condition, which is a declarative formula; at this point the selected rule is detriggered;
- *rule execution* occurs if the condition is true, that is, produces some bindings; the execution is performed by sequentially executing the operations in the reaction part of the rule.

Trigger execution consists of updates, which may in turn trigger other rules. The rule processing activity is iterated until a state is reached where no rule is triggered. Clearly, the possibility of infinite rule processing due to chains of active rules triggering each other exists; the problem of termination of the reactive process has been extensively investigated and techniques and tools for detecting the possible sources of non-termination in a rule set have been developed [65].

A transaction in our language is a sequence of data manipulation commands each of those may trigger some rules. Our language supports a deferred execution model. Considering deferred rules means that for each command the corresponding event is added to the previous set of collected events, but no rule is executed. Only at the end of the transaction rule processing is activated, that is, the set of rules, triggered by the set of events associated with the transaction, is computed. On such set, rule selection and the other phases of rule processing are then iterated until no rule is triggered any longer. Note moreover that in our execution model events are preserved, that is, at each rule activation each rule sees the entire set of events that have occurred since the beginning of the transaction.

In this context we are only interested in describing rule semantics and not transaction semantics. In giving the semantics we suppose to have collected all the rules triggered by the events occurred during a generic transaction; thus, we have to select a rule, evaluate its condition and execute the operations in its reaction part and so on until no rule is triggered. From an intuitive point of view, each rule execution corresponds to two effects:

- collecting new rules activated by the events of the triggered rule;
- a change to the database state.

In order to formalize the semantics we have, first of all, to model the following preliminary notions.

4.3.2 Preliminaries

Before formalizing the rule language semantics, we need to introduce some preliminary definitions. In particular we introduce the notion *reactive process*. In the following the set *Rule* denotes the set of rules defined for a database. Note that though we do not formally specify a revised notion of schema to include rules, the set *Rule* is another component of the database schema.

First we have to establish, given a set of events, which is the set of rules triggered by the occurrence of events in the set.

Definition 4.7 *(Event Instance). Let e be an event as in Definition 4.1, c the class name such event is related to, and O be the set of oids of the objects affected by the event, then the triple $\langle e, c, O\rangle$ is called event instance.* $\square$

For example the event corresponding to the action $\texttt{create}(c, t, O)$ is `create` and a corresponding event instance is $\langle create, c, O\rangle$, where O denotes the set of the *oids* of the created objects[7].

For the sake of simplicity, we will often use the word event to denote event instances, when the meaning is clear from the context.

Given an event instance a relevant issue is how to establish which rule has to be triggered for each *oid* in the set of the objects affected by the event. Consider two classes c and c', where c is a subclass of c' and suppose to execute the action $\texttt{delete}(c, O)$. Since each object member of c is a member of c' as well, objects of class c have been deleted, but objects of class c' have been deleted as well. This means that the delete event is propagated along the ISA hierarchy. Triggers defined in class c', and not redefined in class c, must be activated, as class c inherits them from class c'. Moreover, since our language supports trigger overriding, given an event instance, establishing which rules have to be triggered for each object affected by the event is not trivial. Trigger overriding is accomplished by defining a new rule in the subclass with the same name as the inherited trigger. Trigger semantics, however, is complicated by the fact that usually an event instance does not trigger a single rule but a set of rules, possibly defined in different classes in the ISA hierarchy. The basic idea is to partition the set O of the objects affected by the event into disjoint subsets of *oids*, say $O_1, \ldots, O_n$, according to the most specific class of each object. Then, for each class c_i, $i = 1, \ldots, n$, the set of the triggered rules is determined through a lookup mechanism. An ascending visit of the ISA hierarchy is performed, collecting in the set of triggered rules all triggers whose event part contains the occurred event and for which a trigger with the same name has not yet been included in the set[8]. Thus, the set of rules triggered by an event is computed by taking into account that, for each object, the most specific rules are triggered, as formalized by the following definition.

Definition 4.8 *(Rule Triggered by an Event Instance). A rule $r = (N, c, Ev, C, A)$ is triggered by an event instance $\mathbf{e} = \langle e, c_e, O\rangle$ if the following conditions are verified:*

- $e \in Ev$,
- $\exists\ oid \in O$ *such that* $c = min_{\leq_{ISA}}\{\bar{c} \mid oid \in \pi^*(\bar{c})$ *and* $\exists\ \bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}$. □

We make use of the following notation. Let $oid \in O$, then $trig(oid, \langle e, c_e, O\rangle) = \{r \mid r = (N, c, Ev, C, A)$ and $e \in Ev$ and $c = min_{\leq_{ISA}}\{\bar{c} \mid oid \in \pi^*(\bar{c})$ and $\exists\ \bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}\}$.

[7] Note that O in $\texttt{create}(c, t, O)$ denotes a variable while in $\langle create, c, O\rangle$ denotes a set of *oids*.

[8] This most specific trigger overrides the one which is currently under examination.

Definition 4.9 *(React). Let* $\mathbf{e} = \langle e, c, O \rangle$ *be an event instance,* E *be a set of event instances, then* $react(\mathbf{e}) = \{r | r \in Rule \text{ and } r \text{ is triggered by } \mathbf{e}\}$ *and* $react(E) = \bigcup_{\mathbf{e} \in E} react(\mathbf{e})$. □

Definition 4.10 *(Reactive Process). Given a transaction and the set* E *of event instances associated with it,* $react(E)$ *is said to be the reactive process of the transaction.* □

Given a reactive process, that is, a set of rules, in the rule selection phase we have to choose one of the triggered rules at highest priority. Since there can be several rules at highest priority, the choice is non deterministic. We therefore introduce function *get_max*, which performs a non deterministic choice among the rules with the highest priority in a set of rules. In this context we do not deal with issues related to non determinism in rule selection. We refer the reader to [171] for an analysis of such issues.

Definition 4.11 *(Function get_max). Let* R *be a reactive process, then function* $get\_max : 2^{Rule} \rightarrow Rule$ *returns one of the rules at highest priority belonging to* R*. Formally: given* $R \in 2^{Rule}$ *if* $get\_max(R) = r$ *then* $\nexists\ r' \in R : \ r' <_r r$. □

We now introduce semantic domains and semantics functions. In defining semantic domains we refer to well-formed syntactic objects. The well-formed expressions of our language are all the trigger implementations that verify a number of static constraints. Those expressions are the ones for which a semantics is defined. Intuitively, static constraints are the constraints a trigger implementation must satisfy in order to be correct, and that can be checked at compile time, rather than at execution time. Restricting the set of well-formed expressions of a language makes defining the semantics easier. Indeed, a number of error situations need not to be considered in the semantic evaluation, because they are excluded by static constraints. Examples of static constraints for a trigger implementation are the conditions in Definition 4.5. Each time a static constraint is violated a static error raises. Static errors are opposed to dynamic errors, which are detected at run time.

Definition 4.12 *(Semantic Domains). The semantic domains we consider in giving the semantics are the following:*

- $Bind$ = *set of possible sets of bindings.*
- $State$ = *set of possible database states.*
- $Event$ = *set of possible sets of event instances.*
- $Cond$ = *set of possible condition parts of rules.*

- *Update = set of possible action parts of rules*[9].

- *Rule = set of possible rules of the language.* □

Definition 4.13 *(Semantics). The semantics of the trigger language is a family of functions defined as follows:*

$\mathcal{C} : (Cond \times State) \rightarrow ((State \times Event) \rightarrow Bind)$

$\mathcal{U} : (Update \times State) \rightarrow ((Bind \times State \times Event) \rightarrow (Bind \times State \times Event))$

$\mathcal{R} : (Rule \times State) \rightarrow ((State \times Event) \rightarrow (State \times Event))$

$\mathcal{P} : (2^{Rule} \times State) \rightarrow ((State \times Event) \rightarrow State)$. □

Function $\mathcal{C}$ models condition evaluation; function $\mathcal{U}$ models action execution, whereas functions $\mathcal{R}$ and $\mathcal{P}$ model reactive processing semantics. We notice that the argument of the semantics functions are pairs of the form $\langle$*construct to be evaluated*, *state*$\rangle$, where the state, with our assumption of event preserving rules, is the one at the beginning of the transaction. We need the state as argument for three reasons: (*i*) evaluation of *old* formulas; (*ii*) dynamic errors; (*iii*) rollback.

In the first case to evaluate formulas of the form `old`(F) we need to evaluate the formula F in the state at the beginning of the transaction. In case of dynamic errors in evaluating a rule, according to the transaction *all or nothing* philosophy, the transaction must be aborted and the initial state must be restored. In our model the reactive process is considered as part of the transaction itself. If, by contrast, a decoupled execution model is considered, the reactive process is a separate transaction and, thus, if a dynamic error occurs during the reactive process evaluation, only the reactive transaction must be aborted and the resulting state must be the one at the end of the user transaction. Since the execution model we refer to does not support decoupled rule execution, we do not elaborate further in this context on dynamic errors in trigger execution semantics with respect to different execution models. A typical dynamic error can be a division by zero in a condition. For lack of space we do not analyze here the different types of dynamic errors.

In what follows, to simplify the notation, we will often omit the argument S_i, the initial state, if it is not strictly necessary.

During the execution of the transaction, events (event instances) are collected. When the transaction terminates two situations can arise:

- the transaction ends with a rollback; in this case the resulting state is the one at the beginning of the transaction;

[9] *Update* corresponds to the set of well-formed update sequences of the language.

- the transaction ends with a commit; in this case the reactive process is activated and all rules triggered by the events occurred during the transaction are executed.

Function $\mathcal{P}$ models the semantics of the reactive process associated with a transaction by establishing a transformation from the state at the end of the transaction to the state at the end of the reactive process. Thus, function $\mathcal{P}$, at each step: (i) selects from the set of rules constituting the reactive process the rule at highest priority through function get_max, let r be this rule; (ii) evaluates r through function $\mathcal{R}$; (iii) deletes r from the reactive process; (iv) adds the rules triggered by the action part of r to the reactive process; (v) evaluates the new set of rules with function $\mathcal{P}$ on the state resulting from the evaluation of r through function $\mathcal{R}$ and so on. The evaluation of a rule r is performed by function $\mathcal{R}$ and consists of evaluating the condition of r using function $\mathcal{C}$, obtaining a set of bindings and evaluating through function $\mathcal{U}$ the action of r on this set of bindings. The result of the evaluation of a rule is a new database state and a new set of events. No set of bindings is given as result because in triggers bindings are local to rules.

4.3.3 Semantics: Condition

Let C be a rule condition (a formula), S be a database state and E be a set of event instances, then:

$$\mathcal{C}\,[\![\, C_{S_i} \,]\!]\, SE = B$$

where each substitution $\theta \in B$ is such that the instantiation of C with respect to it, that is $C\theta$, evaluates to true in state S and with respect to events in E, according to first order logic.

As several treatments of semantic evaluation for first order logic formulas can be found in the literature [87], we do not discuss them in this context. Rather, we focus on event and *old* formulas which are typical of trigger conditions.

First we analyze the semantics of `occurred` formulas. We have to evaluate the semantics of a formula of form $\texttt{occurred}(e, X)$ appearing in the condition part of a trigger r, given a database state S and a set of event instances E. Notice that, during a transaction, a trigger r can be activated by several update operations. This fact corresponds to the insertion of r in the reactive process of the transaction and to the insertion in E of event instances activating r. Thus, when we evaluate $\texttt{occurred}(e, X)$ in r we have to take into account of all the event instances that have activated r. Moreover for each event instance $\mathbf{e} = \langle e, c_e, O\rangle$ that has activated r, we have to choose only the *oids* belonging to O which actually have activated r, that is, those *oids* for which r is the most specific trigger. Formally, the evaluation of the formula $\texttt{occurred}(e, X)$ in trigger r is:

$$\mathcal{C}\,[\![\, \texttt{occurred}(e, X)_{S_i} \,]\!]\, SE = B$$

where $B = \{\{X/oid\} \mid \exists \langle e, c, O\rangle \in E \text{ and } r \in trig(oid, \langle e, c, O\rangle)\}$.

For what concerns `old` formulas, as we consider event preserving triggers, the formula F in `old`(F) must be evaluated in the state at the beginning of the transaction, that is, S_i. Formally the semantics is:

$$\mathcal{C}\,[\![\, \mathtt{old}(F)_{S_i} \,]\!]\, SE = \mathcal{C}\,[\![\, F_{S_i} \,]\!]\, S_i E.$$

4.3.4 Semantics: Action

Function $\mathcal{U}$ models the semantic evaluation of a sequence of updates, u_1;...;u_n, that is, the action part of a rule r. The semantics of update concatenation is quite intuitive. At each step the first update of the sequence is evaluated. Such evaluation gives as result a new set of bindings, a new state and a new set of event instances with respect to which the remainder of the sequence is evaluated. Formally:

$$\mathcal{U}\,[\![\, u_1; \ldots; u_n \,]\!]\, BSE = \mathcal{U}\,[\![\, u_2; \ldots; u_n \,]\!]\, (\mathcal{U}\,[\![\, u_1 \,]\!]\, BSE).$$

For what concerns the evaluation of a single update, function $\mathcal{U}$ is specified for each different type of atomic update, that is, *create*, *delete* and so on, and for method calls. Here we only specify the event component of the semantics, since the binding and state components are as specified in Figure 3.5.

$$\mathcal{U}\,[\![\, \mathtt{create}(\bar{c}, \bar{t}, \bar{O}) \,]\!]\, BSE = \langle B', S', E \cup \{\langle create, \bar{c}, \bar{O}_B\rangle\}\rangle$$

$$\mathcal{U}\,[\![\, \mathtt{delete}(\bar{c}, \bar{O}) \,]\!]\, BSE = \langle B', S', E \cup \{\langle delete, \bar{c}, \bar{O}_B\rangle\}\rangle$$

$$\mathcal{U}\,[\![\, \mathtt{specialize}(c_1, c_2, \bar{O}, \bar{t}) \,]\!]\, BSE = \langle B', S', E \cup \{\langle specialize(c_2), c_1, \bar{O}_B\rangle\}\rangle$$

$$\mathcal{U}\,[\![\, \mathtt{generalize}(c_1, c_2, \bar{O}) \,]\!]\, BSE = \langle B', S', E \cup \{\langle generalize(c_2), c_1, \bar{O}_B\rangle\}\rangle$$

$$\mathcal{U}\,[\![\, \mathtt{modify}(\bar{c}.a, \bar{O}, \bar{t}) \,]\!]\, BSE = \langle B', S', E \cup \{\langle modify(a), \bar{c}, \bar{O}_B\rangle\}\rangle$$

Moreover, $\mathcal{U}\,[\![\, \bar{O}.m(\bar{t}_1, \ldots, \bar{t}_n) \,]\!]\, BSE = \langle B', S', E \cup \{\langle m, \bar{c}, \bar{O}_B\rangle\}\rangle$ where B' and S' are as specified by Definition 3.26. One of the problems arising in defining function $\mathcal{U}$ is method selection with respect to inherited triggers. Consider a trigger r defined in a class c' and invoking in its action an operation m on the objects affected by the event. Consider moreover a subclass c of c' and suppose that operation m is redefined in c. Rule r is triggered when the event monitored by r occurs both on objects instances of c' and on objects instances of c. For objects instances of c' the method implementation in class c' is selected, where the trigger itself is defined. By contrast, for objects instances of c two different options are possible: (i) choosing the most specialized implementation of m (that is, the implementation in class c); (ii) choosing the implementation according to the class where the rule is defined (that is, the implementation in class c'). We refer to the first and second approach as *object-specific method selection* and *rule-specific method selection*, respectively. Our choice is the first one, because it is consistent with the object-oriented approach, in that it conforms to the principle of exhibiting the most specific behavior. The rule-specific method selection is not consistent with the object-oriented approach because it refers to the static nature of objects, that is, the class in which the trigger is defined, and not to their dynamic nature, that is, the classes the objects are proper instances of.

Even though the rule-specific method selection is not coherent with the object-oriented approach, it is used in some active object-oriented database systems, like Ode [91]. In our opinion, rule-specific method selection can be useful in some cases. For this reason, a possibility is to allow the user to explicitly require it, by means of an ad-hoc syntactic construct, in a way like *strict binding* can be required, e.g. in Fibonacci [12].

By supporting trigger overriding and object-specific method selection, our execution model is fully object-oriented. For each object affected by an event triggering a rule, the most specific trigger defined for that object is triggered and, during trigger execution, if a method is invoked in the trigger action, for each considered object, the most specific method implementation for that object is chosen.

Our strategy is based on the assumption that methods redefined in subclasses are a behavioral refinement of corresponding methods in superclasses, according to the behavior refinement constraints imposed on method redefinitions in Chapter 3.

To model reactive process activation, at transaction commit, the following semantics is specified for the `rollback` and `commit` transactional statements:

$$\mathcal{U}\,[\![\, \texttt{rollback}_{S_i} \,]\!]\, BSE = \langle \emptyset, S_i, \emptyset \rangle$$

$$\mathcal{U}\,[\![\, \texttt{commit}_{S_i} \,]\!]\, BSE = \langle \emptyset, \mathcal{P}\,[\![\, react(E)_{S_i} \,]\!]\, SE, \emptyset \rangle.$$

Notice that since `rollback` and `commit` are the last command of a transaction, the output values of B and E are set to $\emptyset$ because they are not meaningful. The only interesting value is the resulting state, the second component of the result.

4.3.5 Semantics: Reactive Process

Now we can formally define the semantic functions $\mathcal{R}$ and $\mathcal{P}$.

Definition 4.14 *(Rule Semantics). Let $r = (N, c, Ev, C, A)$ be a rule, let E be a set of event instances, let S be a database state, and, finally, let $\mathcal{C}\,[\![\, C \,]\!]\, SE = B$, then:*

$$\mathcal{R}\,[\![\, r \,]\!]\, SE = \langle S', E' \rangle$$

where $S' = \Pi_2(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$ and $E' = \Pi_3(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$. □

Notice that the set of bindings, resulting from the evaluation of the sequence $u_1; \ldots; u_n$, that is, $\Pi_1(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$, does not appear in the previous definition because in triggers there is no variable passing among rules in the reactive process. The semantics of a reactive process is given by the following definition.

Definition 4.15 *(Reactive Process Semantics). Let R be a set of rules, E be a set of event instances, and S be a database state, $\mathcal{P}\,[\![\, R \,]\!]\, SE$ is defined as follows:*

$$\mathcal{P}\,[\![\, R \,]\!]\, SE = \begin{cases} S & \textit{if } R = \emptyset \\ \mathcal{P}\,[\![\, R \setminus \{r\} \cup react(E') \,]\!]\, S'E \cup E' & \textit{if } R \neq \emptyset, r = get\_max(R), \\ & \quad \textit{and } \mathcal{R}\,[\![\, r \,]\!]\, SE = \langle S', E' \rangle. \end{cases}$$

□

Notice that the recursive definition of semantic function $\mathcal{P}$ corresponds to the idea that the reactive process is iterated till a quiescent state is reached. When there are no more triggered rules, that is, $R = \emptyset$, the reactive process stops and the current state is returned. The reactive process semantics can then be seen as the least fixpoint of function $\mathcal{P}$.

4.4 Trigger Refinement

As we have discussed in Section 4.1, in some situations a class must be able to redefine a trigger of one of its superclasses, instead of simply inheriting it. Rule overriding is supported in some systems such as TriGS [104] and Ode [125], but no restrictions are imposed on rule overriding, thus a rule may override another rule on completely different events.

In our model, as in those systems, trigger overriding is directly supported. However, we believe that trigger redefinition must be carefully handled and is, therefore, subject to a number of restrictions. In particular, in order to preserve trigger semantics, it must be ensured that the trigger in the subclass is executed at least each time the trigger in the superclass would be executed, and that what would be executed by the trigger in the superclass is also executed by the refined trigger. In this case we say that the trigger in the subclass is a *behavioral refinement* of the trigger in the superclass. The topic of behavioral refinement for methods have already been discussed in Chapter 3. Some of the issues dealt with in this section closely resemble similar issues dealt in Section 3.4.1 in the context of method refinement. However, for the sake of clarity, we prefer to revisit these notions in the context of triggers.

A trigger r_2 is a behavioral refinement of a trigger r_1 if the portion of state manipulated by r_2 includes the portion of state manipulated by r_1 and the portion of state modified by both is modified in the same way[10]. To formally define this notion, we must first model the changes made by a trigger execution. Given a trigger r, let $\mu(r)$ be the set of classes manipulated by r (through any of the atomic update primitives)[11]. Given a trigger r and a class c, let $\delta_r(c)$ be the set of objects deleted from class c and $\iota_r(c)$ the set of objects inserted in class c as a consequence of the execution of trigger r; moreover, given an object oid and an attribute name a, let $\nu_r(oid).a$ be defined if and only if the execution of trigger r has modified the value of attribute a of the object identified by oid, and, if defined, let it contain the new value of the attribute. Those notions are formally defined as follows.

Definition 4.16 *(Portion of State Manipulated by a Trigger). Given a trigger r such that $\mu(r)$ is the set of classes manipulated by r and a database state S, such that $\mathcal{R}[\![r]\!] SE = \langle S', E' \rangle$ consider the following functions:*

- $\forall\ c \in \mu(r) : \delta_r(c) = S.\pi(c) \setminus S'.\pi(c)$
- $\forall\ c \in \mu(r) : \iota_r(c) = S'.\pi(c) \setminus S.\pi(c)$
- $\forall\ c \in \mu(r),\ \forall\ oid \in S'.\pi(c) : \nu_r(oid).a = v$ *if and only if the execution of trigger r has modified the value of the attribute a of object oid setting it equal to v.*

The triple $\langle \delta_r, \iota_r, \nu_r \rangle$ models the portion of state manipulated by trigger r. □

Definition 4.17 *(Behavioral Trigger Refinement). Trigger r_2 is a* behavioral refinement *of trigger r_1, with $r_2.Name = r_1.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, if $\mu(r_1) \subseteq \mu(r_2)$ and if, for each database state, the execution of r_1 and r_2 restricted to the objects in $r_2.Class$ satisfies the following conditions:*

[10]Note that a trigger executed on an object (set of objects) instance(s) of a class may manipulate objects of other classes.

[11]They can be deduced syntactically.

- *$\forall \bar{c} \in \mu(r_1)$: $\delta_{r_1}(\bar{c}) \subseteq \delta_{r_2}(\bar{c})$ and $\iota_{r_1}(\bar{c}) \subseteq \iota_{r_2}(\bar{c})$;*
- *$\forall \bar{c} \in \mu(r_1), \forall a$ attribute of $\bar{c}$, $\forall$ oid instance of $\bar{c}$: if $\nu_{r_1}(oid).a$ is defined, then $\nu_{r_2}(oid).a$ is defined and $\nu_{r_1}(oid).a = \nu_{r_2}(oid).a$.* □

Unfortunately, trigger refinement is undecidable. In fact, given a database state in which r_1 and r_2 are executed and two result states, S_{r_1} and S_{r_2}, resulting from the execution of r_1 and r_2, respectively, respect to the portion of state manipulated by r_1 and r_2 is computed, it is easy to verify whether the two conditions of Definition 4.17 hold. However, a trigger r_2 is a behavioral refinement of a trigger r_1 if each database state verifies the conditions of Definition 4.17. Intuitively given a database schema the possible database states on such schema are an infinite number, thus trigger refinement is undecidable. We have, however, devised some sufficient static conditions ensuring that a trigger r_2 is a refinement of a trigger r_1. These conditions can be checked at trigger definition time, so that the overriding of a trigger in a subclass can be disallowed if the overriding trigger is not a refinement of the overridden one. In what follows we illustrate those conditions, by first analyzing each trigger component separately.

4.4.1 Events

The refined trigger must be activated each time the inherited trigger would be activated. Thus, we impose the condition that for each event in the event component of the inherited trigger, a corresponding event is present in the event component of the refined trigger.

Definition 4.18 *(Event Refinement). An event set Ev is a refinement of an event set Ev' if and only if $Ev' \subseteq Ev$.* □

Example 4.5 *Consider a class* `person`*, with an attribute* `income`*, and a trigger that if the value of income is less than* 0*, assigns* 0 *to the income. The event component of that trigger is the set* { `modify(income)` }. *Consider moreover a class* `employee`*, subclass of* `person`*, with two attributes* `income` *and* `fees` *and suppose to refine the trigger so that whenever the amount of income minus the amount of fees is less than* 0*, the attribute* `income` *is set to the value of attribute* `fees`*. The event component of the refined trigger is* { `modify(income)`, `modify(fees)` }. ◇

4.4.2 Condition

The basic idea is that the action of the refined trigger must be executed each time the inherited trigger action would be executed, and for each binding for which the inherited trigger action would be executed. Thus, the condition in the refined trigger must be less selective than the condition in the inherited one (that is, each binding returned by the condition of the inherited trigger must be returned by the condition of the refined one).

Example 4.6 *Consider the classes* `person` *and* `employee`*, and the constraint that the age of a person must be greater than 0, while the age of an employee must be greater than 15. This constraint is enforced in* `person` *by a trigger with the following event and condition components:*

> *Events:* `modify(age)`
> *Condition:* `person(X), occurred(modify(age),X), X.age < 0`

The constraint is enforced in `employee` *by a refined trigger with the following events and condition components:*

> *Events:* `modify(age)`
> *Condition:* `employee(X), occurred(modify(age),X), X.age < 15` ◇

In what follows, we formalize this notion. The bindings produced by the evaluation of the condition are represented as a set of ground substitutions, as defined in Chapter 3. We recall that a substitution ρ is a *renaming* of variables if $\rho = \{X_1/t_1, \ldots, X_n/t_n\}$, and, for each $i, 1 \leq i \leq n$, t_i is a distinct variable, and that given a renaming of variables ρ, -^{ρ} denotes subset (non strict) inclusion modulo renaming ρ, whereas $\mathtt{w}^{\rho}$ denotes equality modulo renaming ρ,

Moreover, we introduce the following notations. Given a trigger r, let $BVar(r)$ denote the set of variables appearing in $r.Condition$ and in $r.Action$ (that is, the variables employed for passing bindings). Given a formula F, let F^{-E} denote the formula obtained by eliminating the event formulas appearing in F.

Definition 4.19 *(Condition Refinement). A condition $r_2.Condition$ is a refinement of a condition $r_1.Condition$ (denoted as $r_2.Condition \leq_c^{\rho} r_1.Condition$) if and only if the following conditions hold:*

1. *$BVar(r_1) \text{-}^{\rho} BVar(r_2)$, and*
2. *for each event formulas $occurred(e, X)$ in $r_1.Condition$, a corresponding event formulas $occurred(e, X')$ is in $r_2.Condition$, such that $X \mathtt{w}^{\rho} X'$, and*

3. $\forall S$ database state,

$$\mathcal{C}\,[\![\, r_1.Condition^{-E}[r_1.Class/r_2.Class]\,]\!]\, S\emptyset \preceq^{\rho} (\mathcal{C}\,[\![\, r_2.Condition^{-E}\,]\!]\, S\emptyset)_{|BVar(r_1)}$$ [12]

for a renaming of variables ρ. □

Condition 3 of Definition 4.19 above is that $r_2.Condition^{-E}$ *subsumes* $r_1.Condition^{-E}$, restricted to be evaluated on the same class and to return the same variables. Query subsumption (also called query containment) has been widely investigated, and algorithms for deciding subsumption among object-oriented queries have been proposed [50, 69]. Subsumption can be easily extended to handle also predicates on past database states, as long as the referred past state is the same state in both formulas[13]. Indeed, `old`(F) subsumes `old`(G) if and only if F subsumes G.

We remark that, though the test for subsumption has a cost exponential in the formulas that can appear in rule conditions, this does not constitute a problem. First of all, this complexity is exponential in the dimension of the formula and the schema, not in the dimension of the database. Second, the analysis of whether a rule is a correct refinement of another one is executed once at rule definition time. More efficient subsumption tests for rule conditions could be used at the expense of reducing the expressiveness of the language for condition specification [50].

Example 4.7 *Given trigger r_2 on class* `employee` *and trigger r_1 on class* `person` *such that*

- $r_2.Condition$ = `employee(X), occurred(modify(age),X), X.age > 65,`
  `department(Y), X.department = Y`, *and*
- $r_1.Condition$ = `person(X), occurred(modify(age),X), X.age > 100`

$r_2.Condition$ is a refinement of $r_1.Condition$ under the empty renaming of variables ϵ, indeed:

1. $BVar(r_1) = \{\mathtt{X}\} \preceq^{\epsilon} BVar(r_2) = \{\mathtt{X,Y}\}$;
2. `occurred(modify(age),X)` *is in $r_1.Condition$, and* `occurred(modify(age),X)` *is in*
   $r_2.Condition$;

[12] We recall that given a formula F and two class identifiers c_1 and c_2, let $F[c_1/c_2]$ denote the formula obtained from F by substituting each class formula $c_1(X)$ with a class formula $c_2(X)$.

[13] Note that this is the case here, since we consider always refer to the state at transaction start.

3. $r_1.Condition^{-E}[r_1.Class/r_2.Class]$ = `employee(X), X.age > 100` *is subsumed by* $r_2.Condition^{-E}$ = `employee(X), X.age > 65, department(Y), X.department = Y` *restricted to variable* `X`. $\diamond$

4.4.3 Action

The basic idea is to ensure behavior consistency, that is, the action of the refined trigger must do at least all what the action of the inherited trigger would do. This means that, for each action in the inherited trigger there must be a corresponding action in the refined one. However, since rule action are sequences, the corresponding action could be discarded by some complementary action executed after it in the sequence. Consider as an example the case of an inherited trigger creating an object in its action, overridden by a trigger whose action first of all creates a corresponding object and then deletes it. We consider, therefore, the net effect of the actions in the sequence and we state that for each action in the inherited trigger there must be a corresponding action in the net effect of the refined one. Note that the notion of net effect employed here is purely syntactical and relies only on complementary database operations.

Definition 4.20 *(Basic Action Refinement). An action u is a refinement of an action u' under the renaming of variables ρ (denoted as $u \leq_b^\rho u'$) if and only if one of the conditions of Definition 3.32 holds or if $u = u' =$* `rollback`. $\square$

Net effect computation has already discussed in Chapter 3. We recall here the definition of action refinement given in Capter 3 (Definition 3.33).

Definition 4.21 *(Action Refinement). A reaction $r_2.Action$ is a refinement of a reaction $r_1.Action$ (denoted as $r_2.Action \leq_a^\rho r_1.Action$) if the following conditions hold:*

- *$Net(r_1.Action) = u'_1; \ldots ; u'_n$, $Net(r_2.Action) = u_1; \ldots ; u_m$ and $m \geq n$;*
- *for each u'_i, $1 \leq i \leq n$, in $Net(r_1.Action)$, u_j, $1 \leq j \leq m$, in $Net(r_2.Action)$ exists, such that $u_j \leq_b^\rho u'_i$, that is, u_j is a refinement of u'_i according to Definition 4.20; let function $\xi : \{1, \ldots , n\} \rightarrow \{1, \ldots , m\}$, such that $\xi(i) = j$, model this correspondence;*
- *if u'_i precedes u'_k in $Net(r_1.Action)$ $u_{\xi(i)}$ precedes $u_{\xi(k)}$ in $Net(r_2.Action)$.* $\square$

We remark that, since both basic action refinement and the computation of net effect only rely on syntactical properties of the trigger action, action refinement is decidable. Note, that our notion of corresponding action is correct since a notion of behavior refinement is imposed on method redefinition. Indeed, we can ensure that if an operation m is invoked in the action of the inherited trigger, then operation m is invoked in the action of the refined one. Conditions on behavioral subtyping has been presented in Chapter 3.

4.4.4 Restrictions on Trigger Overriding

The following subsection summarizes the restrictions on trigger redefinition.

Definition 4.22 *(Static Trigger Refinement). A trigger $r_2 = (N, c, Ev, C, A)$ is a* static refinement *of a trigger $r_1 = (N', c', Ev', C', A')$, if ρ, renaming of variables, exists such that the following conditions are satisfied:*

- *$Ev' \subseteq Ev$, that is, Ev is a refinement of Ev' according to Definition 4.18;*
- *$C \leq_c^\rho C'$, that is, C is a refinement of C' according to Definition 4.19;*
- *$A \leq_a^\rho A'$, that is, A is a refinement of A' according to Definition 4.21.* □

The following properties on redefined triggers hold.

Proposition 4.1 *Given two triggers r_1 and r_2:*

1. *we can decide whether r_2 is a static refinement of r_1;*
2. *if r_2 is a static refinement of r_1, $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, each event on proper instances of $r_2.Class$, that would trigger r_1 triggers r_2;*
3. *if r_2 is a static refinement of r_1, $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, then r_2 is a behavioral refinement of r_1 according to Definition 4.17.*

Proof Sketch: The proof of item 1 is based on the fact that event, condition and action refinement are decidable. In fact, each of such notions is based on a static checking on trigger implementation which can be easily done at compile time. For example, event refinement simply means to check an inclusion between two finite sets of events.

The proof of item 2 is based on the fact that if r_2 is a static refinement of r_1 then $r_1.Event \subseteq r_2.Event$. By Definition 4.8, r_1 is triggered by an event instance $\mathbf{e} = \langle e, c_e, O \rangle$ if $e \in r_1.Event$ and $\exists$ $oid \in O$ such that $r_1.Class = min_{\leq_{ISA}}\{\bar{c} \mid oid \in \pi^*(\bar{c})$ and $\exists$ $\bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}$ with $N = r_1.Name = r_2.Name$. The fact $e \in r_1.Event$ implies $e \in r_2.Event$ for hypothesis. Moreover for such oid proper instances of $r_2.Class$ belonging to O, $r_2.Class = min_{\leq_{ISA}}\{\bar{c} \mid oid \in \pi^*(\bar{c})$ and $\exists\ \bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}$, then r_2 is triggered.

The proof of item 3 is more complex. First we have to prove that if the three properties of Definition 4.22 are verified, $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$ then $\mu(r_1) \subseteq \mu(r_2)$ which follows from the fact that $r_2.Action \leq_a^\rho r_1.Action$ for hypothesis. In fact, $r_2.Action \leq_a^\rho r_1.Action$ implies $Net(r_2.Action) \leq_a^\rho Net(r_1.Action)$ that, for Definition 4.20 and Definition 4.21, means that each class c appearing in an action in $Net(r_2.Action)$ is a subclass of a class c' appearing in an action in $Net(r_1.Action)$ ($c \leq_{ISA} c'$). Note, moreover that if a trigger manipulates a class c, it manipulates each class c' such that $c \leq_{ISA} c'$, then, if for each c manipulated by r_2 exists a superclass c' of c manipulated by r_1 it follows that $\mu(r_1) \subseteq \mu(r_2)$.

For what concerns the two conditions of Definition 4.17, let us consider, for example, the first part of the first one: $\forall \bar{c} \in \mu(r_1)$: $\delta_{r_1}(\bar{c}) \subseteq \delta_{r_2}(\bar{c})$. We remember that given a trigger r and a class c, $\delta_r(c)$ denotes the set of objects deleted from class c by r. Let us focus our attention on the objects deleted through a single delete operation.

For hypothesis, $r_2.Condition \leq_c^\rho r_1.Condition$, that is, $r_2.Condition$ subsumes $r_1.Condition$. This means that given a variable O' belonging to $BVar(r_1)$ and the corresponding, according to ρ, variable O belonging to $BVar(r_2)$, then $O'_{B_{r_1}} \subseteq O_{B_{r_2}}$ where $\mathcal{C}\,[\![\, r_1.Condition \,]\!]\, SE = B_{r_1}$ and $\mathcal{C}\,[\![\, r_2.Condition \,]\!]\, SE = B_{r_2}$. Moreover, as $r_2.Action$ is a refinement of $r_1.Action$ according to Definition 4.21 if $delete(c', O')$ exists in $Net(r_1.Action)$ then $delete(c, O)$ exists in $Net(r_2.Action)$ such that $delete(c, O) \leq_b^\rho delete(c', O')$ according to Definition 4.20, that is $c \leq_{ISA} c'$ and $O \sqsupseteq^\rho O'$. Informally, this fact, together with the fact that $O'_{B_{r_1}} \subseteq O_{B_{r_2}}$, means that starting from the same database state S and the same event set E, the objects deleted by $delete(c, O)$ includes the objects deleted by $delete(c', O')$, which is exactly what we would prove. The proof of the other conditions of Definition 4.17 are similar. $\square$

We can state the following rule.

Rule 4.1 *Given the set of triggers R of a database schema, for each r_1, r_2 belonging to R if $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$ then r_2 must be a static refinement of r_1 according to Definition 4.22.* $\diamond$

Note moreover that, according to Proposition 4.1, r_2 is a behavioral refinement of r_1.

Finally, let us mention that whenever the conditions of Definition 4.22 hold among two triggers, then, under certain constraints, the more general of the two triggers is redundant, as stated by the following definition.

Definition 4.23 *(Redundant Trigger). A trigger* $r_2 = (N, c, Ev, C, A)$ *is redundant with respect to a class* c *if a trigger* $r_1 = (N', c', Ev', C', A')$ *exists such that* $c \leq_{ISA} c'$ *and:*

- $Ev \subseteq Ev'$*, that is,* Ev' *is a refinement of* Ev *according to Definition 4.18;*
- $C' \leq^{\rho}_{c} C$*, that is,* C' *is a refinement of* C *according to Definition 4.19;*
- $A' \leq^{\rho}_{a} A$*, that is,* A' *is a refinement of* A *according to Definition 4.21.* ◇

Example 4.8 *Consider the classes* `person` *and* `employee`*, with* `employee` *subclass of* `person`*, and the following trigger* r_1 *on class* `person`*:*

> *Events:* `modify(age)`
> *Condition:* `person(X), occurred(modify(age),X), X.income < 0`
> *Action:* `rollback`

Suppose that the following trigger r_2 *is defined on class* `employee`*:*

> *Events:* `modify(age)`
> *Condition:* `employee(X), occurred(modify(age),X), X.income < 100`
> *Action:* `rollback`

Then, trigger r_1*, inherited in class* `employee`*, is redundant with respect to this class.* ◇

4.5 Related Work

In this section we compare the work reported in this chapter with existing approaches to active rule semantics and trigger overriding.

4.5.1 Active Rule Semantics

The growing interest in active database systems has led to the development of formal techniques to analyze the main characteristics of these systems as the rule definition language and the execution model. In the following we present a brief overview of how the problem of defining a semantics for active database systems has been addressed. We refer the interested reader to [140] for a complete overview.

The approach to the definition of a semantics which is the most similar to ours is described in [170] and proposes a denotational semantics for the Starbust system. As in our approach, the semantics is seen as a function which maps a transaction and the current database state into a resulting state. The main limitation of these semantics is that it is tightly bound to the Starbust execution model, devoting little attention to the condition and action language.

Another relevant work is [149], where an attempt to develop a generic environment supporting the formalization of the semantics of different systems is proposed. The formalism used is object-oriented but totally abstract. The main drawback of this approach, like the previous one, is that no aspect concerning the active rule language is analyzed.

Another line of thinking is the one which has adapted the semantics of deductive databases to active systems. The main works in this direction are [120] and [175]. In [120] the database version of the *Event Calculus* is exploited. Through the Event Calculus, the set of logical consequences derived by the event history creates a sequence of sets of facts, each of these facts can be seen as the extensional part (EDB) of a deductive database (DDB). The main contribution of this approach is the definition of specification languages for active rule languages with a logical semantics. In [175] an integration between the classical operational semantics of the deductive databases and the semantics of active databases is described. The approach is based on the syntactical notion of *XY-stratification* and underlines the declarative nature of database updates. Such an approach, however, does not support the analysis of some important features of active rules, concerning the execution model, such as coupling modes.

A simple specification language of the condition-action form is introduced in [176]. The aim of that approach is to develop a rule execution formalism for reasoning about the basic properties of rules. A drawback is represented by the restrictive assumptions characterizing the approach, for example a single coupling mode is supported.

The last approach [85] we mention here is particularly relevant as it gives a semantics for the Chimera active rule language. A structured approach for the definition of active database semantics is proposed. Such approach, applicable to a generic active database system, tries to model the various aspects of such systems, such as execution of a set of rule, coupling modes, consumption modes and so on. Three steps are distinguished: the first step consists of a translation of the rules into an intermediate syntax to make explicit the characteristics of the rules of a particular system; the second step consists of a translation of the set of rules, obtained in the previous step, in an inner format (*core format*); the third step consists of an execution model, specified through an algorithm, for the rules expressed in the core format. The main drawback of this approach is that rule semantics is not specified directly, rather through the translation of rules in an auxiliary format, and thus it is less intuitive.

This short overview highlights the fact that most proposals formalizing active database system semantics deal with relational systems rather than with object-oriented ones and the few that follow the object-oriented approach do not address issues related to inheritance and overriding. These aspects, so far neglected, are the starting point of our work and this is why we address the problem from a point of view which completely differs from the others, and which is particularly relevant for the object-oriented model.

4.5.2 Overriding in Active Object Systems

Under most proposals [26, 75, 132], triggers are always inherited and can never be overridden nor refined. Such an approach, that we have referred to as *full trigger inheritance*, simply means that events are propagated across the class inheritance hierarchy. Thus, inheritance of triggers is accomplished by applying a trigger to all the instances of the class in which the trigger is defined, rather than only to proper instances of this class. The approach proposed by Shyy and Su [160] achieves the same result, but it is complicated by the fact that events are not propagated along the class hierarchy.

Rule overriding is supported in TriGS [104]. A lookup mechanism is used that starts from the class of the object for which the event was signaled, and ends at the class where the method corresponding to the event[14] is defined (in the worst case at the root of the class hierarchy). To recognize rule overriding, a mechanism based on the equality of rule names is used. At the implementation level, a transient rule dictionary is used to filter overridden rules. In TriGS, however, this process is simplified since only single inheritance is considered. No restrictions are imposed on rule overriding, thus a rule may also override another rule on completely different events.

In Ode [125] a subclass may contain a different definition for a trigger defined in a superclass. Then, if the trigger is activated on a subclass object, only the most specific trigger applies to it, thus trigger overriding is supported. Note, however, that if the trigger activation is part of the superclass constructor, than both triggers apply to a subclass object, and there is no way to override the trigger. Moreover, also in Ode no controls are performed on trigger redefinition; thus a rule may override another rule on completely different events.

In [75] the possibility is suggested to program rule overriding "by hand" as follows. Suppose that a trigger $r' = (N, c', Ev', C', A')$ is defined and suppose, moreover, that it needs to be overridden by a rule with the same events Ev', but different conditions and actions, C and A, in a class c, with c subclass of c'. Then, in class c a trigger r can be declared such that r has priority over r', and $r.Action$ executes A and deactivates (that is, disables) trigger r'. With this approach, only r will be executed, but it is responsibility of the rule programmer to enforce the overriding. This approach, however, does not account for true overloading and overriding, as the new trigger r cannot have the same name as trigger r'. Other drawbacks of this approach are that only few systems allow a rule to be disabled in

[14] We recall that in TriGS events are only method calls.

the action of another rule and that most systems require rules to be explicitly re-activated once deactivated; therefore, it is not clear when and by whom rule r' is re-activated.

4.6 Concluding Remarks

In this chapter we have defined the syntax and the semantics of our active rule language, supporting trigger inheritance and overriding. We have moreover investigated under which restrictions triggers can be overridden in subclasses.

Our set-oriented rule language is a subset of the Chimera active rule language [61]. Active rules in Chimera have several innovative features: (*i*) they support optional composition of event effects (called *net effect computation*), when the same object is the target of multiple operations, (*ii*) they support different *event consumption modes*, that is, different models for processing events, (*iii*) they support different *processing modes*, that is, different activation times, and finally (*iv*) they provide mechanisms for accessing intermediate states of affected objects during transaction execution. In Chimera, the processing mode of an active rule (which is the coupling mode between event and condition) may be either *immediate* or *deferred*. Immediate rules are considered for execution at the end of the transaction unit or reaction in which triggering occurs. Deferred rules are processed at the end of the transaction (after the *commit* command). In Chimera, two distinct event consumption modes are possible for each active rule; this feature is relevant when a given rule is considered multiple times within the same transaction. Events can be *consumed* after the consideration of a rule, therefore, each event instance is considered by a rule only at its next execution, and then disregarded. Alternatively, events can be *preserved*, that is, all events since the transaction starting are considered at each rule consideration. Our language, by contrast, supports only deferred, preserving rules without net effect composition. Moreover, we consider here only rules targeted to classes, disregarding untargeted ones. These restrictions allows us to focus on problems related to trigger inheritance and overriding. The proposed approach can however be generalized to the full Chimera language. By restricting ourselves to a simpler rule execution model, a clear understanding of rule overriding is achieved.

Chapter 5

Views

The definition of a view mechanism is an important issue for object-oriented database systems, in order to provide a number of features that are crucial for the development of advanced applications. Due to the complexity of the data model, the object-oriented paradigm introduces new problems in the definition of a view mechanism. Several approaches have been defined, each defining a particular view mechanism tailored to a set of functionalities that the view mechanism should support. In particular, views can be used as shorthand in queries, can support the definition of external schemas, can be used for content-dependent authorization, and, finally, can support some form of schema evolution. In this chapter, we define a view mechanism for our data model, thus formally defining a view model for object-oriented databases.

The chapter is organized as follows. In Section 5.1 we discuss the main alternatives in designing a view mechanism for object databases, also comparing the view mechanism we propose with existing approaches in the literature. Section 5.2 introduces views, that is, virtual classes, whereas Section 5.3 discusses view instances, introducing the notion of virtual object. Section 5.4 is devoted to inheritance relationships among views, and Section 5.5 revisits the notions of schema and database, in presence of views. Finally, Section 5.6 discusses object accesses from different contexts, and Section 5.7 concludes the chapter.

5.1 View Mechanisms for Object Databases

The object-oriented paradigm has been recognized as a sound basis for a new generation of database systems. Indeed, its ability to model complex objects, together with its modularity and extensibility properties, overcomes most of the problems arising in the use of the simple relational model [42]. A general agreement exists on the fact that object-oriented database features should meet as much as possible functionalities of the relational database systems [19, 25, 111]. Over the last years, a considerable research effort has been devoted to explore how the paradigm shift from the relational data model to an object-oriented model affects notions such as query languages, authorization, indexing, schema evolution and concurrency control.

An important relational functionality is represented by views. In the relational model, a view is a virtual (i.e. not physically stored) relation, defined by a query on one or more stored relations. As relational languages are closed (i.e. the result of a query expressed in a relational language is a relation), the relation returned by such a query represents the view content. Thus, relational views can be used in (almost) any context in which a relation may appear. Moreover, authorizations may be granted and revoked on views as on ordinary relations. At the same time, views can be used to define external schemas, in that virtual relations are generated by combining base relations. Views are an integral component of the ANSI three-level schema architecture standard that has driven the construction and use of relational database systems. Such a schema architecture consists of the storage schema describing the storage structures for a database, the conceptual schema describing the logical model of the database, and the external schema describing the derived views of the conceptual schema for particular users or group of users.

The definition of a view mechanism has been recognized to be a fundamental aspect also for the practical development of object-oriented applications. In this new context, views should be still used as shorthand in queries, should support the integration of heterogeneous databases, should be the basis for content-dependent authorization and, finally, should support the simulation of schema changes. Indeed, as it has been recognized by several researchers [28, 47, 147], views allow to dynamically modify a database schema yet retaining its older versions, a very important capability for advanced applications [115]. Moreover, a foundation for external schemas will complete the development of a three-level schema architecture for object-oriented database systems, comparable to that for relational database systems.

Unfortunately, the definition of an object-oriented view mechanism does not come for free from the relational approach. The main problems in the definition of an object-oriented view mechanism can be summarized as follows:

1. The object-oriented model is far more complex than the relational one. Whereas a relational schema consists of a set of independent relations, an object-oriented schema is a class hierarchy, where classes are connected by inheritance relationships. A view model should provide an answer to the question: *How are views integrated in the existing class hierarchy?*

2. Objects have an identity. A view, at the data level, should be a class. But what are view instances? Are they values, or existing objects, or newly generated objects?

Note that this problem does not arise in the relational model, where no strong identity concept, such as the object identifier, is modeled.

Several approaches have been proposed to model views in object-oriented database systems [2, 28, 150, 155, 162] (see [135] for a survey). They address the previous issues according to different approaches. In general, besides being based upon different data models and exploiting different query languages to express view populations, the proposals differ for: the set of functionalities supported by the view mechanism (shorthand in queries, external schema definition, schema evolution, authorization, etc.); the approach with respect to the placement of views in the schema; the properties assigned to views objects (i.e., whether or not persistent object identifiers are provided); the update operations allowed on views. An optimal solution to the view mechanism does not exists. Rather, some *good* solutions can be defined for each class of chosen functionalities. Thus, a view mechanism defined to support schema evolution may be different from a view mechanism defined for only using views as shorthand in queries.

In this section, we discuss the main dimensions in the design of a view mechanism. For each dimension, we contrast our choice with the ones made by most relevant view models in the literature.

Besides choosing a reference data model and a query language, the main design choices concern:

- how views are inserted in the database schema;
- whether views are only populated with base objects (object-preserving views), or it is possible to populate a view by creating new objects (object-generating views).

The choices to be taken with respect to those dimensions are strongly influenced by the functionalities to be supported by the view mechanism. In what follows, we first present the goals of the proposed model, and analyze their implications on the view mechanism. Next, we analyze the two main dimensions, showing how the chosen objectives affect our choices. Subsection 5.1.4 concludes the discussion by summarizing in Table 5.1 the comparison among our model and other view models proposed in the literature.

5.1.1 View Functionalities

Whereas relational views have been used for external schema definition, data protection (content-based authorizations) and shorthand for queries, object-oriented views can be exploited also for other kinds of functionality, such as supporting schema evolution and integrating heterogeneous databases. The use of views for integrating heterogeneous database schemas has been considered in [103, 117]: a view definition integrates semantically equivalent classes belonging to different schemas. The use of views to simulate schema evolution, allowing the users to experiment with schema changes without affecting other users, was first proposed in [28]. Views can support the implementation of a schema versioning mechanism, such that any object stored in the database can be accessed and modified from any schema version including a view of the object class. Recently, other view models have considered that use of views [47, 117, 147]. The properties of a view mechanism determines which schema modifications can be simulated. For instance, the models presented in [47, 147] allow to simulate the addition of a new attribute to a class because they consider views which can include non-derived additional attributes. However, the view model introduced in [117] supports a very limited number of changes because it does not support neither the generation of persistent identifiers for view instances nor views augmenting class definitions. Views allow the definition of external schemas with the meaning proposed by the ANSI three-level architecture: each external schema consists of a set of views and base classes specifying how a user perceives the database. Some of the proposed object-oriented view models [121, 155, 162] have considered external schemas. In those models view definitions are part of a schema view.

The design of our view model has been influenced by two main objectives: (*i*) it must be sufficiently powerful for supporting all the kinds of schema changes included in well-known taxonomies [24, 142], and (*ii*) it must support the definition of external schemas whose properties are identical to those of a base schema, so that it is possible to develop object-oriented applications on an external schema. The second goal implies that classes and views must have the same nature. In order to satisfy these requirements we have introduced the concept of *schema view*. A schema view encapsulates a set of related view definitions, with a well-defined purpose, such as to define an external schema or to perform some schema changes. The database administrator initially defines a base schema on which the object database is created. A schema view can be derived from either the base schema or from another schema view.

5.1.2 View Placement

A relational schema consists of a set of relations, while an object-oriented schema consists of a class hierarchy, being classes connected by inheritance relationships. Therefore, an object-oriented view model must deal with the problem of inserting views in the class hierarchy. We refer to this problem as *view placement problem*. Three kinds of solutions have been proposed to solve it:

1. Views are automatically positioned in the class hierarchy by an integration algorithm [2, 151, 155].

2. The user explicitly specifies the position of the view in the class hierarchy [107].

3. The view and class hierarchies are kept separated, that is, a class can inherit only from classes and a view can inherit only from views. By contrast, a *view derivation relationship* relates a base class and a view; the semantic of this relationship is "the view is derived from the base class" [28].

Bearing in mind that views must behave like classes, the integration of views and classes in a unique schema may appear the most appropriate solution. However, solution 1 above has two problems: first, the problem of integrating a view in an existing schema is in general undecidable [25, 155]; second, the hierarchy may become very large because of many intermediate classes that are not semantically meaningful. With the second solution the mix of classes and views in the same hierarchy may cause confusion in the user. Moreover, checks must be made by the system to ensure that the specified position is coherent with the definition of the view. Note, moreover, that when one wants to support the mingling of classes and views in a single inheritance hierarchy the placement of a view in the inheritance hierarchy must be made by considering the two aspects of a view definition: the signature (list of attributes and methods, along with their domains) and the extent of the view (the set of instances that will be materialized when the query part of the view is evaluated). It may happen that the type of a view is a supertype of the type of the class from which it is derived but its extent is a subset of the extent of that class. Consider as an example a view extracting (that is, selecting) some objects from a class and projecting out some of their attributes. In a well-formed inheritance hierarchy, subtyping and set containment between class extents go together, since the extent of a class is defined as a subset of the extent of its superclass(es). Finally, note that none of the models taking the approach of inserting views in the class hierarchy supports object-generating views[1].

Therefore, we think that the more adequate approach is the separation of classes and views in different hierarchies, extending the object-oriented schema with the *view derivation relationship*, because it leads to a schema easier to understand. A view can be derived from other views, but in any case it will always be connected to base classes. With respect to the view derivation relationship, the term *root class* always refers to classes or views from which a view is derived. Let v be a view derived from the root class c, then the view derivation relationship denotes that v *is a view of* c, or, equivalently, that c *is a root class of* v.

In order to satisfy the above mentioned goals, it is also necessary to introduce a *view inheritance relationship*, similar to the class inheritance relationship existing in the base schema. The inheritance relationship is orthogonal to the view derivation relationship.

[1] An exception is represented by [2] but, as we will see, that model does not support truly object-generating views in that a query returns a set of tuples, that are converted to new objects outside the query language.

The view inheritance relationship organizes views in an ISA hierarchy similar to the class hierarchy. An important difference between class and view inheritance is the fact that unlike a class, a view is not explicitly populated; rather its population is derived from the population of its root classes by the view query. Thus, to ensure that the instances of a view are a subset of the instances of its superviews, we impose two restrictions: (*i*) a view v_1 can be declared subview of a view v_2 if and only if the root class of v_1 is a direct or indirect subclass of the root class of v_2, and (*ii*) the query of a view must be stronger than the view queries of its superviews. This topic is dealt with in Section 5.4. Like the second solution, the user is responsible for the declarations of inheritance relationship, but the system must ensure that the placement of a view does not violate the semantics of a well-formed view inheritance hierarchy. Recently, two view models have been proposed which also solve the view placement problem by combining the view derivation and view inheritance relationships: the extension presented in [161, 162] to the view model described in [2] for the O_2 system, and the view mechanism for the UniSQL system [117].

5.1.3 Object-generating vs Object-preserving Views

If views are to be used as classes, it is essential that their instances are objects, that is, that they are provided with persistent identifiers. In most situations there is indeed the need of referencing view instances. Two distinct kinds of views can however be identified:

- *object-preserving views*: they are views that only extract objects from existing classes; the instances of these views can be identified by the identifiers of the extracted base objects.
- *object-generating views*: they are views creating new objects; the instances of these views must be identified by newly generated object identifiers.

Most of the approaches [138, 154, 156, 163] only consider object-preserving views. Thus, views only provide different views of existing objects. This approach is particularly useful for supporting objects with multiple interfaces and context-dependent behavior. In this sense, object-preserving views are similar to *roles* [12, 94, 148]. However, this kind of views is not powerful enough for supporting all kinds of database reorganization. In [146] a model is described whose views are object-preserving, but with the capability of including new non-derived attributes from existing data. By contrast, object-generating views are proposed in [28] for supporting schema evolution, so that the evaluation of a query always returns new objects. Object-generating views are also considered in [107], where a query can include an *oid function*: a partial function indicating that the query returns a set of objects whose identifiers are generated by applying the function on the object identifiers assigned to its argument variables.

The approach proposed in [2] supports object-preserving views as well as value-generating views. Thus, two kinds of views are considered: *virtual classes*, populated by objects selected from already existing classes, and *imaginary classes*, populated by tuples for which new oids are generated. Virtual classes are defined by specialization or generalization of base classes, while imaginary classes are declared by queries that return sets of values. In this approach, an imaginary class C is populated by a query that returns a set of tuples. To each tuple t an oid denoted as $C(t)$ is assigned, using a function associated with the class that is applied to the tuple. Thus, in this approach queries create relations rather than creating sets of objects. Therefore, queries cannot be used to define views, since it is necessary to convert tuples into objects outside the query language. This object "creation" is performed at each view evaluation, thus the problem arises of assigning the same oid to the same tuple at each evaluation.

In our model, we consider both object-preserving and object-generating views. Thus, when the view query evaluation involves the creation of view instances in order to populate the view class (e.g. a join operation), new oids are generated. By contrast, when views are populated by extracting existing objects from a class, possibly, modifying their structure and behavior, (e.g. a selection or projection operation) the view instances preserve the identifiers of base objects, instead of generating new objects. The support of object-preserving views requires that an object instance of a base class can also be an instance of all those views whose query is satisfied by the object. Thus, the use of the same identifier for denoting an object which is instance of both a class and a view implies that references to this object can only be solved by taking into account the *context* of the reference. However, this is already the case in our model, because it supports objects belonging to multiple most specific classes. We will elaborate further on the problem of solving object references in Section 5.6.

Together with views generating objects, we also allow a user to specify that the instances of a view are *not* objects, and thus are not provided with persistent identifiers. Therefore persistent identifiers are generated only when needed, that is, only when the view must be used as a class. Views whose instances are values rather than objects (which we refer to as *set-tuple views*) are useful to include relations in the object data model , thus providing a form of downward compatibility with respect to the relational model. However, views, whose instances are not provided with persistent identifiers, can only be used as shorthand in queries and cannot have additional attributes. As a default, we assume the view query returns a set of persistent objects, so that the view has an extension defined as the set of the oids of objects that belong to the view (both for object-generating and object-preserving views). Whereas the user needs to specify whether a view generates persistent identifiers or not, the system will be able to check whether a view preserves or generates objects, by analyzing the view query.

5.1.4 Discussion

	[28]	[117]	[150]	[155]	[162]	Our model
data model	general	UniSQL	MultiView	COCOON	O_2	
query language	object-oriented predicate calculus	Object SQL	Object algebra	Object algebra	O_2	deductive rules
external schemas	NO	NO	YES	NO	YES	YES
schema evolution	YES	L.F.	L.F.	L.F.	L.F.	YES
view integration problem	separate view hierarchy	separate view hierarchy	views inserted in class hierarchy	views inserted in inserted in class hierarchy[a]	separate view hierarchy	separate view hierarchy
object-preserving/ generating	generating	no persistent identifiers	preserving	preserving	both[b]	both
augment class definition	YES	NO	YES	NO	NO	YES

Legenda:

L.F. limited form.

[a] For operator individuals, it is specified how the view is inserted in the class hierarchy, but the classification of the views resulting of composite queries is not addressed.

[b] Actually, a query may return a set of tuples that are converted to new objects outside the query language.

Table 5.1: Comparison of the proposed view model with other view models for object databases

Table 5.1 compares the most relevant proposals for object-oriented view mechanisms, by taking into account the dimensions dealt with throughout the section. The table shows that our view model aims at supporting both external schemas and schema evolution. The main features of our approach thus strictly depend on those choices. In particular:

- In defining a view, the user can choose among *object-preserving views*, *object-generating views* or *set-tuple views*, depending on whether the view is populated with objects extracted from an existing class, or the view must be instantiated with new objects, or the view instances do not require persistent object identifiers. Set-tuple views allow to support relations in the object data model, thus meeting the requirements of relational object models such as UniSQL [112] or Matisse [7].

- Following [28], we do not integrate views in the class inheritance hierarchy. Rather, views are organized in a separate hierarchy: they are related by a view inheritance relationship which is analogous to the inheritance relationship on classes. Moreover, the schema is extended with a new relationship, called *view derivation* relationship, connecting a view with the classes from which it is derived. The view derivation hierarchy is orthogonal to the class inheritance hierarchy.
- Two view levels are devised: views and schema views. Views are virtual classes and can be used in any context in which classes can be used; schema views provide the capability of restructuring a schema so that it meets the need of specific applications. A schema view is a virtual schema, that is, a schema which consists of views rather than of classes.

In the table, we have also considered whether the model allows a view to add new non-derived attributes. A view may indeed hide, modify and add features to those of the classes (or views) it is derived from. All models allow to hide features, as well as to add new methods and to change the implementation of methods. The addition of non-derived attributes has only been considered in some models supporting schema evolution. This property, obviously, increases the number of possible schema changes, but on the other hand it makes the implementation more difficult. Our model has this augmentation capability, thus view instances can have an additional storage for new attributes. Of course, other actions (e.g. hide attributes, hide/add/modify methods) are also possible.

Finally, let us mention that our view mechanism is comparable to (or includes) existing ones [2, 28, 150, 155, 162] for what concerns the supported features; however, our model is the only one for which a formal definition is given. Because of the similarity of features, our definition can be adapted to other object-oriented view mechanisms and, thus, has value beyond the particular view mechanism we propose.

5.2 Views

In our model, a view is defined as a query on one or more base classes whose result is a new class, as a natural adaptation of relational views. A view is identified by a name, which is the proposed identifier for the view. A view definition is thus similar to a class definition (name, list of attributes, list of methods, list of constraints) except that it includes a query on one or more base classes determining the view population. A view can be used like a class. For example, the domain of an attribute, parameter or variable of a view can be another view.

Throughout the chapter, we use as a running example the base schema presented in Figure 5.1. The classes of the `FacultyLibrary` schema only contain instance attributes, the

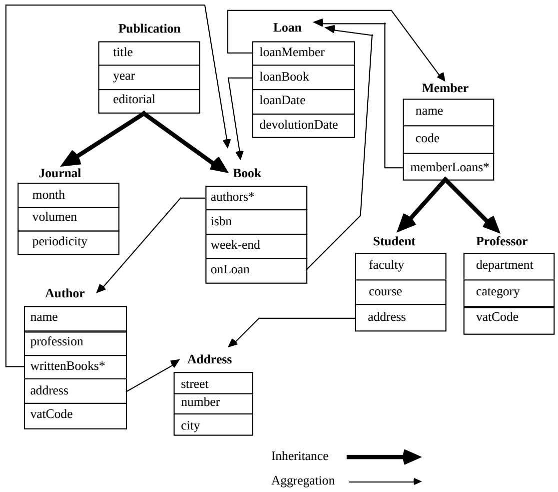

Figure 5.1: `FacultyLibrary` example schema.

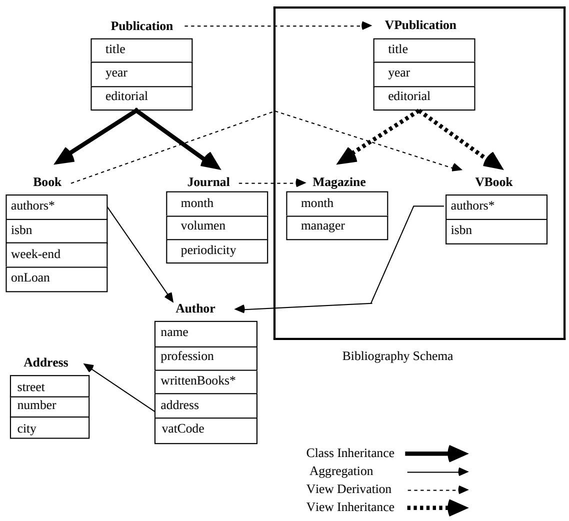

Figure 5.2: Schema including views derived from the `FacultyLibrary` schema.

other features are excluded for the sake of simplicity. The symbol '*' denotes that an instance attribute is multi-valued. Figure 5.2 shows a schema view named `Bibliography` derived from the `FacultyLibrary` base schema of our running example. This schema view illustrates how *view derivation* and *view inheritance* relationships can be used when views are derived from a given schema, in order to create a new schema. The `Bibliography` schema contains the views `Vpublication`, `Magazine` and `Vbook` which have been derived from the classes `Publication`, `Journal` and `Book`, respectively. In this example, the root classes are base classes because the view schema is derived from the base schema. The view `Magazine` has an additional attribute, `manager`, and hides the attributes `periodicity` and `volumen`; the view `Vbook` has no additional attributes and hides `week-end` and `onLoan`, while the view `Vpublication` imports all the attributes from the class `Publication` and does not add new attributes. For the remaining features, we suppose that the views import all features from their root classes and do not augment class definitions. The views `Magazine` and `Vbook` have been declared subviews of the view `Vpublication`. Note that the view `Vpublication` is the identity view[2] of the class `Publication`, being intended to have a hierarchy of publications in the schema view. As remarked in [155], schema views must satisfy some consistency constraints concerning the schema closure. For instance, because the domain of the attribute `authors*` is the class `Author`, the closure of the schema view `Bibliography` must contain the identity views of classes `Author` and `Address`. The closure of schema views will be discussed in Section 5.5.

As for classes, we consider two components in a view: *signature* and *implementation*, the following subsections discuss these components.

In the following, let $\mathcal{VI}$ denote a set of view identifiers, and $\mathcal{CVI}$ denote the set of class and view identifiers, thus, $\mathcal{CVI} = \mathcal{VI} \cup \mathcal{CI}$.

Definition 5.1 *(View). A view V is a tuple*

$$V = (id, sign, impl, state)$$

where:

- *$id \in \mathcal{VI}$ is the view identifier;*
- *sign is the view signature (see Definition 5.2);*
- *impl is the view implementation (see Definition 5.3);*

[2]As we will see in Section 2.7, given a class c, its identity view is a view v having c as root class, such that c and v are equivalent, both at the extensional and at the intensional level.

- *state* $\in \mathcal{V}$ *is a value containing the values for the class attributes of the view plus two set values, containing all the objects belonging to* V *as members and as instances, respectively; it is a record value of the form*

  $$(a_1 : v_1, \ldots, a_n : v_n, extent : E, proper\text{-}extent : PE)$$

  where $a_1, \ldots, a_n$ *are the attribute names of the c-attributes of* V*;* E *and* PE *are the sets of identifiers of objects belonging to view* V*.* □

The attributes *extent* and *proper_extent* in the view state denote respectively the set of all the oids of objects members of the view and the oids of objects instances of the view. Therefore the *proper_extent* field of the view state contains the set of objects belonging to the view and not belonging to any of its subview in the view inheritance hierarchy.

The *definitional component* of a view $V = (id, sign, impl, state)$ is $(id, sign, impl)$, that is, its state-independent components.

5.2.1 View Signature

Definition 5.2 *(View Signature). A view signature is a tuple*

$$(struct, beh, constr, mc, q)$$

where:

- *struct, beh, const are the information about the structure, the behavior, and the constraint of the instances of the view, and they are as for classes (see Definition 2.12);*
- *mc is the identifier of a virtual metaclass corresponding to the view; this identifier can be any name not used as name of other class or metaclass;*
- *q is the view-query, that is, a set of deductive rules specifying the view population.* □

Example 5.1 *Suppose we wish to define a view* `ProfAuthor` *containing all professors which are authors of some book published by the* `University Editorial` *and available in the library. The view signature corresponding to that view is as follows. Let* { (*name, string, ext*), (*vatcode, integer, ext*) } *be included in the struct component of the class identified by* `Author`*. Then:*

$$
\begin{aligned}
struct &= \{\ (name, string, ext), (vatcode, integer, ext),\\
&\quad\ (bookTitles, set\text{-}of(string), der),\\
&\quad\ (city, string, der)\}\\
beh &= \{(changecity, string \rightarrow ProfAuthor)\}\\
constr &= \emptyset\\
mc &= \texttt{MProfAuthor}\\
q &= \texttt{ProfAuthor(X)} \leftarrow \texttt{Author(Y), Professor(Z),}\\
&\qquad \texttt{Book(W), Y.vatCode = Z.vatCode,}\\
&\qquad \texttt{W in Y.writtenBooks,}\\
&\qquad \texttt{W.editorial = `University Editorial'}
\end{aligned}
$$

$\diamond$

Example 5.2 *Suppose we wish to consider a class representing magazines whose periodicity is weekly. We can derive the* `Magazine` *view from the* `Journal` *class, populated with all the objects from* `Journal` *class such that* `periodicity = `weekly'`. *The view has an additional attribute representing the magazine editor while the* `volumen` *and* `periodicity` *attributes are hidden. The view signature of the corresponding view* `Magazine` *is as follows.*

$$
\begin{aligned}
struct &= \{\ (title, string, ext), (year, integer, ext),\\
&\quad\ (editorial, string, ext), (month, string, ext),\\
&\quad\ (manager, string, ext)\}\\
beh &= \{(changeManager, string \rightarrow Magazine)\}\\
constr &= \emptyset\\
mc &= \texttt{MMagazine}\\
q &= \texttt{Magazine(X)} \leftarrow \texttt{Journal(X), X.periodicity = `weekly'}
\end{aligned}
$$

$\diamond$

The identifier of a view V denotes the object type corresponding to V. Such object type is the type of the identifiers of the objects instances of the view. A value type is moreover implicitly associated with each view, representing the type of values that constitute the state of the view instances. If the *struct* component of a view V is the set $\{(a_1, T_1, at_1), \ldots, (a_n, T_n, at_n)\}$, each object instance of V must have as state a value of (record) type $record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$. This type, which describes the structure of the objects instances of the view, is the *structural type* of the view, and it is denoted by $stype(v)$, being $v \in \mathcal{VI}$ the view identifier of V ($V.id = v$). Two components can be distinguished in the state of a view instance: an additional component and an imported component. Therefore, given the view structural type $stype(v) = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$, it is possible to partition this record type in two record types $stype\_imp(v)$ and $stype\_add(v)$. Therefore, $stype\_add(v)$ denotes the additional structural type while $stype\_imp(v)$ denotes the imported structural type. We distinguish these two components in the structural component of a view, because the additional component is related to additional attributes that must be stored for view instances, whereas the imported component is related to attributes imported from root classes. The values of the imported component are, therefore, retrieved (or computed) starting from values already stored in the database. The system does not allocate space for imported attributes, while it does for additional ones.

The set of the classes[3] from which a view is derived can be represented as a function $defined\_on : \mathcal{VI} \rightarrow 2^{\mathcal{CVI}}$. For example, referring to Example 5.1, $defined\_on(\texttt{ProfAuthor}) = \{\texttt{Professor}, \texttt{Author}\}$. Note that, since views can be defined in terms of other views, the function $defined\_on$ returns a set containing base classes as well as views. It is however possible to determine the base classes on which a view is defined, by recursively applying function $defined\_on$. For that purpose the function $defined\_on^* : \mathcal{VI} \rightarrow 2^{\mathcal{CI}}$, defined as follows, can be used:

$$defined\_on^*(v) = \begin{cases} defined\_on(v) & \text{if } defined\_on(v) \subseteq \mathcal{CI} \\ \\ \{c_1, \ldots, c_k\} \cup defined\_on^*(v_1) & \text{if } defined\_on(v) = \{c_1, \ldots, c_k, v_1, \ldots, v_h\}, \\ \cup \ldots \cup defined\_on^*(v_h) & \{c_1, \ldots, c_k\} \subseteq \mathcal{CI} \text{ and } \{v_1, \ldots, v_h\} \subseteq \mathcal{VI} \end{cases}$$

We represent through a boolean function $oid : \mathcal{VI} \rightarrow Bool$ whether or not the view instances are provided with persistent identifiers. The default value is $true$, corresponding to the case in which view instances are objects. The $false$ value is used for views whose instances are not provided with persistent identifiers, being thus values rather than objects (set-tuple views). This can be specified by the user through an appropriate clause in the view definition statement.

Moreover, we represent through a boolean function $new\_oid : \mathcal{VI} \rightarrow Bool$ whether the view is object-generating or object-preserving. That is, $new\_oid(v) = true$ for object-generating views, while it is $false$ for object-preserving ones.

[3] We refer with the term root classes to the views and classes a certain view has been defined on. Thus, we sometimes improperly denote as classes a set of classes and views. Whenever confusion may arise, we adopt the term base classes to distinguish classes from views.

5.2.1.1 View Query

Whereas most view models express the view query by an object algebra or calculus, we use deductive rules. The view-query contains one or more deductive rules specifying the view population. The head of these rules is a class formula on the name of the view whose population is being defined and the body is an arbitrary formula on instances of the root classes (see Definition 3.3). We remark that only side-effect free methods are allowed in queries. From now on, the *view-query* term will denote the collection of rules that define the view population. The view-query defines the extension of the view, while the structure of each instance belonging to this extension is based on imported and additional attributes.

Example 5.3 *The following query defines the population of a view* `CsStudent` *derived from the class* `Student` *of the schema* `FacultyLibrary`*, retrieving the students whose faculty is* `Computer Science` *and that have borrowed at least a book.*

```
CsStudent(X) ← Student(X), Book(Z),
               X.faculty = `Computer Science´,
               Z in X.memberLoans
```

◇

If the variable of the class formula in the head of the rule appears in (a class formula of) the rule body, then the view is an object-preserving view (examples of object-preserving views are the views of Example 5.2 and of Example 5.3), otherwise the view is an object-generating view (an example of object-generating view is the view of Example 5.1).

For object-preserving views, if the class formula in the rule body which contains the variable in the head of the rule is on the root class c, then the view instances are objects extracted from c. Thus, all the members of c that satisfy the body of the rule are instances of the view. For example, instances of the `CsStudent` view of Example 5.3 are objects belonging to class `Student`. By contrast, the query of an object-generating view returns a set of base object tuples. For each tuple in this set a new object is generated and added to the view extension, and the correspondence between the new object identifier and the base object identifier is stored in a persistent table, named DERIVED BY. It is sometimes useful, in the definition of the implementation for derived imported attributes, to explicitly refer to the base objects a certain view object has been derived from. We thus extend the syntax of deductive rules with a special atomic formula built using the ternary predicate `derived-by`, whose first argument is the identifier of a view instance, second argument is a class identifier and third argument is an object identifier. The third argument is bound to the base object, instance of the specified class, from which the specified view object has been derived. This predicate is simply a mean to refer to the DERIVED BY table from the body of a deductive rule.

If the instances of a view are newly generated objects, the view-query of such a view contains a variable in the head which is not contained in any atomic formula in the body. In such cases, we may think that an atomic formula `next-oid(X)` is automatically added to the rule body, being `X` the variable appearing as argument of the class formula in the rule head, denoting the newly generated object. In Example 5.1 the body of the view-query will include `next-oid(X)`, being the rule head the class formula `ProfAuthor(X)`. The meaning of atomic formulas built with the `next-oid` predicate is the following:

1. a new object identifier must be generated for each successful evaluation of the formula in the rule body;

2. the generated identifier must be added to the view extension;

3. for each new generated oid, an entry containing the generated oid and the oids of base objects from which it has been derived, must be stored in the DERIVED BY table.

Note that a view-query may consist of several deductive rules. Thus, our view language is able to express views defined by queries on a union of classes using alternative predicates [28], as shown by the following example. The example also shows that it is possible to define an object-preserving view by extracting objects from two or more root classes.

Example 5.4 *The following query retrieves all the members that have borrowed at least one book, such that, if they are student, their faculty is* `Computer Science` *and if they are professors, their category is* `full time`*.*

```
LoanMembers(X) ← Student(X), Book(Z),
                 Z in X.memberLoans,
                 X.faculty = `Computer Science´
LoanMembers(X) ← Professor(X), Book(Z),
                 Z in X.memberLoans,
                 X.category = `full time´
```

◇

Finally, we remark that at the time being we do not consider recursive views. A view v can be defined in terms of another view v' provided that v' is not defined, directly or indirectly, in terms of v. Thus, each view can be ultimately seen as defined in terms of base classes.

5.2.1.2 Imported and Additional Features

Besides specifying the root classes and the query defining the view, the view signature also specifies information on view features. We distinguish between *imported* and *additional* view features: imported features are obtained from one of the root classes, while additional features are explicitly defined for the view. When considering attributes, however, a view can have some attributes not belonging to the signature of any root class, but whose value can be derived (that is, computed) starting from the values of some attributes in the root classes. We consider this kind of attribute as imported rather than as additional, to remark that for additional attributes new storage space must be allocated and a value must be provided for each view object, since all additional attributes have a null value upon view materialization.

We do not consider here the view definition language (which is however presented in [99]). We only mention that in the view definition statement a clause allows to specify the lists of features imported from root classes. In the case of imported attributes, a clause of the view definition statement specifies which attributes among the ones of the base objects retrieved by the query are part of the view instances. For each kind of features (attributes, methods, constraints, both at the instance and at the class level) the associated list contains one or more items specifying which features are imported. There are different options to specify the imported features: listing the features to be imported, specifying that all the features (of a root class) are imported, specifying that a feature is hidden, or specifying that a feature is renamed in the view. The option of specifying which features (of a given root classes) are hidden is useful when the number of imported features is greater than the number of hidden ones; another important (semantic) advantage of allowing the specification of hidden features rather than requiring the specification of imported ones, is that the view may change if the root class changes, e.g. if new attributes are added to the root class they are added to the view in the former case, whereas they are not added in the latter. In the case of imported attributes, there are two additional formats for introducing derived attributes not corresponding to any attribute of the root classes, but whose value can be computed from attributes in the root classes by means of deductive rules. Those attributes are part of the state of the instances of the view but do not belong to the state of any base object the view object is derived by. A first option is to indicate that a view attribute is derived; its implementation must be given in the view implementation. The derived imported attribute `bookTitles` in the view of Example 5.1 could be implemented by the following deductive rule:

```
X in self.bookTitles ← Author(Y), Book(W),
     derived-by(self, Author, Y),
     W.editorial = `University Editorial´,
     Y in W.authors
```

The above option supports the redefinition of the domain of an attribute imported from a root class. It may be useful for restricting the domain of an attribute to a subtype of the current attribute type, or for changing it to a view derived from the class which is the attribute domain.

A second option is to specify that a view attribute corresponds to a nested attribute of one of the root classes of the considered view. The imported attribute `city` of view `ProfAuthor` of example 5.1 can be defined through the path expression `Author.address.city`. This expression specifies that the view `ProfAuthor` has an attribute named `city` whose value, for each object o belonging to the view, is the value of `address.city` in the base object from root class `Author` by which o is derived[4].

The various formats for importing features in view specification are specified in Appendix B.

5.2.2 View Implementation

The view implementation is exactly like a class implementation, including the set of rules that specify the implementation of the imported and additional derived attributes, methods and constraints.

The view implementation is expressed by making use of deductive rules, as defined in Chapter 3, extended as follows:

- the rules specifying the view population may contain a variable in the head that do not appear in any atomic formulas in the bodies; this is the format for specifying object-generating views and it is handled, as seen in Section 5.2.1.1, by inserting in the rule body a special atomic formula on the `next-oid` predicate;
- the rules specifying the implementation for derived attributes may contain, in their body, atomic formulas on the ternary predicate `derived-by`, which allows to refer to the base object(s) from which the view object at hand has been derived.

More precisely, a view implementation consists of three sets of rules:

[4]Any attribute expressed with a path expression can be as well declared as derived and its implementation specified by deductive rules. Path expressions are therefore a shorthand of deductive rules specifying a navigation through an aggregation hierarchy. For example the declaration `city: Author.address.city` could be expressed by the declaration `city: string derived`, being its implementation expressed by the following deductive rule:

```
self.city = X ← Author(Y), Y.address.city = X
                derived-by(self, Author, Y)
```

1. a set of deductive rules specifying the implementations of the view derived attributes; an implementation must be provided for

   - each additional derived attribute; in this case the implementation is specified in the implementation part of the view definition;
   - each imported attribute declared as derived in the signature part of the view definition; in this case the implementation is specified in the implementation part of the view definition;
   - each imported attribute which is derived in the root class from which it is taken; in this case the implementation is the same as the one in the (implicitly or explicitly) referred class;
   - each imported attribute declared as $\mathtt{a : c.a_1. \cdots .a_n}$ in the signature part of the view definition; in this case the implementation consists of

     (a) the rule
     $self.a = X \leftarrow c(Y), Y = self,$
     $X = Y.a_1. \cdots .a_n$
     if $new\_oid(v) = false$;

     (b) the rule
     $self.a = X \leftarrow c(Y), derived\_by(self, c, Y),$
     $X = Y.a_1. \cdots .a_n$
     if $new\_oid(v) = true$;

2. a set of deductive rules specifying the constraints on the view population; this set consists of the rules specified in the view class implementation for additional constraints and the rules in the intended root class implementation for imported constraints;

3. a set of update rules specifying the implementation of the view operations; this set consists of the rules specified in the view class implementation for additional operations and the rules in the intended root class implementation for imported operations.

Definition 5.3 *(View Implementation). Given a view V whose signature $V.sign = (struct, beh, constr, mc, q)$, an implementation for V consists of a set of deductive rules, specifying*

- *an attribute implementation for each derived attribute in struct;*
- *a constraint implementation for each constraint in constr;*

plus a set of update rules specifying an operation implementation for each operation in beh.

□

Example 5.5 *The implementation of the view of Example 5.1 consists of the following rules.*

```
X in self.bookTitles ← Author(Y), Book(W),
     derived-by(self, Author, Y),
     W.editorial = `University Editorial´,
     Y in W.authors

self.city = X ← Author(Y),
     derived-by(self, Author, Y),
     Y.address.city = X

changecity(City):
      → modify(ProfAuthor.city, Self, City).
```

◇

5.3 View Instances

In our approach, the evaluation of a view definition results in a view whose instances have a structure defined by function *stype* introduced above. Thus, a view is a class and, if it is not specified to be a set-tuple view, its instances are objects referred through immutable identifiers. The extent of a view may consist of objects extracted from an existing class or view (for the views v such that $new\_oid(v) = false$), or it may consist of newly generated objects (for the views v such that $new\_oid(v) = true$). In the first case, the extracted object is, obviously, an instance of the class from which it has been extracted and it is also an instance of the view. Thus, the object belongs to multiple most specific classes. We have addressed the problem of objects belonging to multiple most specific classes in Chapter 2 where only base classes are considered. Here, we extend that approach by considering also views. Thus, an object may belong to several most specific classes and to a set of views derived from them. The notion of object can be formalized as follows.

Definition 5.4 *(Object). An object is a tuple*

$$o = (i, v, VS)$$

where:

- $i \in \mathcal{OI}$ *is the identifier of* o;

- *$v \in \mathcal{V}$ is a value, called state of o;*
- *$VS \subseteq \mathcal{CVI}$ is the set of most specific classes and views to which o belongs.* □

For object-generating views, a new persistent oid is generated for each view instance, in the same way as base object identifiers are generated upon object creation. The objects generated by the view evaluation have only a virtual nature (though part of their state, that is, the values for additional attributes, is stored) and are referred to as *view objects*. They are objects according to Definition 5.4, though they do not belong to any base class. We may thus partition the set $\mathcal{OI}$ of object identifiers in two sets: $\mathcal{BOI}$, the set of base object identifiers; and $\mathcal{VOI}$, the set of view object identifiers, that is, the set of identifiers corresponding to view objects, generated upon view materialization. An object o has an identifier belonging to $\mathcal{VOI}$ if it has only a virtual nature, that is, if it is not an instance of any base classes.

Definition 5.5 *(View Object). A view object is an object o defined according to Definition 5.4 such that*

- *$o.i \in \mathcal{VOI}$, and*
- *$o.VS \subseteq \mathcal{VI}$.* □

We remark that $o.i \in \mathcal{VOI} \Rightarrow o.VS \subseteq \mathcal{VI}$, that is, a view object belongs only to views. By contrast, $o.i \in \mathcal{BOI} \Rightarrow \exists c \in o.VS$ such that $c \in \mathcal{CI}$, that is, a base object belongs to at least one base class.

The function $derived\_by : \mathcal{VOI} \rightarrow 2^{\mathcal{BOI}}$, for each view object identifier $i \in \mathcal{VOI}$, returns the set of identifiers of base objects from which the view object identified by i has been derived. This function is defined by recursively replacing each view object identifier i with the set of identifiers appearing in the columns of the Derived By table row, whose first column contains i. This process ends when the set contains only base object identifiers. If, given a view identifier v, $defined\_on^*(v) = \{c_1, \ldots, c_n\}$ then, for each view object identifier i such that $i \in [\![v]\!]$, $derived\_by(i) = \{i_1, \ldots, i_n\}$ and for each $j, 1 \leq j \leq n$, a class c_k exists, $1 \leq k \leq n$, such that $i_j \in [\![c_k]\!]$.

5.3.1 Object State

The state of an object belonging to several most specific classes (and views), should be a record value having as fields the union of all the attributes in those classes. However, the sets of attributes in the object most specific classes and views may be non-disjoint. To handle this situation the notion of *source* of an attribute is introduce. If an attribute belongs to the intersection of the attribute sets of two classes and it has in both classes the same source, then the attribute is semantically unique, and thus the object must have a unique value for this attribute. If, by contrast, the attribute has different sources, then the two attributes in the two classes (views) have accidentally the same name, but represent different information, that must be kept in separate ways. Thus, the object may have two different values for the two attributes (a renaming policy is applied).

We now specify the notion of *source* of an attribute. For base classes[5], the source of an attribute a in a class c is the most general superclass of c in which the attribute a is defined. Thus, it is the class from which c has inherited attribute a. Two base classes have a common attribute with the same source if they inherit it from a common superclass. For views, the source of an attribute can be:

- either the view itself, if the attribute is neither inherited nor included (with the meaning specified below) from any root class;
- the source of the attribute in the most general superview from which the view has inherited the attribute, for inherited attributes;
- the source of the attribute in the root class from which the view has taken the attribute, for included attributes.

In the second case, the most general superview from which the view has inherited the attribute is determined as for base classes. In order to better explain the third case, consider the different formats for imported attributes presented in Appendix B. In cases a) and c) an attribute is included from a root class in the view, and the source is the class from which the attribute is taken (either the one explicitly specified or the only one containing that attribute). By contrast, in cases d), e) and f) the attribute is not actually included from the root class, and then its source is the view itself.

Operator $\uplus$ and the characterization of an object state in terms of the most specific classes to which it belongs presented in Chapter 2 generalize to objects also belonging to views.

A function $value : \mathcal{OI} \times \mathcal{CVI} \rightarrow \mathcal{V}$ is defined that, given an object o and a class (or view) c, if o is a member of c (that is, if $o.i \in [\![c]\!]$), returns the state of object o seen as an instance of class c. That function only returns the fields of the state value proper of the structural component of c. Let $o = (i, v, VS)$ be an object and v be the record value $record\text{-}of(a_1 : v_1, \ldots, a_n : v_n)$. Then $value(i, c)$ is determined as follows, for $j = 1 \ldots n$:

- if $a_j \in A(c)$ then $a_j : v_j$ is a field of $value(i, c)$, and

[5] The notion of source of an attribute for base classes only has been formally defined in Chapter 2.

- if $a_j = c\text{-}a'_j$ and $a'_j \in A(c)$, then $a'_j : v_j$ is a field of $value(i, c)$.

If an object $o = (i, v, VS)$ is a member of a class c, then $value(i, c)$ is uniquely determined and it is a legal value for the type $stype(c)$. This function is applied whenever we want to see an object as a view instance.

Since our model supports a strongly typed database language, each object reference is assigned a single context in each expression. Thus, for each object reference we are able to determine (starting from the types declared for variables and using schema information) the class or view the referenced object must be seen an instance of. Note that this allows us to model notions such as context dependent access restriction and context dependent behavior, typical of data models including roles.

We remark that we have denoted as state the collection of all the attribute values of an object. Not all these values are stored, since some of them can be computed. In particular, derived attributes are not stored.

Example 5.6 *Consider view* `ProfAuthor`*, specified in Example 5.1. Then:*

$$
\begin{array}{l}
(i_1,\ record\text{-}of(name : \text{`}JohnSmith', vatcode : 6432957, \\
\qquad city : \text{`}NewYork' \\
\qquad bookTitles : set\text{-}of(\text{`}\textit{Object-Oriented Databases'})), \\
\quad \{\ \texttt{ProfAuthor}\ \}\ )
\end{array}
$$

with $i_1 \in \mathcal{VOI}$, is an example of object instance of the (object-generating) view `ProfAuthor`*.*

Consider now view `Magazine`*, specified in Example 5.2. Then:*

$$
\begin{array}{l}
(i_2,\ record\text{-}of(title : \text{`}InternationalJournal', year : 1995, \\
\qquad\qquad editorial : \text{`}ACM', month : \text{`}April', \\
\qquad\qquad manager : \text{`}AlanFord', volume : 17, \\
\qquad\qquad periodicity : \text{`}weekly'), \\
\quad \{\ \texttt{Magazine, Journal}\ \}\ )
\end{array}
$$

with $i_2 \in \mathcal{BOI}$, is an example of an object instance of the object-preserving view `Magazine`*, and of the class* `Journal`*, whose signature C is such that $C.id =$* `Journal` *and*

$$
\begin{array}{rl}
C.sign.struct = \{ & (title, string, ext), (year, integer, ext), \\
& (editorial, string, ext), (month, string, ext), \\
& (volume, integer, ext), \\
& (periodicity, string, ext)\ \}.
\end{array}
$$

Moreover,

$$
\begin{aligned}
&value(i_2, \texttt{Magazine}) = \\
&\qquad record\text{-}of(title : \text{‘}InternationalJournal'\text{,} \\
&\qquad\qquad year : 1995, \\
&\qquad\qquad editorial : \text{‘}ACM', \\
&\qquad\qquad month : \text{‘}April', \\
&\qquad\qquad manager : \text{‘}AlanFord'), \\
&value(i_2, \texttt{Journal}) = \\
&\qquad record\text{-}of(title : \text{‘}InternationalJournal', \\
&\qquad\qquad year : 1995, \\
&\qquad\qquad editorial : \text{‘}ACM', \\
&\qquad\qquad month : \text{‘}April', \\
&\qquad\qquad volume : 17, \\
&\qquad\qquad periodicity : \text{‘}weekly') \\
&value(i_2, \texttt{Publication}) = value(i_2, \texttt{VPublication}) = \\
&\qquad record\text{-}of(\ title : \text{‘}InternationalJournal', \\
&\qquad\qquad year : 1995, \\
&\qquad\qquad editorial : \text{‘}ACM').
\end{aligned}
$$

$\diamond$

5.3.2 Object Consistency

Each view object must be a consistent instance of all the views to which it belongs, exactly as each object must be a consistent instance of all the classes and the views to which it belongs. The following definitions formalize the notions of consistency we consider.

In addition to structural consistency and constraint consistency, introduced in Chapter 2, for objects belonging to views view-query consistency is also required.

Definition 5.6 *(View-query Consistency). An object $o = (i, v, VS)$ is view-query consistent if for each view $\bar{v} \in VS$, a rule $H \leftarrow B$ in the view-query of $\bar{v}$ exists such that o meets B.* $\square$

Definition 5.7 *(Consistency). An object o is a consistent instance of a view $v \in \mathcal{VI}$ if o, besides being both a structural and constraint consistent instance of v, is also a view-query consistent instance of v.* $\square$

We have seen in Chapter 2 that a finite set of objects is consistent if it is closed under the refer to and composed of relationships. A finite set of objects OBJ containing also view objects is closed under the *derived_by* relation if the base objects from which a view object in OBJ is derived belong to OBJ, too, as stated by the following definition.

Definition 5.8 *(Closed Set of Objects). A (finite) set of objects OBJ is consistent if and only if* $\forall o \in OBJ$ *such that* $o.i \in \mathcal{VOI}$, *derived_by*$(o.i) \subseteq \{\hat{o}.i \mid \hat{o} \in OBJ\}$. □

5.4 Subview Relationships

An object-oriented view mechanism should keep the basic concepts of the object-oriented paradigm, so that a view can be used in any context where a class is. Following this guideline, inheritance relationships among views are supported in our model. Views are organized in a view hierarchy exactly as classes are organized in a class hierarchy. The view hierarchy is thus modeled by an ISA relationship, representing which views are subviews of other views, and must be established by the user.

The information of a view hierarchy can be expressed by two functions $VISA : \mathcal{VI} \rightarrow 2^{\mathcal{VI}}$ and $VISA^* : \mathcal{VI} \rightarrow 2^{\mathcal{VI}}$, such that given a view v, $VISA(v)$ denotes the set of direct superviews of v and $VISA^*(v)$ denotes the set of all the superviews of v, similarly to the ISA and ISA^* functions which describe the class hierarchy. An ordering $\leq_{VISA}$ on views is defined, by simply stating that $v_1 \leq_{VISA} v_2$ if and only if $v_2 \in VISA^*(v_1)$.

However, a certain number of conditions on the well-formedness of the view inheritance hierarchy must be imposed. Those conditions concern the following aspects:

- *subtyping among the structural types*: a view must have all the attributes of its superviews; attribute domains may be specialized, the implementation for a derived attribute may be redefined and new attributes can be added;
- *behavior specialization*: a view must have all the operations of its superviews; method signatures can be redefined, by applying the covariance rule for method results and the contravariance rule for method parameters, the method implementations may be redefined and new operations may be added;
- *constraint inheritance*: on a view all the constraints of its superviews must hold; constraint redefinition is not currently supported in our model;
- *extent inclusion*: the extent of a view class is a subset of the extents of all its superviews.

Those conditions are formalized as follows.

Definition 5.9 *(Int-well-defined Subview Relationship). A subview relationship $VISA$ is int-well-defined, if, for any v_1 and v_2 such that $v_2 \leq_{VISA} v_1$, V_2 precedes V_1 in the intensional ordering specified in Definition 2.31, being $V_2.id = v_2$ and $V_1.id = v_1$.* □

The definition above only concerns the structure and behavior of views, and it does not consider the extensional components of views, thus no conditions are imposed on view-queries. Let us now turn to the extensional level.

Definition 5.10 *(Ext-well-defined Subview Relationship). A subview relationship $VISA$ is ext-well-defined, if, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, V_2 precedes V_1 in the extensional ordering specified in Definition 2.31, being $V_2.id = v_2$ and $V_1.id = v_1$.* □

While the conditions for int-well-definedness of a hierarchy are conditions on the schema level, and thus can be checked at view definition time, the condition for ext-well-definedness is a state-dependent (time varying) condition, that can only be checked at run-time. However, since the instances of a view are computed starting from the view root classes and the view-query, we impose a number of conditions on view root classes and query. Such conditions ensure that if the ISA hierarchy on base classes is ext-well-defined, the same property holds for the VISA hierarchy upon view materialization.

The conditions we impose on view queries in subclasses are syntactic conditions, and they are quite restrictive. Actually, we should impose that the view-query of a view is subsumed by the view-queries of its superviews. However, the problem of query subsumption is undecidable in general, and, even for such query languages for which it is decidable, it has a very high complexity test [50]. For our (recursion-free) query language, query subsumption is decidable, though intractable because of negation and disjunction [50].

We thus impose the syntactical restriction that the view-query of the subview is stronger than the view-query of the superview, as formalized by the following definition. This syntactical condition ensures that the view query of the subview is subsumed by the view-query of the superview. The condition requires that: (i) the root classes of the subview are the same or subclasses of the root classes of the superview; (ii) for each rule R_1 in the view query of the subview there must be a corresponding rule R_2 in the view-query of the superview such that the body of R_1 can be obtained from the body of R_2 by adding some atoms and by replacing some class formulas with class formulas on subclasses.

Definition 5.11 *(View-query Strengthening). A subview relationship $VISA$ is view-query strengthening, if, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, $V_1.id = v_1$, $V_1.sign.q = q_1$, $V_2.id = v_2$, $V_2.sign.q = q_2$, both the following conditions hold:*

- *if* $defined\_on(v_1) = \{c_1, \ldots, c_n\}$, *then* $defined\_on(v_2) = \{c'_1, \ldots, c'_n\}$ *and* $\forall i, 1 \leq i \leq n$, *either* $c_i, c'_i \in \mathcal{CI}$ *and* $c_i \leq_{ISA} c'_i$ *or* $c_i, c'_i \in \mathcal{VI}$ *and* $c_i \leq_{VISA} c'_i$*;*

- *if* q_2 *consists of the rules* $H'_1 \leftarrow B'_1 \quad \ldots \quad H'_n \leftarrow B'_n$, *then* q_1 *consists the rules* $H_1 \leftarrow B_1 \quad \ldots \quad H_m \leftarrow B_m$, *and*

  - $m \geq n$,
  - $\forall j, 1 \leq j \leq m$, $\exists i, 1 \leq i \leq n$, *such that*
    - $B_j = c_j^1(X_1), \ldots, c_j^p(X_p), B_j^*$,
    - $B'_i = c_i^1(X_1), \ldots, c_i^p(X_p), B_i'^*$,
    - $B_j^* = B_i'^*, \tilde{B}$ *where* $\tilde{B}$ *is any conjunction of atomic formulas,*
    - $\forall k, 1 \leq k \leq p$, *either* $c_j^k, c_i^k \in \mathcal{CI}$ *and* $c_j^k \leq_{ISA} c_i^k$ *or* $c_j^k, c_i^k \in \mathcal{VI}$ *and* $c_j^k \leq_{VISA} c_i^k$. □

If a subview relationship is view-query strengthening and the corresponding subclass relationship is ext-well-defined, then the subview relationship is ext-well-defined.

We remark that, according to the definition of view-query strengthening, if a subview relationship $VISA$ is view-query strengthening, then, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, $new\_oid(v_1) = new\_oid(v_2)$.

Example 5.7 *Consider the view* `Magazine` *defined in Example 5.2 and the view* `VPublication` *defined as identity view of the class* `Publication`*, both belonging to the* `Bibliography` *schema of Figure 5.2. Let* $V_1.id =$ `Magazine`*,* $V_2.id =$ `VPublication`*,*

$V_1.sign.struct = \{$ *(title, string, ext), (year, integer, ext),*
(editorial, string, ext), (month, string, ext)
(manager, string, ext) $\}$.
$V_2.sign.struct = \{$ *(title, string, ext), (year, integer, ext),*
(editorial, string, ext) $\}$.

The int-well-definedness of the $VISA$ *relationship stating that* `Magazine` $\leq_{VISA}$ `VPublication` *holds, because:*

- *the condition on the struct components is verified, since* $V_2.sign.struct \subseteq V_1.sign.struct$*;*

- *the condition on the beh components follows from the int-well-definedness of the* ISA *relationship stating that* `Journal` $\leq_{ISA}$ `Publication`*, since* $V_2.sign.beh = C_2.sign.beh$*, being* $C_2.id =$ `Publication` *and* $V_1.sign.beh = C_1.sign.beh \cup \{changeManager : string \rightarrow Magazine\}$*, being* $C_1.id =$ `Journal`*;*

- *the condition on the constr components follows from the int-well-definedness of the ISA relationship stating that* `Journal` $\leq_{ISA}$ `Publication`*, since* $V_2.sign.constr = C_2.sign.constr$*, being* $C_2.id =$ `Publication` *and* $V_1.sign.constr = C_1.sign.constr$*, being* $C_1.id =$ `Journal`*.*

The ext-well-definedness of the $VISA$ *relationship stating that* `Magazine` $\leq_{VISA}$ `VPublica–tion` *follows from its view-query strengthening, since:*

- $defined\_on$(`Magazine`) $=$ {`Journal`},

  $defined\_on$(`VPublication`) $=$ {`Publication`}

  and `Journal` $\leq_{ISA}$ `Publication`*;*

- *the view-query of* `VPublication`*, since* `VPublication` *is an identity view, is*

  `VPublication(X)` $\leftarrow$ `Publication(X)`

  the view-query of `Magazine` *is*

  `Magazine(X)` $\leftarrow$ `Journal(X)`, `X.periodicity` $=$ ‘`weekly`’

  and `Journal` $\leq_{ISA}$ `Publication`*.*

$\diamond$

The generation of view object identifiers must be carefully handled when views are related by inheritance hierarchies. Recall that for a view v such that $new\_oid(v) = true$, new object identifiers are generated for each object satisfying the view-query of v. The generation of these new oids is exactly like the generation of base object identifiers upon object creation. Thus, when views are related by a subview relationship, v is a view such that $new\_oid(v) = true$ and $VISA(v) = \emptyset$, the new oids for v are incrementally generated. For the views v' such that $new\_oid(v') = true$ and $VISA(v') = \{v_1, \ldots, v_n\}, n > 0$, the objects instances of the view are by contrast extracted from the extents of the superviews, rather than being generated upon view materialization. We thus ensure that if the view-query is stronger than the view-queries of its superviews, the extent of the view is a subset of the extents of its superviews. Note that, if we had generated new oids for each view v such as $new\_oid(v) = true$, without taking into account the VISA relationships among views, the view-query strengthening of the VISA relationship would not have ensured the ext-well-definedness of the hierarchy. Thus, for object-generating views, new oids are generated upon view materialization for the views that are roots of the VISA hierarchy (which are materialized first), while for views having at least a superview the extent is determined by extracting from the superview extents those objects meeting the condition in the subview query. This process is sound though in our model multiple inheritance is supported, since the constraint is imposed that for multiple inheritance a common ancestor must exist.

View identifiers can be used as types for expressions. Thus, the notion of type proposed in Chapter 2 is modified to include also view class identifiers. The set of types $\mathcal{T}$ then consists of the value types in $\mathcal{VT}$ and the object types in $\mathcal{OT}$, that is, of class and view identifiers. The set of object types (that is, of types whose values are used to identify objects) is thus defined as the union of class and view identifiers, that is, $\mathcal{OT} = \mathcal{CVI}$.

The notion of type extension is easily extended to types that are view identifiers (that is, for $v \in \mathcal{VI}$) since views, like classes, have an explicit extent, thus $[\![v]\!] = V.state.extent$, where $V.id = v$. The subtype relationship $\leq_T$ can be simply adapted to a set $\mathcal{T}$ of types containing also view identifiers in $\mathcal{VI}$, by stating that:

- if $v_1, v_2 \in \mathcal{VI}$ and $v_1 \in VISA^*(v_2)$ then $v_2 \leq_T v_1$ that is, v_2 is a subtype of v_1 (as stated for object types in $\mathcal{CI}$);
- view class identifiers in $\mathcal{VI}$ are not related by the subtype relationship with other types.

Since we consider only int-well-defined subview relationships, this is a sound definition of subtyping.

5.5 Schema Views and Database Views

In this section, we "put things together", discussing the notions of schema views and database views.

5.5.1 Schema views

The definition of schema view or subschema often corresponds to the concept of external schema given in the ANSI three-level schema architecture. In our approach, schema views are also intended to encapsulate base schema evolutions, as a mean to prevent that a schema update affects the base schema. Thus schema views can be useful to define external schemas as well as to create new schema versions.

The notion of schema view is similar to that of base schema, except that it consists of views instead of classes. A schema view consists of a collection of views connected by *aggregation* and *view inheritance* relationships. All views in a schema view are derived from the same schema. The schema, from which a schema view is derived, is called *root schema*. A root schema can be either a base schema or another schema view. Frequently, the definition of a schema view requires to include some base classes. We propose the notion of *identity view* to satisfy this requirement, still having the schema view consisting only of views. Given a class c, its identity view is a view v having c as root class, such that c and v are equivalent, both at the extensional and at the intensional level.

The following definition formalizes the notion of schema view. Let $\mathcal{SVI}$ denote a set of schema view identifiers.

Definition 5.12 *(Schema View) A schema view having as root schema a schema S[6] is defined as a tuple*

$$(id, VT, VCl, VMCl, VISA)$$

where

- *$id \in \mathcal{SVI}$ is the schema view identifier;*
- *$VT \subseteq \mathcal{VT} = BVT \cup NVT$, is the set of value types included in the schema view, being BVT the set of value types imported from the root schema S and NVT the set of value types defined in the schema view;*
- *VCl is a finite set of definitional components of views;*
- *$VMCl \subseteq \mathcal{MC}$ is a finite set of view metaclasses corresponding to the views in VCl;*
- *$VISA : VCl \rightarrow 2^{VCl}$ is a total function on VCl for which the following conditions hold:*
  - *a) $VISA$ is a DAG;*
  - *b) $VISA$ is int-well-defined;*
  - *c) $VISA$ is view-query strengthening.*

All view and view metaclass names are distinct and for each view the corresponding view metaclass must exist. □

The following example is an example of schema view.

[6] S may be either a base schema or a schema view.

Example 5.8 *Figure 2 shows a schema view named* `Bibliography`*, directly derived from the base schema. The* `Bibliography` *schema view contains the views* `VPublication`*,* `Magazine` *and* `VBook`*. The* `VPublication` *view has been imported as an identity view, whereas* `Magazine` *and* `VBook` *views are derived from* `Journal` *and* `Book` *classes, respectively, and they are declared as subviews of the* `Vpublication` *view. The importation of* `Publication` *as identity view allows to define a virtual hierarchy of publications.* ◇

5.5.1.1 Closure of Schema Views

A schema view is a collection of views, grouped together to form a subschema, or to model a schema evolution. One is not completely free in choosing which views to include in a schema view. In particular, if a view is included in the schema view, also the domain of each attribute as well as the components of the signature of each operation of the view must belong to the schema view. Thus, including a view into the schema view may require the inclusion of other views. In the following, we formalize these notions, which are referred to as *closure* property of a schema view.

In what follows, the term *entity* refers to an attribute, c-attribute, parameter of an operation or parameter of a c-operation of a class or view. We introduce a *client_of* relationship among classes and views in $\mathcal{CVI}$. A class (or view) c_1 is said to be *client_of* a class (or view) c_2, if some entity of c_1 has as domain the class (or view) c_2. Since a user (e.g., an application program) must receive a schema view consisting of a complete and coherent set of views, it is clear that a schema view must be closed under the *client_of* relation. Given a view v included in a schema view, the closure property involves:

1. for each entity of v, whose domain is a class, the corresponding identity view must belong to the schema view and it is the new domain of the entity;
2. for each entity of v, whose domain is a view, this view must belong to the schema view.

A closed schema view contains all the views referenced directly or indirectly by the schema view definition. Since we have chosen to model a schema view as a collection of views, if the domain of an entity of v is a class of the base schema, the corresponding identity view is introduced in the schema view, thus "virtualizing" the class without modifying it.

We remark two important aspects related to our definition of schema view closure. First, since our model does not require the existence of a common superclass of all the classes of the system, the closure property does not involve the inclusion of all the superviews of the views belonging to the schema. Second, the closure of a schema does not require the inclusion of all the subviews of the views belonging to the schema. Thus, a schema view is closed with respect to aggregation hierarchies, while it is not closed with respect to inheritance hierarchies. Indeed, the schema closure must contain the essential views for a schema to be consistent. In some contexts, it seems reasonable that a schema view includes a view and it does not include some of its subviews (so that some of the subviews are hidden in the schema view). As a consequence of this choice, the database view (that is, the database seen through the schema view) may contain objects whose most specific classes and views do not belong to the schema view. Such objects are seen through the schema view as instances of the most specific view to which the object belongs, among the ones included in the schema view. Note that this will require a careful propagation of updates, since an attribute could result in having different domains depending on through which schema view it is accessed. Possible solutions are: to consider the attribute as a different attribute (for example, prefixing its name with the schema name) and thus allocating different storage space; to handle the propagation through triggers which specialize the value assigned to the attribute to the required domain. The most conservative solution to avoid those problems is to constrain a schema to contain also the subclasses of the classes in the schema, for those subclasses that refine the domain of some included attribute. Since those problems are related to update propagation, we do not elaborate on them further in this chapter.

To formalize the closure property of a schema view, a function *id_view* is introduced. It applies to a schema view and returns the schema view modified by substituting each class identifier belonging to $\mathcal{CI}$ and appearing as domain of an entity in the schema, with the identifier of the corresponding identity view, which belongs to $\mathcal{VI}$. Thus, given a schema view SV, $id\_view(SV)$ denotes a corresponding schema view having only views or value types as entity domains. Furthermore, given a view v, belonging to a schema view SV, let $dom(v)$ denote the set of value types and views which are domains of the entities of v in $id\_view(SV)$. This set can be partitioned in $vdom(v)$, only containing view identifiers, obtained as $dom(v) \cap \mathcal{VI}$, and $tdom(v)$, only containing value types, obtained as $dom(v) \cap \mathcal{VT}$.

Definition 5.13 *(Schema View Closure). A schema view SV is said to be closed if both the following conditions are satisfied:*

1. $\forall v \in SV.VCl$, $vdom(v) \subseteq SV.VCl$
2. $\forall v \in SV.VCl$, $tdom(v) \subseteq SV.VT$. □

We remark that, given a schema view SV, it is decidable whether SV is closed.

The following example illustrates the closure property of a schema view.

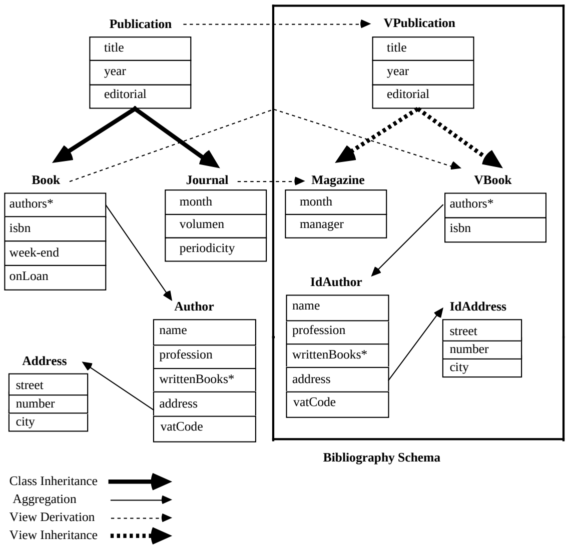

Figure 5.3: Closure of the `Bibliography` view schema.

Example 5.9 *Consider again the* `Bibliography` *schema view of Figure 2. Since the view* `Vbook` *contains an attribute named* `authors` *whose domain is class* `Author`*, the domain of this attribute must be replaced by the identity view* `IdAuthor` *of the* `Author` *class, and this identity view must be included in the schema. This inclusion is propagated to the* `Address` *class because there is an* `address` *attribute in the* `Author` *class whose domain is* `Address`*, so that the identity view* `IdAddress` *is also included. The closure of the schema is depicted in Figure 5.3. The derivation links between* `IdAuthor` *and* `Author`*, and between* `IdAddress` *and* `Address`*, are not depicted in the figure, to point out that these views are added to the schema view to obtain a closed schema view.* ◇

5.5.1.2 Global Database Schema

Now, by using the notion of schema views, we give a formal definition of global database schema. A global database schema consists of a base schema together with a set of schema views. The schema derivation and view derivation relationships are part of the global database schema, too.

Definition 5.14 *(Global Database Schema). A global database schema is a tuple*

$$(bs, SVS, defined\_on^s, defined\_on),$$

where:

- *bs is a base schema identifier,*
- $SVS = \{sv_1, \ldots, sv_n\}, n \geq 0$, *is a set of schema view identifiers,*
- $defined\_on^s : \mathcal{SVI} \rightarrow \mathcal{SVI} \cup \{bs\}$ *is a total function, that given a schema view returns its root schema,*
- $defined\_on : \mathcal{VI} \rightarrow 2^{\mathcal{CVI}}$ *is the function that represents the view derivation relationship.*

The schema must satisfy the following conditions:

1. *each schema in SVS must be closed according to Definition 5.13;*
2. *for each $sv_i, sv_j \in SVS$, if $defined\_on^s(sv_j) = sv_i$, then $\forall v \in SV_j.VCl$, $defined\_on(v) \subseteq SV_i.VCl$, being $sv_i = SV_i.id$ and $sv_j = SV_j.id$.* □

Note that the view derivation relationship may connect view and classes belonging to different schemas, thus it is part of the global database schema rather than of an individual schema view.

Example 5.10 *Referring to our running example, the global database schema consists of the* `Faculty- Library` *base schema and the* `Bibliography` *schema view, with the following relationships:*

- $defined\_on^s$(`Bibliography`) = `FacultyLibrary`;
- $defined\_on$(`VPublication`) = `Publication`

  $defined\_on$(`Magazine`) = `Journal`

  $defined\_on$(`VBook`) = `Book`

  $defined\_on$(`IdAuthor`) = `Author`

  $defined\_on$(`IdAddress`) = `Address`. ◇

5.5.2 Object Database

In our approach, we consider a single database which is associated with the global schema, and which is shared by all the existing schemas. The database contains all the instances created from classes and views belonging to the global schema. Thus, the database consists of objects defined according to Definition 5.4, that can be, as a particular case, view objects as in Definition 5.5. An object $o = (i, v, VS)$ can be accessed from any schema including a view or class of which o is member. We remark that this view or class does not necessary belong to the set VS, since VS only contains the classes and views of which o is an instance. Suppose that an object o can be accessed through the schema view SV, the function *value*, defined in Section 5.3.1, is then used to provide the different aspects under which the object can be seen. Therefore, each schema view in the global schema has associated a view database [32]. Now, we reformulate the definition of database, before giving a definition of view database.

Given a global database GS, let $\mathcal{CI}^{GS}$ denote the set of identifiers of classes belonging to the global database schema, which are those included in the base schema, that is

$$\mathcal{CI}^{GS} = \{c \mid \ c \in \mathcal{CI}, \ \ c = C.id, \ \ C \in S.VCl, \ \ S.id = GS.bs\}.$$

Let moreover $\mathcal{VI}^{GS}$ denote the set of identifiers of views belonging to the global database schema, which are those included in any schema view, that is

$$\mathcal{VI}^{GS} = \bigcup_{sv_i \in GS.SVS} \mathcal{VI}^i$$

where each $\mathcal{VI}^i$ is obtained as

$$\mathcal{VI}^i = \{v \mid \ v \in \mathcal{VI}, \ \ v = VC.id, \ \ VC \in SV_i.VCl, \ \ SV_i.id = sv_i\}.$$

Finally, $\mathcal{CVI}^{GS} = \mathcal{CI}^{GS} \cup \mathcal{VI}^{GS}$.

Definition 5.15 *(Object Database). Let GS be a global database schema defined according to Definition 5.14, a database over GS is a tuple*

$$(OT, \pi, cval)$$

where

- $OT = OB \cup OV$, *is a consistent set of objects, being*

- OB *a set of base objects;*
- OV *a set of view objects;*

- π *is a pair of functions*

  $\pi_{OC} : \mathcal{CI}^{GS} \rightarrow 2^{\mathcal{BOI}}$, *oid assignment for base classes,*
  $\pi_{OV} : \mathcal{VI}^{GS} \rightarrow 2^{\mathcal{VOI}}$, *oid assignment for views,*

  which handle class extents;

- *cval is a total function* $cval : \mathcal{CVI}^{GS} \rightarrow \mathcal{V}$, *such that* $\forall c \in \mathcal{CVI}^{GS}$ $cval(c)$ *is a legal value for* $stype(C.mc)$, *being* $C.id = c$, *that is, the function cval assigns values to the class attributes of* C. □

For an object database to be consistent, we require that it satisfies a number of conditions, as stated by the following definition. These conditions are mainly related to the proper assignment of objects to classes and views.

Definition 5.16 *(Consistent Object Database). Let* GS *be a global database schema defined according to Definition 5.14, a consistent object database over* GS *is a tuple*

$$(OT, \pi, cval)$$

defined according to Definition 5.15 such that:

(i) $\forall o \in OT, o$ *is a consistent instance of each class and view in* $o.VS$, *that is* $o.v$ *holds all the values for the attributes of classes in* $o.VS$*;*

(ii) the ISA component of GS *is ext-well-defined;*

(iii) the VISA components of GS *are ext-well-defined;*

(iv) $\forall c \in \mathcal{CVI}^{GS}$, $c = C.id$,
$cval(C).extent = \pi^*(c)$ *and*
$cval(C).proper\_extent = \pi(c)$*;*

(v) $\forall o \in OT,\ \ \forall c \in \mathcal{CVI}^{GS}$, $c = C.id$, $o.i \in cval(C).proper\_extent$ *iff* $c \in o.VS$*;*

(vi) $\forall o \in OT, o.VS \subseteq \mathcal{CVI}^{GS}$ *and*

- $\forall o \in OB, o.VS \subseteq \mathcal{CVI}^{GS}$ *and* $\exists c \in o.VS$ *such that* $c \in \mathcal{CI}^{GS}$, *while*

– $\forall o \in OV, o.VS \subseteq \mathcal{VI}^{GS}$;

(vii) $\forall v \in \mathcal{VI}^{GS}$, *if* $next\_oid(v) = false$ *and* $defined\_on(v) = \{c_1, \ldots, c_n\}$, *then* $\pi(v) \subseteq \bigcup_{i=1,\ldots,n} \pi(c_i)$;

(viii) $\forall o \in OV$, $\exists v \in \mathcal{VI}^{GS}$ *such that* $v \in o.VS$ *and* $next\_oid(v) = true$;

(ix) $\forall o \in OT$, $\forall v \in o.VS \cap \mathcal{VI}^{GS}$, $oid(v) = true$;

(x) $\forall v \in \mathcal{VI}^{GS}$ *such that* $oid(v) = false$, $\pi(v) = \emptyset$;

(xi) $\forall c_1, c_2 \in \mathcal{CVI}^{GS}$

– *if* $c_1, c_2 \in \mathcal{CI}^{GS}$

$$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow \exists c_3 \in \mathcal{CI}^{GS} \textit{ such that } c_1 \leq_{ISA} c_3, c_2 \leq_{ISA} c_3;$$

– *if* $c_1, c_2 \in \mathcal{VI}^{GS}$ *and* $new\_oid(c_1) = new\_oid(c_2) = true$

$$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow \exists c_3 \in \mathcal{VI}^{GS} \textit{ such that } c_1 \leq_{VISA} c_3, c_2 \leq_{VISA} c_3;$$

– *if* $c_1, c_2 \in \mathcal{VI}^{GS}$ *and* $new\_oid(c_1) = new\_oid(c_2) = false$

$$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow (\exists c_3 \in \mathcal{VI}^{GS} \textit{ such that } c_1 \leq_{VISA} c_3, c_2 \leq_{VISA} c_3) \vee \vee (\exists c_4 \in \mathcal{VI}^{GS} \textit{ such that } c_4 \in defined\_on^*(c_1) \wedge c_4 \in defined\_on^*(c_2)));$$

– *if* $c_1, c_2 \in \mathcal{VI}^{GS}$ *and* $new\_oid(c_1) = true$ *while* $new\_oid(c_2) = false$

$$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow c_2 \in defined\_on^*(c_1);$$

– *if* $c_1 \in \mathcal{VI}^{GS}$, $c_2 \in \mathcal{CI}^{GS}$

$$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow c_2 \in defined\_on^*(c_1) \wedge new\_oid(c_1) = false.$$

□

In the definition above, condition *(vi)* requires that each base object is instance of at least one base class whereas each view object only belongs to views. Condition *(vii)* ensures that the extent of an object-preserving view is contained in the union of the extents of its root classes (we recall that an object-preserving view could select objects from different classes), while condition *(viii)* imposes that the views to which a view object belongs are object-generating views. Finally, condition *(xi)* states the conditions under which two classes or views may have non disjoint extents: they can be either both base classes or both object-generating views with a common ancestor in the inheritance hierarchy; they can be both object-preserving views with a common ancestor in the inheritance hierarchy or a common root class; or they can finally be a base class and an object-preserving view such that the view is derived from the base class.

Definition 5.17 *(Global Database). Let GS be a global database schema, then a global database on GS is a tuple exactly like in Definition 5.15, of the form $(OT, \pi, cval)$ such that OT is closed under the view derivation relationship according to Definition 5.8.* $\square$

Each schema views specifies a different view of the global database, as stated by the following definition.

Definition 5.18 *(Database View). Let SV be a schema view on a consistent object database ODB, then a database view is a tuple exactly like in Definition 5.15, of the form $(OT_{SV}, \pi_{SV}, cval_{SV})$ satisfying the following further conditions:*

(i) $I(OT_{SV}) \subseteq I(ODB.OT)$[7] *is such that*

- $\forall o \in OT_{SV}$, $\exists v \in \mathcal{VI}^{SV}$ *such that* $o.i \in \pi(v)$;
- $\forall o \in OT_{SV}$, *let* $o' \in ODB.OT$ *be the object such that* $o.i = o'.i$; *then*
  - $o.v$ *contains*[8] $o'.v$, *and*
  - $o.VS = o'.VS \cap \mathcal{CVI}^{SV} \cup ms(\{v \mid v \in \mathcal{VI}^{SV}, \exists v' \in o.VS, v' \notin \mathcal{VI}^{SV}, v' \leq_{VISA} v\}$
    where, given a set of views VS, $ms(VS)$ denotes the set $\{v \mid v \in \mathcal{VI}, \nexists v' \in \mathcal{VI}$ *such that* $v' <_{VISA} v\}$[9];

(ii) π_{SV} *is the restriction of the function in* $ODB.\pi$ *to views in* $\mathcal{VI}^{SV}$;

(iii) $cval_{SV}$ *is the restriction of the function in* $ODB.cval$ *to views in* $\mathcal{VI}^{SV}$. $\square$

5.6 Object References and Contexts

Another important aspect concerning views is how object references are solved, since "different" objects can be identified by the same object identifier. Indeed, different views of the same object are allowed, depending on the *context* in which the object is considered.

In our approach, the class (or view) the referenced object must be seen an instance of, is chosen among the ones belonging to the current schema view, taking into account the

[7] Given a set of objects OBJ, $I(OBJ)$ denotes the set of identifiers of objects in OBJ, that is, the set $\{o.i \mid o \in OBJ\}$.

[8] We say that a record value *record-of*$(a_1 : v_1, \ldots, a_n : v_n)$ *contains* a record value *record-of*$(a'_1 : v'_1, \ldots, a'_m : v'_m)$ if $m \leq n$ and, $\exists f : \{1, \ldots, m\} \to \{1, \ldots, n\}$ total injective function, such that $\forall i, 1 \leq i \leq n$, $a'_i = a_{f(i)}$ and $v'_i = v_{f(i)}$.

[9] $<_{VISA}$ denotes the non-reflexive relation obtained from the order $\leq_{VISA}$ as follows: $v_1 <_{VISA} v_2$ if and only if $v_1 \leq_{VISA} v_2$ and $v_1 \neq v_2$.

context of the object reference. The context of an object reference is simply determined by the static type of the object in the expression containing the object reference. Indeed, each object reference in each expression is assigned a single static type (see Chapter 3). Let this type be $static\_type(e)$. Thus, it is possible in each expression to derive a unique context for each expression denoting an object (object reference). The context of an object reference can be derived from the types declared for the variables in the expression and from schema information.

As far as attribute access is concerned, an attribute access $e.a$ is solved by simply returning $value(o, static\_type(e)).a$ where $value$ is the function defined in Section 5.3, $static\_type(e)$ is the static type of the object reference e in the considered expression and o is the object to which reference e is instantiated. Note that, for the expression containing the reference to be a legal expression, $static\_type(e)$ must be a view identifier belonging to the schema view of the user that has written the expression and that attribute a must belong to the structure of $static\_type(e)$ in this schema view.

Method dispatching may however become more complicated when several method implementations are applicable to a method invocation. In general, the implementation specified in the most specific class of the invocation receiver is executed, as it is the one that *most closely matches* the invocation. However, when objects belong to classes and views not related by an inheritance relationship, the choice of the method implementation that "most closely matches" the invocation is not obvious. We will address the problem of dispatching for objects belonging to multiple most specific classes in Chapter 6[10]. The approach is based on the idea that each object has in each context a *preferred* class, among its most specific ones. This approach can be easily adapted to our framework. Each method invocation is dispatched choosing the implementation in the preferred class in the current context. This approach supports a context-dependent behavior, as the same method invocation may be dispatched differently, and thus may return different results and perform different updates, depending on the context where the method is invoked. This approach is based on a total ordering on views, defined consistently with the inheritance ordering, in such a way that a method invocation is dispatched by executing:

- the method in the view which is the static type of the object reference, if this view is among the most specific ones to which the object belongs;
- the minimum with respect to the considered total ordering, of a set of views that verify the following conditions:
  - they are subviews of the static type of the object reference (this ensures that no run-time type errors occur),

[10] Actually, in Chapter 6 we will consider only base classes, but considering views does not introduce new problems with respect to dispatching.

- they belong to the current schema view,

otherwise;

- the method in the root class of the view which is the static type of the object reference, if the two cases above fail to dispatch the method (resolution in the root schema).

Note that while the static type of the object reference certainly belongs to the user schema (otherwise the expression containing the reference would not be correct), this may not be true for any subview/subclass of this type. Thus, we remark that, though we store a single object database, the objects stored in the database behave differently depending on through which schema view they are accessed. The behavior does not only depend from the static type of the object reference, but also from the schema view in which the object reference is contained.

5.7 Concluding Remarks

In this chapter we have proposed a view mechanism for our object-oriented data model. An important aspect of our view mechanism is that different views of a single object database are provided, through different schema views. A schema view is a coherent set of views. The schema view through which an object database is accessed also influences the object behavior, thus providing a context-dependent behavior. Schema views also support a mechanism of schema versions, such that a single database is shared by all the schemas.

Our view model is comparable to existing view models for what concerns the supported features; however, our model is the only one for which a formal definition is given. This formal definition of object-oriented view mechanisms is useful both for understanding what views are and as a basis for further investigations on view properties.

Chapter 6

Method Dispatching

In most object-oriented systems each object must be an instance of a single class. This exclusive link between an instance and a class is often not adequate to model real-world situations. In our model, by contrast, objects can be members of multiple classes, even not related in the inheritance hierarchy. However, when a model supports multiple class direct membership, conflicts among different definitions may arise. If an object has several most specific classes, the object takes the union of the features of all the classes to which it belongs. Such classes can, however, have some common properties and the problem thus arises of selecting the appropriate property for the object. Such conflicts are similar to conflicts due to multiple inheritance. However, multiple inheritance conflicts can be detected when classes and inheritance hierarchies are defined at compile-time, whereas in the case of multiple class direct membership an object may become and cease to be a member of a class at run-time.

The problem is particularly relevant for methods. Each class of the inheritance hierarchy can specify a different implementation for a certain method. For each method invocation, an implementation must be chosen among the most specific ones for the receiver object. In a model supporting multiple class direct membership, however, the choice of a method implementation that "most closely matches" an invocation is not obvious. The problem of dispatching in object-oriented systems with multiple class direct membership has been quite neglected in the literature.

In this chapter, we address the problem of method dispatching in an object data model supporting multiple class direct membership. In particular, two different dispatching rules have been proposed: one based on the notion of *preferred class*, and the other relying on the notion of *argument specificity*. The preferred class dispatching rule is based on the idea that each object has in each context a *preferred* class among its most specific ones. A method invocation is dispatched by choosing the implementation in the preferred class in the current context. The preferred class is determined by considering the static type of the receiver object and a total order on classes (that may depend on the specific object). The argument specificity dispatching rule, by contrast, does not use the invocation context to dispatch the message; rather it considers the dynamic types of other actual parameters of the invocation, in addition to the dynamic type of the object receiver of the method invocation. The first approach ensures context-dependent behavior; the second one ensures behavior identity. In the chapter we also show how the techniques can be efficiently realized and how type safety can be ensured.

The chapter is organized as follows. Section 6.1 introduces the problem of method dispatching and surveys existing approaches. Section 6.2 introduces preliminary notions and addresses the issue of method applicability. Section 6.3 presents the two dispatching rules, whereas Section 6.4 presents the lookup automata approach for an efficient implementation of the argument specificity dispatching rule. In Section 6.5 the two approaches are compared and typing issues are addressed.

6.1 Method Dispatching Techniques

In an object-oriented language, a method is characterized by a certain number of arguments, each associated with a type, and can be executed on any object instance of this type or of any subtype of this type. The choice of the actual method to execute (referred to as method *dispatching* or method *lookup*) depends on the types of the actual parameters of the method invocation. In presence of subtype relationships, this selection is in general performed at run-time (*late binding*). In most object-oriented languages, a message is sent to a receiver object and the dynamic type of the method receiver determines the method to execute. The remaining arguments of the invocation provide values for the selected method, but they play no role in method selection. Methods dispatched in such a way are referred to as *selfish* methods since methods are dispatched depending only on the type of a single argument (called *distinguished parameter*). Single dispatching is adequate for several kinds of messages, especially those in which the first argument is more "interesting" than other ones, in that the first argument alone is sufficient to determine the method code to be executed. It is however not adequate for all kinds of methods; an example is given by binary methods[1] [49]. *Multi-methods* [45, 56, 68, 136] are a generalization of selfish methods. A multi-method is a collection of method bodies associated with a single message name. The selection of which method body to execute depends on the classes of one or more method parameters (rather than just on the class of the receiver as in selfish methods).

[1]A binary method of some object is a method that expects to be passed another object of the same class as an argument.

In the following subsections we describe the existing techniques for efficiently dispatching methods, both in the case of selfish methods and in the case of multi-methods. Then we discuss dispatching for predicate classes and for models supporting roles.

6.1.1 Selfish Method Dispatching

Each class c owns a method dictionary, associating a method body with each name for which c is a target argument. A naive approach to method dispatch is the following. Given an invocation of method `m` on an object of class c, the method dictionary of c is searched to find the method body associated with the name `m`. If a method is found, that method is selected for execution. If no method is found, the method dictionary of the superclass of c is searched. The search process stops when either a method is found or the class under examination has no superclasses. Ambiguities due to multiple superclasses are usually handled by setting a precedence order on the superclasses. This approach, referred to as *schema search*, is expensive, since it involves searching method dictionaries and the number of searches is proportional to the depth of the class hierarchy. Several optimization techniques have been proposed to avoid unnecessary searches. The technique known as *caching* consists of recording the most specific applicable method found by previous searches in a cache. The cache is then searched first and if a method is not found, the most specific applicable method is retrieved using the schema search approach. The technique known as *inlining* performs type analysis and run-time type tests to avoid method dispatching and inlines the code of the most specific applicable method.

In the *dispatch table* technique, the most specific applicable method is precomputed at compile time for every pair of method name and target class, and stored into a dispatch table. Thus, the most specific applicable methods for all possible invocations are always available. The main advantage of dispatch tables is to allow a fast, constant-time method selection, whereas the major drawback is their size. Thus, dispatch table compression techniques have been proposed. *Coloring* [82] is a compression techniques that aims at eliminating the large number of empty entries in dispatch tables. Indeed, most methods are not applicable to every class. Two methods names that are never applicable to the same class are said to be non-conflicting as they can share the same index in the dispatch tables and their rows can be merged.

6.1.2 Multi-method Dispatching

As the selection of multi-methods involves all the arguments, a multi-method cannot be attached to a single class. To test the applicability of a multi-method, it must be checked that the types of all actual arguments are subtypes of the types of corresponding formal arguments of the method. Among the applicable methods, the method whose signature most closely matches the dynamic types of the actual arguments (*most specific applicable method*) is searched for. The basis for establishing the specificity of a method is a precedence relationship referred to as *argument subtype precedence* [10]: a method $\mathtt{m}_i$ is more specific of a method $\mathtt{m}_j$ (denoted as $\mathtt{m}_i <_{\mathtt{m}} \mathtt{m}_j$) if all the arguments of $\mathtt{m}_i$ are subtypes of the arguments of $\mathtt{m}_j$. That relationship, however, does not produce a total order in presence of multiple inheritance or multi-methods. Thus, to ensure the uniqueness of the most specific applicable method (and to enable dispatching) a *complementary precedence order* $\mathtt{E}$ is considered.

The search for the most specific applicable method cannot be accomplished by ascending the class hierarchy as in the case of selfish methods. Given a method invocation, the selection of the most specific applicable method is achieved as follows: the set of all method bodies corresponding to the same name, ordered from the most specific to the least specific, is iterated over. The first method that is applicable is selected. As in the case of selfish methods, schema search is very expensive in terms of execution time.

Limited work has been done in optimizing multi-method dispatch. Amiel et al. in [15] have proposed a constant-time dispatching technique for multi-method, with a low spatial complexity. They have extended the dispatch table approach to multiple-dispatching. A dispatch table for each method name is kept. The dispatch table for a method $\mathtt{m}$ with arity n is an n-dimensional array. Every entry in the dispatch table represents a signature for the method. The entry at position $(T_1, \ldots, T_n)$ holds the most specific applicable method for the invocation $\mathtt{m}(T_1, \ldots, T_n)$ and a null value if there is no most specific applicable method. Dispatching of a method $\mathtt{m}$ using a dispatch table is performed by selecting the method contained in the entry of the dispatch table for method $\mathtt{m}$ corresponding to the types of the actual arguments of the invocation. The selection of the most specific applicable method for an invocation is thus very fast, because it requires a single n-dimensional array access. By contrast, the size of the dispatch table is very high, as it is $\mid \mathcal{T} \mid^n$[2], being $\mathcal{T}$ the set of types. The main drawback of dispatch tables for multi-methods is thus related to their dimension, which is exponential in the number of arguments. It has been then proposed to reduce the number of entries of each dispatch table thus obtaining a compressed dispatch table. The compression technique is based on two basic ideas: eliminating rows or columns containing only null values, and grouping identical rows or columns.

An alternative approach to multi-method dispatch, based on the use of automata, has been proposed by Chen, Turau and Klas [71]. They propose the use of lookup automata (LUA) to simulate method dispatch. The idea is to associate with states of the automaton precedence lists of methods (that is, lists of method sets), representing the methods applicable in the state, ordered according to their specificity. Transitions are labeled by types, so that, in the simulation of the automaton, the transition labeled by the most specific supertype of the type of the corresponding actual parameter, is performed. The automaton has as

[2] Given a set $\mathbf{S}$, the notation $\mid \mathbf{S} \mid$ denotes its cardinality.

many levels as the arity of the method it corresponds to. Lookup automata ensure a minimal space-complexity, because states with the same precedence lists are never duplicated. Moreover, simulation is such that backtracking is never required; thus, its cost is linear in the number of method parameters (depth of the automaton).

To conclude the discussion, we can state that automata have some advantages over compressed dispatch tables for multi-methods dispatch, namely: (*i*) automata reduce space occupation also during their construction, whereas dispatch tables are first built and then compressed; (*ii*) dispatch table, even if compressed, are often sparse (causing space waste); (*iii*) it is not possible to group together two rows or columns of a dispatch table that are only partially identical, without increasing the temporal complexity. Finally, both the dispatching techniques (lookup automata and compressed dispatch tables) use the total order `E` only during construction (at compile-time), whereas during simulation (at run-time) they only make use of the partial order $<_{\mathtt{m}}$.

6.1.3 Method Dispatching for Predicate Classes

Among languages supporting predicate classes, Cecil [67] is the only one for which a dispatching mechanism has been developed. Cecil is based on multiple-dispatching. In Cecil, methods are defined by specifying a name, the formal parameters and an implementation. Each formal parameter can, optionally, be associated with an *argument specializer* $\mathtt{obj}_i$, specifying that the method is defined only if the actual parameters (that is, the message arguments) are descendants of object $\mathtt{obj}_i$[3]. For non-specified formal parameters (that is, without argument specializer), any value is legal as actual parameter. Argument specializers are the mean for associating the (multi-)method with specialized objects.

In Cecil methods are dispatched as follows. First, methods applicable for the message, that is, methods with the same name and number of arguments of the message, and whose argument specializers are ancestors of corresponding actual parameters, are determined. Applicable methods are then ordered by their specificity: a method $\mathtt{m}_1$ is more specific than a method $\mathtt{m}_2$ if any argument specializer of $\mathtt{m}_1$ is a descendant of the corresponding argument specializer of $\mathtt{m}_2$ and at least one of the argument specializers of $\mathtt{m}_1$ is a proper descendant of (that is, different from) the corresponding argument specializer of $\mathtt{m}_2$. If a unique most specific applicable method does not exist, an "ambiguous message" error is generated and the message is not dispatched. Differently from other languages supporting multi-methods, Cecil does not make use of any ordering on objects or on arguments to solve ambiguities in an automatic way. Thus, no complementary precedence order is employed.

[3]Cecil does not support the notion of class, thus inheritance relationships are specified at the object level.

6.1.4 Method Dispatching in Object Data Models Supporting Roles

The emphasis in data models supporting roles is on context-dependent behavior. In Fibonacci [12], the selection of the method to be executed depends on the role receiving the message. Dispatching is based on the following basic principles: (*i*) the most specific behavior prevails (unless a strict interpretation of messages is explicitly required); (*ii*) the most recently acquired behavior prevails. In Fibonacci, messages are interpreted as follows. When a role receives a message, first it is checked whether a descendant of this role exists having a proper (that is, non-inherited) method to reply to this message. Descendants are considered in inverse temporal order, that is, the more recently acquired descendant is considered first. Subtyping rules ensures that the delegated role can safely replace the receiving role. If no descendant role is able to handle the message, an implementation for the message is then looked for among the methods of the receiver itself. If also this search fails, an implementation for the message is finally looked for in the ancestor role from which the property corresponding to the message has been inherited. If the method invocation is type correct, this last search will surely succeed. Fibonacci also supports an alternative dispatch mechanism (referred to as *strict binding*) to force an object to exhibit the behavior of a certain role without keeping into account possible specializations of the role. This mechanism must be explicitly required, through a special operator, when a message is sent to a role.

In the role model of Gottlob et al. [94], a message that cannot be handled by any role instance is delegated to the more general instance in the role hierarchy. In that model, however, no priorities are used to select a subrole among a number of candidates; this approach can lead to unproper behaviors. Consider, for example, the roles *Entrepreneur* and *Employee* of type *Person*, each defining a method *income*. Referring to this example, the income of a person might not be reduced neither to its income as an employee nor to its income as an entrepreneur, nor to the one of the most recently acquired role. By contrast, the income of a person can be obtained as an aggregate of the incomes of all its roles. Aggregation is one choice, but it is not always the most meaningful one.

6.2 Preliminaries

In this section we introduce some preliminary definitions concerning type hierarchies and methods which we need for describing our dispatching techniques. In particular, we first introduce some basic notions on types and inheritance orders, then we investigate the relationships among static and dynamic types in an object data model supporting multiple class direct membership, and finally address the issue of method applicability.

6.2.1 Type Hierarchy and Type Orders

We have already introduced in Chapter 2 our type hierarchy $(\mathcal{T}, \leq_T)$. We recall that relation $\leq_T$ defines a *partial* order and that the type system $(\mathcal{T}, \leq_T)$ can be represented as a directed acyclic graph (DAG).

The following definition introduces the notion of most specific types in a set. Intuitively, the most specific types in a set of types are the minimal elements of the set with respect to the partial order $\leq_T$.

Definition 6.1 *(Most Specific Type). Given a set of types* $\mathbf{S}$*,* $ms(\mathbf{S})$ *denotes the set* $\{ T \mid T \in \mathbf{S} \wedge \not\exists T' \in \mathbf{S}$ *such that* $T' <_T T \}$*. A type* T *is said a most specific type in* $\mathbf{S}$ *if* $T \in ms(\mathbf{S})$*.* $\square$

We introduce the following notation representing the most specific classes to which an object o belongs.

Definition 6.2 *Given an object* o*,* $msc(o)$ *denotes the set of most specific classes to which* o *belongs, that is, the set* $ms(\{ c \mid c \in \mathcal{CI} \wedge o.i \in [\![c]\!] \})$*.* $\square$

As $\leq_T$ is a partial order, not all the types in a hierarchy can be compared under $\leq_T$. Thus, as we will see in Section 6.3, we need to introduce a *total* order on types, that we denote as $\preceq_T$, to dispatch messages for objects that are direct members of multiple classes. Indeed, in systems supporting multiple class direct membership, as in systems supporting multi-methods or multiple inheritance, the partial order $\leq_T$ alone is not sufficient to unambiguously dispatch each method invocation. The total order $\preceq_T$ we introduce has the same role of the complementary precedence order `E` we have described in Section 6.1.2 for multi-method dispatching. The order $\preceq_T$ must support the comparison of the types in a hierarchy that cannot be compared by means of $\leq_T$. Therefore $\preceq_T$ must respect the partial order $\leq_T$ (that is, $\leq_T \subseteq \preceq_T$). There are different possible options for defining the total order $\preceq_T$. This order can be determined by the definition order of classes, eventually overridden by `before/after` clauses in class definitions. The order could also be obtained as a linearization of the class hierarchy, as the ones considered for solving conflicts arising due to multiple inheritance. Alternatively, we may consider for each object a total order on its most specific classes (for example, the one determined by the acquisition order, in such a way that the most recently acquired class precedes the others in the order). We are indeed particularly interested in an order $\preceq_T$ that depends on the object and is thus known only at run-time.

6.2.2 Static and Dynamic Types

As we have discussed in Chapter 3, our model supports a strongly typed database language. Thus, when writing an expression, one must explicitly state the type of each variable used in the expression. This is accomplished by *class formulas*, that are used as a typing mechanism. Class formulas are built from class or type names representing unary predicate symbols. Thus, to state in an expression that a variable X is of type T the class formula $T(X)$ is added to the expression. The class formula $T(X)$ has also the effect of stating that $X \in [\![T]\!]$, thus providing a domain for the evaluation of the expression.

The scope of a type assignment for a variable is that of an expression. Each expression must contain a class formula for each variable and thus each variable is associated with a unique type, with respect to which type checking is performed. Declaring a variable X of type T influences type checking in that the only features available for X are those of type T; that is, type checking of the expression in which X appears is performed regarding X as a term of type T. We say that T is the **static type** of variable X, because it is the type with respect to which the static type checking of the expression containing X is performed.

At execution time variable X is instantiated with a value of type T. Note that when T is a class name, the variable is instantiated with the identifier of an object member of the class identified by T. This means that the variable may also be instantiated with the identifier of an object which is a direct member of a class T', subclass of T. T' is said to be a **dynamic type** of variable X. However, since we consider objects possibly belonging to several most specific classes, the dynamic type of a variable denoting an object is, in general, not unique. Thus, a variable is characterized by a set of dynamic types. The only constraint on those dynamic types is that they must have a common ancestor in the inheritance hierarchy. Moreover, the static type T of the variable denoting the object must be a subtype of the common ancestor. Finally, at least one dynamic type must be a subtype of the static one. These relationships are stated by the Rule 6.1 below.

Rule 6.1 *(Relationship between the Static Type and the Dynamic Types of Objects). Let $(\mathcal{T}, \leq_T)$ be a type system. Let $static\_type(X)$ be the static type of an object reference X. Let o be an object to which X is instantiated at run-time, and let $msc(o)$ be the set of dynamic types of o. The following relationships hold:*

1. $\exists c \in msc(o)$ *such that* $c \leq_T static\_type(X)$;
2. $\exists\ c \in \mathcal{T}$ *such that* $\forall\ c' \in msc(o) : c' \leq_T c \wedge static\_type(X) \leq_T c$[4]. $\Box$

[4] Due to the properties of inheritance hierarchies, this is equivalent to the statement that each class in $msc(o)$ and $static\_type(X)$ belong to the same strongly connected component $\mathcal{H}_i$ of the inheritance DAG.

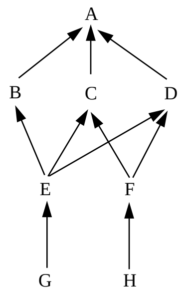

Figure 6.1: Type hierarchy $\mathcal{T}$

Example 6.1 *Referring to the inheritance hierarchy of Figure 6.1, which we will use as a running example throughout the chapter, consider a variable X declared of type F. At run-time variable X can be instantiated with an object o such that $msc(o) = \{G, H\}$. The relationships stated in Rule 6.1 between static_type$(X) = F$ and $msc(o)$ hold; in particular, relationship 1 holds with $c = H$, whereas relationship 2 holds with $c = A$ (or C, or D).* $\diamond$

The static type of an object is not only exploited in restricting access to a particular context, but may also determine a context-specific behavior. Dynamic types are used for choosing the appropriate method definition to be used on the object denoted by X. Note that the chosen implementation must always be the implementation provided by one of the most specific classes of the object.

6.2.3 Method Applicability

A method implementation is applicable to a method invocation if such an implementation is defined in a class among the most specific ones to which the distinguished receiver object belongs (according to the late binding principle), and if actual arguments are legal values for the corresponding formal ones. The following definition formalizes such notion.

Definition 6.3 *(Method Applicability). Let $sign(\mathtt{m}, c_j) = T_1^j \times \cdots \times T_n^j \to T_r^j$ [5] be the signature of method* m *in class c_j. The implementation of such method is applicable to a method invocation $o.\mathtt{m}(v_1, \ldots, v_n)$ if*

1. $c_j \in ms(\{\ c \mid o.i \in [\![c]\!] \ \wedge\ \mathtt{m} \in M(c)\})$;

[5] For the sake of simplicity in the presentation, in this chapter we restrict ourselves to methods with a single return type, that is, with a single outout parameter. The extension to methods with several output parameters is indeed straightforward, since output parameters are not relevant to dispatching.

2. $\forall\ i,\ 1 \leq i \leq n,\ \exists\ T_i \in ms(\{\ T \mid v_i \in [\![T]\!] \})$ *such that* $T_i \leq_T T_i^j$. $\square$

Because in our model objects may possibly belong to multiple most specific classes, the applicable method is in general not unique.

Example 6.2 *Consider the type hierarchy of Figure 6.1, where the following methods are defined:* $\mathtt{m}^A(E, F, F)$[6]*,* $\mathtt{m}^H(B, C, D)$*,* $\mathtt{m}^G(E, D, C)$*,* $\mathtt{m}^F(B, C, D)$[7]*.*

Consider moreover the invocation $o_1.\mathtt{m}(o_2, o_3, o_4)$ *where* $msc(o_1) = \{F, G\}$*,* $msc(o_2) = \{F, G\}$*,* $msc(o_3) = \{F\}$ *and* $msc(o_4) = \{F, G\}$*. The methods applicable for the invocation are* $\mathtt{m}^F$ *and* $\mathtt{m}^G$*. Indeed:*

- 1. $F \in msc(o_1), \mathtt{m} \in M(F)$*;*
  2. $G \in msc(o_2)$*, and* $G \leq_T B$*;* $F \in msc(o_3)$*, and* $F \leq_T C$*;* $F \in msc(o_4)$*, and* $F \leq_T D$*.*
- 1. $G \in msc(o_1), \mathtt{m} \in M(G)$*;*
  2. $G \in msc(o_2)$*, and* $G \leq_T E$*;* $F \in msc(o_3)$*, and* $F \leq_T D$*;* $G \in msc(o_4)$*, and* $G \leq_T C$*.* $\diamond$

In what follows we present conditions ensuring that a method invocation is type correct. Note that those conditions are static, that is, they can be tested at program compilation time. First of all, we introduce the following notation. Let c be a class and $\mathtt{m}$ be a method name; $s(\mathtt{m}, c) \in \mathcal{CI}$ denotes the class from which c inherits method $\mathtt{m}$. $s(\mathtt{m}, c)$ is the most specific superclass of c where method $\mathtt{m}$ is defined, that is $min_{\leq_T}\{\ c' \mid c \leq_T c' \wedge \mathtt{m} \in M(c')\}$. If method $\mathtt{m}$ is defined in several superclasses of c, not related by inheritance relationships, the resolution of name conflicts for multiple inheritance requires that in the definition of c an explicit qualification mechanism is used to state from which class method $\mathtt{m}$ is inherited. Let c^q be the qualified class, then $s(\mathtt{m}, c) = c^q$.

The following condition guarantees that a method invocation is type correct. Note that c_{st} is the class whose method $\mathtt{m}$ would be selected for execution if static dispatching (that is, early binding) were adopted.

Rule 6.2 *(Type Correctness of a Method Invocation) Let* $c_{st} = s(\mathtt{m}, static\_type(X))$ *be the class from which* $static\_type(X)$ *inherits method* $\mathtt{m}$*; let* $sign(\mathtt{m}, c_{st}) = T_1^{c_{st}} \times \cdots \times T_n^{c_{st}} \rightarrow T_r^{c_{st}}$ *be the signature of method* $\mathtt{m}$ *in class* c_{st}*. The invocation* $X.\mathtt{m}(X_1, \ldots, X_n)$ *is statically type correct if* $\forall\ i,\ 1 \leq i \leq n,\ static\_type(X_i) \leq_T T_i^{c_{st}}$*.* $\square$

[6] This notation is equivalent to $\mathtt{m} \in M(A)$ and $sign(\mathtt{m},\ A) = E \times F \times F \rightarrow \ldots$.

[7] Note that these method definitions respect the contravariance rule for method signature refinement

Condition 6.2 is exactly the condition that must be tested for static type checking systems based on selfish methods, where the binding object-class is exclusive (that is, objects cannot belong to several most specific classes). The condition ensures the existence of an applicable method also for models supporting multiple class direct membership, as stated by the following proposition.

Proposition 6.1 *Let $X.\mathtt{m}(X_1, \ldots, X_n)$ be a method invocation. If the invocation is statically type correct according to Condition 6.2, an applicable method for the invocation always exists.* $\square$

Proof Sketch: Let o be an object to which variable X is instantiated at run-time. Because of the relationship between static and dynamic types stated by Rule 6.1, a class c exists, with $c \in msc(o)$, such that $c \leq_T static\_type(X) \leq_T c_{st}$. Let c^* denote the most specific ancestor of c that specifies an implementation for method $\mathtt{m}$, that is, $c^* = ms(\{c_i \mid c_i \in \mathcal{CI}, c \leq_T c_i \leq_T c_{st}, \mathtt{m} \in M(c_i)\})$[8]. Then:

(i) $c^* = ms(\{c_i \mid \mathtt{m} \in M(c_i) \wedge o.i \in [\![c_i]\!] \})$, because $\mathtt{m} \in M(c^*)$ and $o.i \in [\![c^*]\!]$ (by $c \leq_T c^*$);

(ii) let $sign(\mathtt{m}, c^*)$ be $T_1^{c^*} \times \cdots \times T_n^{c^*} \rightarrow T_r^{c^*}$; by the contravariance rule for input arguments, since $c^* \leq_T c_{st}$, $T_i^{c_{st}} \leq_T T_i^{c^*}\ \forall\, i,\ 1 \leq i \leq n$, follows; moreover, by Condition 6.2, $static\_type(X_i) \leq_T T_i^{c_{st}}$, and thus $static\_type(X_i) \leq_T T_i^{c^*}$.

Thus, method in class c^* (that is, $\mathtt{m}^{c^*}$) is certainly applicable. $\square$

We remark that the tests for verifying whether a method invocation is type correct are performed statically (and thus without introducing any overhead at run-time). We can evaluate the cost for testing Condition 6.2 as follows:

1. the cost of determining c_{st} is at most equal to the height[9] of the type hierarchy $\mathcal{T}$;
2. the test to verify whether $static\_type(X_i) \leq_T T_i^{c_{st}}$ is performed n times, where n is the number of method $\mathtt{m}$ arguments.

Thus, since the cost for one test with respect to the partial relation $\leq_T$ is $O(1)$ [8], the total cost for verifying Condition 6.2 is $O(n + height(\mathcal{T}))$.

[8]As particular cases, c^* can be c if class c defines an implementation for method $\mathtt{m}$, whereas c^* can be c_{st} if no subclass of c_{st} redefines method $\mathtt{m}$. Note, however, that c^* is always defined.

[9]Given a type hierarchy $\mathcal{T}$, the *height* of the hierarchy, denoted as $height(\mathcal{T})$, is the maximum among the lengths of paths with origin in the root of $\mathcal{T}$.

6.3 Dispatching Rules

In this section, we deal with the problem of selecting a method implementation among the applicable ones. Each class in a type hierarchy may define a different implementation for the same method, and for each method invocation on an object, an implementation must be chosen among the most specific ones. Note that different implementations may return different results or may perform different updates on data.

According to one of the basic principles of object-orientation, when, because of subtype polymorphism [55], several method implementations are applicable to a method invocation, the implementation specified in the most specific class of the invocation receiver is executed, as it is the one that *most closely matches* the invocation. Thus, the most specialized behavior prevails, according to the classical late binding mechanism. However, in a model where an object is not characterized by a single most specific class, the choice of the method implementation that "most closely matches" the invocation is not obvious.

There are two different approaches to determine the implementation which most closely matches the invocation, among different implementations in different most specific classes of the object. The first approach, which we call **preferred class** approach, is based on the idea that each object has in each context a *preferred* class, among its most specific ones. Thus, each method invocation is dispatched choosing the implementation in the preferred class in the current context. This approach supports a context-dependent behavior, as the same method invocation may be dispatched differently, and thus may return different results and perform different updates, depending on the context where the method is invoked.

The second approach, which we call **argument specificity** approach, does not determine the preferred class of an object to dispatch a method invocation, rather it makes use of the other actual arguments of the method call, thus considering the method as a multi-method.

In the following subsections we illustrate these two approaches.

6.3.1 Preferred Class

According to this approach, a method invocation is dispatched by taking into account the context-dependent preferred class of the receiver object. As we have seen, in each expression, each object reference has a single static type. However static typing alone is not enough to assign a preferred class to each object in each expression for method dispatching. Indeed, the static type of the object may not belong to the set of most specific classes of the object. Consider for example (see Figure 1.1(a)) an expression where an object reference has the static type `person`; if at run-time the reference denotes an object belonging both to the class `noble` and to the class `female`, the context of the object reference does not help in choosing the preferred class. In those cases, we must use a total order on classes; the total order $\preceq_T$ we have introduced in Section 6.2 is thus used. We remark that those orders, that may be considered arbitrary and unpredictable by the user, are taken into account only when the context does not uniquely determine a preferred class for the object. The only alternative in these cases, apart from using these orders, would be to simply not dispatch the message, because it is *ambiguous*.

The preferred class approach is based on both static and dynamic information. The static information consists of the static type of the expression, whereas the dynamic information consists of the set of the most specific classes of the object (such classes, in fact, can only be determined at run-time). The total order on classes can be fixed and thus known statically or can, by contrast, be object-dependent and thus known only at run-time.

The following rule formalizes the preferred class dispatching method.

Rule 6.3 *(Preferred Class Dispatching Rule). Let* $o_1.\mathtt{m}(o_2, \ldots, o_n)$ *be a method invocation. The method invocation is dispatched as follows:*

method $\mathtt{m}$ *in class* c *is executed if* $c = min_{\prec_T}\{\, c_i \mid \mathtt{m}^{c_i}$ *is applicable and* $c_i \leq_T static\_type(o_1)\,\}$. □

Example 6.3 *Consider the type hierarchy of Figure 6.1 and method definitions of Example 6.2. Suppose moreover the total order on classes is as follows:*

$$H \prec_T G \prec_T F \prec_T E \prec_T D \prec_T C \prec_T B \prec_T A.$$

Consider finally the invocation $o_1.\mathtt{m}(o_2, o_3, o_4)$ *of Example 6.2. Then:*

- *if* $static\_type(o_1) = F$ *then* $\mathtt{m}^F$ *is executed;*
- *if* $static\_type(o_1) = C$ *then* $\mathtt{m}^G$ *is executed, since* $G \prec_T F$. ◇

The preferred class dispatching approach models context-dependent behavior. In particular, a given method invocation with a fixed set of parameters may produce different results (both in terms of results and data updates), though executed on the same database state, depending on the context of the receiver object reference in the expression containing the invocation.

Under the preferred class approach, any type correct method invocation can be dispatched. Moreover, when the preferred class dispatching strategy is used with a contravariant redefinition rule for method arguments, type correctness is ensured.

A possible implementation of the preferred class dispatch rule is reported in Figure 6.2. The cost of dispatching, if we consider the comparison of two types under the $\leq_T$ and $\preceq_T$

```
ĉ ← static_type(o₁)
foreach c ∈ msc(o₁) do
  begin
    1 if c ≤_T static_type(o₁) then
      begin
        2  c* ← min_{≤_T}{c' | c ≤_T c' ≤_T static_type(o₁) ∧ m ∈ M(c')}
        3 if method m^{c*} is applicable then
          4 if c* ≺_T ĉ then
            ĉ ← c*
      end
  end
return m^ĉ
```

Figure 6.2: Possible implementation of the preferred class dispatching rule

orders as a time constant operation [8], is in $O(length(path(c, static\_type(o_1))))$, for step 2, plus $O(n)$, with n arity of the method, for step 3, that is, checking the applicability. The process is repeated $|msc(o_1)|$ times. The total cost of the dispatching can then be estimated as $O(h \cdot (n + height(\mathcal{T})))$, being h the number of most specific classes to which an object can belong. Note that this estimation is very pessimistic, because, in most cases, an object is a direct member of very few classes, and the "distance" between static and dynamic types of an object is not the entire height of the inheritance hierarchy. Finally, we remark that, if order $\preceq_T$ is known at compile-time, classical selfish method dispatch optimization techniques (e.g., dispatch tables) can be used.

6.3.2 Argument Specificity

The second approach we consider does not take into account the preferred class of an object, rather it tries to determine the method implementation that most closely matches the invocation by taking into account the types of (all) the actual parameters of the invocation (in addition to the type of the receiver object). This approach is similar to multiple dispatching or multi-method approaches where the selection of the method to execute depends on the types of all the actual arguments of the invocation. In the preferred class approach only the type of the receiver object determines the method to execute and the other arguments only provide the actual values for the method arguments, however, they play no role in method selection. By contrast, in this approach, the method selection is based on the types of all arguments, the receiver as well as the other ones.

This approach can be regarded as fully dynamic, as opposed to the other, which is only partially dynamic. Indeed, in this approach dispatching is based only on run-time information, that is, the types of the actual parameters of the invocation. Moreover, whereas the previous approach models context-dependent behavior, the argument specificity approach ensures a notion of behavior identity. In particular, it ensures that a given method invocation, with a fixed set of actual parameters, executed on a given database state, returns the same results and produces the same database state, regardless of the expression in which the method invocation is contained. Finally, note that we use multiple dispatching only for choosing an implementation among the ones in sibling classes, and never to choose an implementation among the ones in a path in a given inheritance hierarchy. Our methods, indeed, are not really multi-methods [86] in that they are associated with classes. Thus, the "privileged receiver", though it is not the only one involved in dispatching, has higher priority with respect to other arguments, in that only implementations in classes that are most specific for the receiver are considered as "candidates" for the dispatching. Thus, the dispatching we propose here is not purely multiple in that we maintain a form of "privilege" for the receiver of the method: other arguments are taken into account only to choose among sibling implementations, in the different most specific classes of the receiver object.

To define the argument specificity dispatching rule a notion of *method specificity*, that is, an order $\prec_{\mathtt{m}}$ on methods, must be used. This order is based on the argument specificity (considered in the order from left to right[10]), and, when all the arguments are not comparable under $<_T$, on the total order of classes where the methods are defined. Such an order is exploited in choosing the method to be executed, among the applicable ones.

Definition 6.4 *(Method Specificity). Let* $\mathtt{m}^{c_i}$ *denote method* $\mathtt{m}$ *in class* c_i *and* $\mathtt{m}^{c_j}$ *denote method* $\mathtt{m}$ *in class* c_j *where* $sign(\mathtt{m}, c_i) = T_1^i \times \cdots \times T_n^i \rightarrow T_r^i$ *and* $sign(\mathtt{m}, c_j) = T_1^j \times \cdots \times T_n^j \rightarrow T_r^j$.

$\mathtt{m}^{c_i}$ *is more specific than* $\mathtt{m}^{c_j}$*, denoted as* $\mathtt{m}^{c_i} \prec_{\mathtt{m}} \mathtt{m}^{c_j}$*, if and only if:*

(i) $\exists\ k,\ 1 \leq k \leq n$, *such that* $T_k^i <_T T_k^j$ *and* $\forall\ h,\ 1 \leq h < k$, *neither* $T_h^i <_T T_h^j$ *nor* $T_h^j <_T T_h^i$ *hold*[11]*;*

or, if (i) does not hold,

(ii) $c_i \prec_T c_j$. □

Example 6.4 *Referring to the type hierarchy of Figure 6.1, to method definitions of Example 6.2 and to the total order* $\prec_T$ *of Example 6.3, the following method specificity relationships hold:*

[10]Note that this order corresponds to *argument order precedence* proposed in [10].

[11]Note that, as a particular case, it can be $T_h^i = T_h^j$.

- $\mathtt{m}^A \prec_{\mathtt{m}} \mathtt{m}^H$, $\mathtt{m}^A \prec_{\mathtt{m}} \mathtt{m}^F$, $\mathtt{m}^G \prec_{\mathtt{m}} \mathtt{m}^H$, *and* $\mathtt{m}^G \prec_{\mathtt{m}} \mathtt{m}^F$, *since* $E <_T B$;
- $\mathtt{m}^A \prec_{\mathtt{m}} \mathtt{m}^G$, *since* $F <_T D$;
- $\mathtt{m}^H \prec_{\mathtt{m}} \mathtt{m}^F$, *since* $H \prec_T F$.

Thus, the method specificity order is as follows: $\mathtt{m}^A \prec_{\mathtt{m}} \mathtt{m}^G \prec_{\mathtt{m}} \mathtt{m}^H \prec_{\mathtt{m}} \mathtt{m}^F$.

$\diamond$

The following rule formalizes the argument specificity dispatching method.

Rule 6.4 *(Argument Specificity Dispatching Rule). Let* $o_1.\mathtt{m}(o_2, \ldots, o_n)$ *be a method invocation, and let* $\prec_{\mathtt{m}}$ *be the considered method specificity order. The method invocation is dispatched as follows:*

method $\mathtt{m}$ *in class* c *is executed if* $\quad \mathtt{m}^c = min_{\prec_{\mathtt{m}}}\{\ \mathtt{m}^{c'} \mid \mathtt{m}^{c'}$ *is applicable* $\}$ $\square$

The most specific among the applicable methods is thus executed.

Example 6.5 *Consider the method invocation of Example 6.3 and the method specificity order of Example 6.4. Since both* $\mathtt{m}^F$ *and* $\mathtt{m}^G$ *are applicable, and* $\mathtt{m}^G \prec_{\mathtt{m}} \mathtt{m}^F$, *method* $\mathtt{m}^G$ *is selected.* $\diamond$

The argument specificity approach ensures the identity of behavior of a method invocation. According to this dispatching rule, the class to which a given message is dispatched does not depend on the context of the message receiver in the expression containing the invocation. Under the argument specificity approach, moreover, any type correct method invocation can be dispatched. The argument specificity approach, however, does not ensure type correctness, as we will discuss in Section 6.5.

Next section shows how the argument specificity dispatching rule can be *efficiently* supported.

6.4 Implementation Technique for the Argument Specificity Dispatching Rule

In this section we show how methods can be efficiently dispatched according to the argument specificity approach by means of lookup automata (LUA). We first informally introduce the technique, by means of examples, and then we present LUA construction and simulation algorithms. Finally, we address efficiency issues.

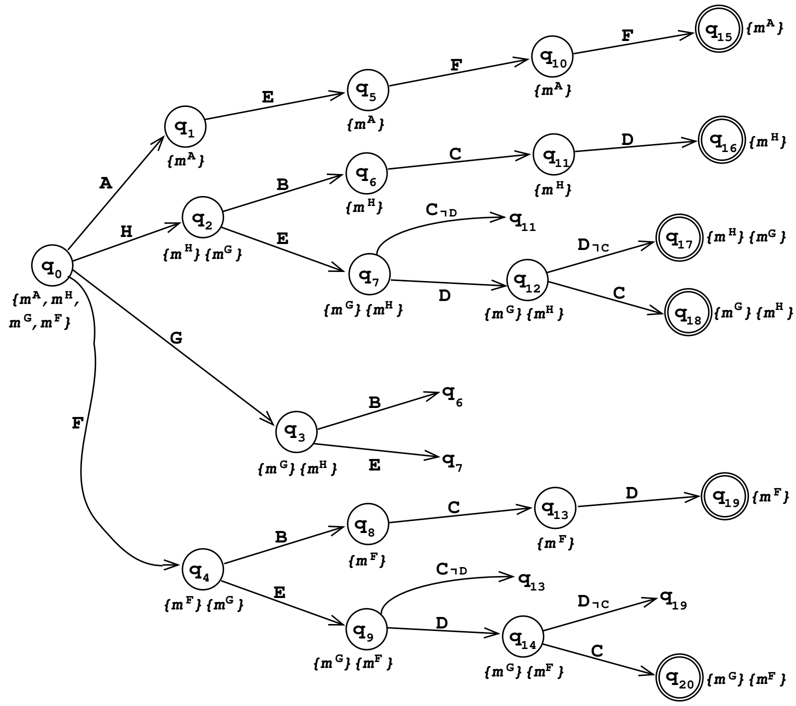

Figure 6.3: Lookup automaton for type hierarchy of Figure 2

6.4.1 Overview of the Proposed Approach

Our technique is based on a lookup automaton that simulates dynamic dispatching. Given a method invocation, it selects the method implementation (among the applicable ones) whose formal arguments most closely match the actual ones of the invocation.

Definition 6.5 *(Lookup Automaton). A LUA is a deterministic finite automaton [11] defined as a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states; Σ is a finite set of input symbols (alphabet); δ is a state transition function, which is a "mapping": $Q \times \Sigma \rightarrow Q$; $q_0 \in Q$ is the initial state; and $F \subseteq Q$ is the set of final states.* □

The LUA we introduce has three important properties: its simulation does not require backtracking; it has a minimal number of states; its construction can be performed without using the total order $\preceq_T$. The first and the second property are important because they give an upper bound to temporal and spatial complexity, respectively (see Section 6.4.3); the third property, instead, is important because it allows the use of a total order which is known only at run-time and which can thus be object-dependent.

Figure 6.3 shows the final result of the LUA construction for the type hierarchy and methods of the running example, where final states are characterized by bold circles. In order to avoid crossing arrows in the graphical representation, the states without circles refer to the corresponding circled ones. In the LUA, transitions are labeled by types, whereas with each state a *method precedence order*, that is, a list of sets of methods, is associated. This list represent the applicable methods, ordered according to their specificity. Suppose that the list $S_1 S_2 \cdots S_l$ (where S_i are sets of methods) is associated with state q, and that method $\mathtt{m}^{c_1} \in S_i$ and $\mathtt{m}^{c_2} \in S_j$: if $i < j$, then method $\mathtt{m}^{c_1}$ at state q has higher priority than $\mathtt{m}^{c_2}$; if $i = j$, then $\mathtt{m}^{c_1}$ and $\mathtt{m}^{c_2}$ at state q have the same priority. For example, consider the list $\{\mathtt{m}^A, \mathtt{m}^H, \mathtt{m}^G, \mathtt{m}^F\}$ at state q_0 in Figure 6.3: methods $\mathtt{m}^A$, $\mathtt{m}^H$, $\mathtt{m}^G$ and $\mathtt{m}^F$ have the same priority, while if we consider the list $\{\mathtt{m}^H\}\{\mathtt{m}^G\}$ at state q_2 then method $\mathtt{m}^H$ has higher priority priority than method $\mathtt{m}^G$.

The basic idea in building the LUA is the following: in order to reduce its size, the number of states introduced should be as small as possible. Informally, if some states have the same precedence order, they will be "merged" together. In order to easily catch the idea, we first state how to simulate this LUA; then, we describe the overall approach to construct it.

6.4.1.1 Lookup Automaton Simulation

The basic idea in the simulation of the LUA is that, from a state q at level i in the automaton a transition labeled by a type T is chosen if T is the most specific, among the types labeling transitions following state q, type to which the i-th actual parameter of the invocation belongs. Since in our model multiple class direct membership is supported, there can be two types labeling transitions following a state q, that are not comparable under the subtype relation, and such that the i-th parameter of the invocation belongs to both. This situation is handled differently depending on whether or not $i = 1$ (that is, the involved argument is or not the distinguished parameter). This reflects the fact that we maintain a form of privilege for the first argument. Thus, in the first level of the LUA, if more than one transition is labeled by a most specific class to which the receiver object belongs, then the transition labeled by the type which precedes the others in the $\prec_T$ order is followed. This reflects condition (ii) of the definition of method specificity (Definition 6.4). In other levels, transitions labeled by expressions of the form $X_{\neg Y}$ are employed. The notation $X_{\neg Y}$ denotes that the transition labeled by $X_{\neg Y}$ can be chosen only if the class X is one of the most specific to which the considered object belongs, among those labeling a transition following the current state, <u>and</u> the considered object is not a member of class Y. Thus, the notation $X_{\neg Y}$ is a compact form which is equivalent to $\{X \ \wedge \ \neg Y\}$.

Consider the invocation $o_1.\mathtt{m}(o_2, o_3, o_4)$ of the running example. The LUA simulation consists of the following four steps:

1. Initially, we are at start state q_0. There are four types A, H, G and F following

state q_0. Among these, only F and G are candidates for being selected because $msc(o_1) = \{F, G\}$. Between F and G, type G is chosen because $G \prec_T F$. Indeed, as we have anticipated, when we must choose among sibling types (that is, types that are not comparable under the subtype relation $<_T$) in the considered type hierarchy, in the first level of the automaton, the most specific type with respect to the total order $\prec_T$ is chosen. Thus, at the first step, transition $\delta(q_0, G) = q_3$ is selected.

2. At state q_3, two types B and E follow; we select transition $\delta(q_3, E) = q_7$ because E is the most specific supertype of at least one type belonging to $msc(o_2)$ (which, in this case, is G); E is indeed more specific than B (that is, $E <_T B$).

3. At state q_7, on the contrary, two types $C_{\neg D}$ and D follow; we select transition $\delta(q_7, D)$ $= q_{12}$ because F, which is a subtype of D, belongs to $msc(o_3)$.

4. Finally, also at state q_{12} two types $D_{\neg C}$ and C follow; we select transition $\delta(q_{12}, C)$ $= q_{18}$ because, similarly to what happened at the third step, F, which is a subtype of C, belongs to $msc(o_4)$.

At the final state q_{18}, method $\mathtt{m}^G(E, D, C)$ is the most specific applicable method for the given invocation.

Note that, during LUA simulation, the use of the supplementary information $\prec_T$ is needed. Besides, the number of state transitions processed is equal to the method arity, which is four in this case.

Before concluding the description of LUA simulation, we point out that the $\neg$ operator applies to the type hierarchy (and thus also to subtypes); for example, the automaton edge labeled with $C_{\neg D}$ of Figure 6.3 is only a compact form which is equivalent to $\{C \land \neg(D \land E \land F \land G \land H)\}$. For this reason, in the previous simulation, at the third and at the fourth step we select, respectively, the transitions labeled by D (instead of $C_{\neg D}$) and by C (instead of $D_{\neg C}$) have been selected.

6.4.1.2 Lookup Automaton Construction

The LUA in Figure 6.3 is constructed as follows. *Initially*, the initial state q_0 is created, where four methods $\mathtt{m}^A$, $\mathtt{m}^H$, $\mathtt{m}^G$ and $\mathtt{m}^F$ are applicable with the same priority and thus at q_0 the method precedence order is $\{\mathtt{m}^A, \mathtt{m}^H, \mathtt{m}^G, \mathtt{m}^F\}$. At next level of the LUA (which we refer to as the *first level* because we assume the level of the state q_0 to be level 0), the set of the types of the first argument for the methods $\mathtt{m}^A$, $\mathtt{m}^H$, $\mathtt{m}^G$ and $\mathtt{m}^F$ is $\{A, H, G, F\}$[12]; therefore, we consider the state transitions $\delta(q_0, A)$, $\delta(q_0, H)$, $\delta(q_0, G)$ and $\delta(q_0, F)$. At

[12]Note that methods are associated with classes and for this reason, it is impossible that a class contains two different implementations for the same method.

state $\delta(q_0, A)$, only method $\mathtt{m}^A$ is applicable, so that we create a new state $q_1 = \delta(q_0, A)$ with precedence order $\{\mathtt{m}^A\}$; at state $\delta(q_0, H)$, two methods $\mathtt{m}^H$ and $\mathtt{m}^G$ are applicable, but $\mathtt{m}^H$ has higher priority than $\mathtt{m}^G$ because $H \prec_T G$: for this reason the precedence order is $\{\mathtt{m}^H\}\{\mathtt{m}^G\}$. Since there is no state with this precedence order, a new state $q_2 = \delta(q_0, H)$ with precedence order $\{\mathtt{m}^H\}\{\mathtt{m}^G\}$ is created. At state q_2 there are two applicable methods (and not only one) because, among the defined methods, there are two methods ($\mathtt{m}^H$ and $\mathtt{m}^G$) whose first arguments H and G are not comparable under the subtype relation $<_T$[13]. At state q_2 the method precedence order is $\{\mathtt{m}^H\}\{\mathtt{m}^G\}$ because $H \prec_T G$. According to Definition 6.4, if argument types do not help in selecting the method (because they are not comparable under $<_T$), the most specific method under the total order $\prec_T$ is selected.

In a similar way, the new states $q_3 = \delta(q_0, G)$ (with precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^H\}$) and $q_4 = \delta(q_0, F)$ (with precedence order $\{\mathtt{m}^F\}\{\mathtt{m}^G\}$) are created. Consider now the situation at the *second level* of the LUA. At state q_1, only $\mathtt{m}^A$ are applicable, thus only state transition $\delta(q_1, E)$ is considered (E is the type of the second argument of $\mathtt{m}^A$). A new state $q_5 = \delta(q_1, E)$ with precedence order $\{\mathtt{m}^A\}$ is thus created. At state q_2, $\mathtt{m}^H$ and $\mathtt{m}^G$ are applicable and thus state transitions $\delta(q_2, B)$ and $\delta(q_2, E)$ must be considered: they lead to the creation of new states q_6 (with precedence order $\{\mathtt{m}^H\}$) and q_7 (with precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^H\}$), respectively. Also at state q_3 two state transitions must be considered (because $\mathtt{m}^G$ and $\mathtt{m}^H$ are applicable): $\delta(q_3, E)$ and $\delta(q_3, B)$. Since state $\delta(q_3, B)$ has precedence order $\{\mathtt{m}^H\}$, which is also the precedence order of state q_6, whereas state $\delta(q_3, E)$ has precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^H\}$, as state q_7, we define $\delta(q_3, B) = q_6$ and $\delta(q_3, E) = q_7$ without introducing new states. At state q_4, instead, we must consider the transitions $\delta(q_4, B)$ and $\delta(q_4, E)$ (because $\mathtt{m}^F$ and $\mathtt{m}^G$ are applicable) which leads to the creation of new states q_8 (with precedence order $\{\mathtt{m}^F\}$) and q_9 (with precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^F\}$), respectively. Now consider the situation at the *third level* of the LUA. At state q_5, only $\mathtt{m}^A$ is applicable, thus only state transition $\delta(q_5, F)$ is considered (F is the type of the third argument of $\mathtt{m}^A$). A new state $q_{10} = \delta(q_5, F)$ with precedence order $\{\mathtt{m}^A\}$ is thus created. Similarly, at state q_6 only state transition $\delta(q_6, C)$ is considered; it leads to the creation of new state q_{11} with precedence order $\{\mathtt{m}^H\}$.

Let us point out some aspects concerning the definition of state q_7; $\mathtt{m}^G$ and $\mathtt{m}^H$ are applicable and thus we should consider the state transitions $\delta(q_7, D)$ and $\delta(q_7, C)$, respectively. The former transition $\delta(q_7, D)$ leads to the creation of new state q_{12} with precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^H\}$; the latter one, by contrast, presents a new operator: indeed we must "mark" type C with the notation $\neg D$ which denotes that the transition $\delta(q_7, C_{\neg D})$ may be applied only if, among types of the corresponding actual parameter, there is the type C but there

[13]Since argument specificity dispatching rule privileges the distinguished parameter (that is, the first formal argument of a method), starting from the second level of the LUA, the only cases where the method order precedence is a list with more than one method are those in which there are methods whose first arguments are sibling types. In all other cases, because of the privilege on the first formal argument, each state is associated with only one applicable method.

is not the type D. Now, we state when the $\neg$ operator is inserted in the LUA.

> **$\neg$ Operator**. The $\neg$ operator is inserted in the LUA if the following conditions hold:
>
> - there are two transitions, following a state q_i, labeled with two sibling types;
> - in the state q_j destination of the transition, the method precedence order is different from the one associated with the previous state q_i.
>
> The type to be "marked" with $\neg$ operator on the LUA is the one which leads to state q_j (that is in the transition which determines a change in priority order for method selection).

Note that the $\neg$ operator allows to keep the method list order consistent with the method specificity notion (Definition 6.4).

Note finally that transition $\delta(q_7, C_{\neg D})$ leads to the creation of a new state $\delta(q_7, C_{\neg D}) = q_{11}$ with precedence order $\{\mathtt{m}^H\}$. By continuing with this procedure, we first consider the situation at the *fourth and last level* of the LUA and then the construction of the LUA will be completed, as Figure 6.3 shows. The constructed LUA presents six final states (q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}), with each of them a most specific applicable method is associated.

The LUA construction technique we present overcomes spatial complexity problems in performing dispatching. Indeed, the LUA construction and minimization are simultaneously addressed. The procedure to reduce the size of the LUA is performed during LUA construction. Consequently, our approach prevents a large memory overhead, because unnecessary states of the LUA are never created.

6.4.2 Construction and Simulation Algorithms

In this subsection, we present algorithms to *construct* and *simulate* the LUA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ which depend on the type hierarchy and on the set of methods $\mathbf{M}$ defined on it.

First of all, we introduce the terminology, notations and definitions which we will use in the algorithms. In the following we call *method precedence order* a list $\sigma = S_1 S_2 \cdots S_l$, $l>0$, where $S_i \in 2^{\mathbf{M}}$ $(i \in \{1, \ldots, l\})$. Besides, σ must be such that if $i \neq j$ then $S_i \cap S_j = ?$ $(i, j \in \{1, \ldots, l\})$. The list $S_1 S_2 \cdots S_l$ is denoted by $\prod_{i=1}^{l} S_i$ and each S_i $(i \in \{1, \ldots, l\})$ is called a *factor* of σ.

Definition 6.6 *($\sqcap$ Operator). Let $\sigma = S_1 S_2 \cdots S_l$ and $\rho = R_1 R_2 \cdots R_p$ be two method precedence orders. We define $\sigma \sqcap \rho = \prod_{i=1}^{l} \prod_{j=1}^{p} (S_i \cap R_j)$ where $\sigma = \prod_{i=1}^{l} S_i$ and $\rho = \prod_{j=1}^{p} R_j$.* □

Definition 6.7 *(Function domain). Let $S_1 S_2 \cdots S_l$ be a method precedence order. We define*

$domain(S_1 S_2 \cdots S_l) = \bigcup_{i=1}^{l} S_i$. □

Definition 6.8 *(Boolean Function is_equal). Let $\sigma = S_1 S_2 \cdots S_l$ and $\rho = R_1 R_2 \cdots R_p$ be two method precedence orders. Then, $is\_equal(\sigma, \rho) =$ `True` iff $\sigma = \rho$ and $S_i = R_i$ ($i \in \{1, \ldots, l\}$).* □

We introduce now some functions which we will use in this subsection.

- Function τ_k: $\mathbf{M} \rightarrow \mathcal{T}$ maps a method to its k-th argument type ($k \in \{1, \ldots, n\}$), i.e.,

  $$\tau_k(\ \mathtt{m}^{T_1}(T_2, \ldots, T_k, \ldots, T_n)\ ) = T_k, \text{ if } \mathtt{m}^{T_1}(T_2, \ldots, T_k, \ldots, T_n) \in \mathbf{M}.$$

  For the sake of convenience, sometimes we extend the argument domain of τ_k from only one method to a set of methods, i.e.,

  $$\tau_k(\mathbf{M}^*) = \{\ \tau_k(\mathtt{m}) \mid \mathtt{m} \in \mathbf{M}^*\ \} \quad \text{if } \mathbf{M}^* \in 2^{\mathbf{M}}.$$

- The inverse function τ_k^{-1}: $\mathcal{T} \rightarrow 2^{\mathbf{M}}$ maps a type T to the set of methods whose k-th argument type is T, i.e.,

  $$\tau_k^{-1}(T) = \{\ \mathtt{m} \in \mathbf{M} \mid \tau_k(\mathtt{m}) = T\}.$$

- Finally, function μ_k maps a type $T \in \mathcal{T}$ and a set of types $\mathbf{S} \in 2^{\mathcal{T}}$ into a method precedence order, i.e.,

  $$\mu_k(T, \mathbf{S}) = \tau_k^{-1}(T) \cdot \prod_{i=1}^{q} \tau_k^{-1}(T_i)$$

  where $(T_1, \ldots, T_q)$ is a type precedence order (established by $<_T$) over $\mathbf{S}$ for T.

We introduce moreover the following definition.

Definition 6.9 *(Function lst). For any $T \in \mathcal{T}$ and $\mathbf{S} \subseteq \mathcal{T}$, we define the set of* least supertypes *for T with respect to $\mathbf{S}$ as $lst(T, \mathbf{S}) = \{\ S \in \mathbf{S} \mid T \leq_T S$ and $\nexists\ S' \in \mathbf{S}$ s.t. $T \leq_T S' <_T S\ \}$* □

For example, referring the type hierarchy shown in Figure 6.1, $lst(G, \{A, C, D\}) = \{C, D\}$.

In the following, let $level(q)$ denote the length of a path from the initial state q_0 to state q (i.e., the number of transitions from state q_0 to state q). Moreover, $\Sigma(q) = \{ T \in \Sigma \mid \delta(q, T)$ is defined $\}$.

Now, given all necessary definitions, we can present the algorithms to *construct* and to *simulate* the LUA.

6.4.2.1 LUA Construction

The LUA construction mechanism can be implemented by means of algorithm *Construct* (shown in Figure 6.4) which, in its body, calls two algorithms (*Build_State_Level_1*, shown in Figure 6.5 and *Build_Next_State*, shown in Figure 6.6) for the state construction, respectively, of the LUA first level and of the LUA other levels. In these algorithms, each state $q \in Q$ is represented by a record with a field called *ord* which contains method precedence order at state q. Besides, because some types must be "marked" during construction, the state transition function contains a further argument (of type `bool`) which is equal to 1 if the type associated to transition that are considering has not been "marked"; 0 otherwise.

In the algorithm, *Construct*, Q' represents the set of all states built at level k ($k \in \{1, \ldots, n\}$). The subprocedure *Build_State_Level_1*(q_0,Q'') constructs all possible state transition $\delta(q_0, T)$ (where $T \in \mathcal{T}$) for the first level of the LUA, while the subprocedure *Build_Next_State*(k,q,Q'') constructs them for the other levels of the LUA (that is, from the second to the n-th); after every subprocedure has been executed, the argument Q'' represents the set of all states built so far at level $k+1$ of the LUA.

Let us explain the above algorithms in the context of the example shown in Figure 6.3. Suppose that we have already constructed all states and transitions at level 1 of the LUA, where (after execution of algorithm *Build_State_Level_1*(q_0,Q'')) $Q' = \{q_1, q_2, q_3, q_4\}$. Now we are going to construct states and transitions at level 2 (`for` with $k = 2$ of algorithm *Construct*). The executed steps lead to the following situation:

- Q'' = ?;
- procedure *Build_Next_State*$(2, q_1, Q'')$ is called, where:
  - $q_1.ord = \{\mathtt{m}^A\}$, and thus $\mathbf{T'} = \tau_2(domain(q_1.ord)) = \tau_2(\{\mathtt{m}^A\}) = \{E\}$ and $\mathbf{T}^* = \mathbf{T'}$;
  - in the first loop body of first `foreach`, $ord = \mu_2(E, \{E\}) \sqcap q_1.ord = \tau_2^{-1}(E) \sqcap q_1.ord = \{\mathtt{m}^A, \mathtt{m}^G\} \sqcap \{\mathtt{m}^A\} = \{\mathtt{m}^A\}$; therefore a new state q_5 with precedence order $\{\mathtt{m}^A\}$ is created, $\delta(q_1, E) = q_5$ and now $Q'' = \{q_5\}$;

```
procedure Construct();
  begin
    "create the initial state q_0";
    q_0.ord ← M;
    Q ← {q_0};
    Σ ← ?;
    Q'' ← ?;
    Build_State_Level_1(q_0, Q'');

    Q ← Q ∪ Q'';
    Q' ← Q'';
    for k := 2 to n do
      begin
        Q'' ← ?;
        foreach q ∈ Q' do Build_Next_State(k, q, Q'');
        Q ← Q ∪ Q'';
        Q' ← Q'';
      end
    F ← Q';
  end
```

Figure 6.4: LUA construction algorithm

```
procedure Build_State_Level_1(q_0,var Q'');
  begin
    T' ← τ_1(q_0.ord);
    Σ ← Σ ∪ T';
    foreach T ∈ T' do
      if "T is comparable under <_T with all types in T' − {T}"
        then
          begin
            "create a new state q_new";
            Q'' ← Q'' ∪ {q_new};
            q_new.ord ← {m^T};
            δ(q_0, T) ← q_new;
          end;
        else
          begin
            let T, T_1,..., T_p be the types that are not comparable under <_T;
            "create a new state q_new";
            Q'' ← Q'' ∪ {q_new};
            q_new.ord ← {m^T}{m^T_1, ..., m^T_p};
            δ(q_0, T) ← q_new;
          end;
  end
```

Figure 6.5: Algorithm that builds states of LUA first level

```
procedure Build_Next_State(k,q,var Q'');
  begin
    T' ← τ_k(domain(q.ord))
    Σ ← Σ ∪ T';
    let T* be the types of T' that are comparable under <_T;
    foreach T ∈ T* do
      begin
        ord ← μ_k(T,T*) ⊓ q.ord;
        /* if the state is final the most specific method is chosen */
        if k = n then ord ← head(ord);
        if ∃ q' ∈ Q'' t.c. is_equal(q'.ord, ord)
          then δ(q, T) ← q'; /* the state exists and it is not created */
          else
            begin
              "create a new state q_new";
              Q'' ← Q'' ∪ {q_new};
              q_new.ord ← ord;
              δ(q, T) ← q_new;
            end;
      end;
    /* for the types that are not comparable under <_T */
    T ← τ_k(head(q.ord));
    "create a new state q_new";
    Q'' ← Q'' ∪ {q_new};
    q_new.ord ← q.ord;
    δ(q, T) ← q_new;
    foreach T' ∈ T' − { T* ∪ {T} } do /* ¬ operator */
      begin
        "set" T';
        ord ← μ_k(T,T' − { T* ∪ {T} }) ⊓ q.ord;
        if ∃ q' ∈ Q'' t.c. is_equal(q'.ord, ord) then δ(q, T') ← q';
        else
          begin
            "create a new state q_new";
            Q'' ← Q'' ∪ {q_new};
            q_new.ord ← ord;
            δ(q, T') ← q_new;
          end;
      end;
  end
```


Figure 6.6: Algorithm that builds states from the second level of the LUA

- procedure *Build_Next_State*$(2, q_2, Q'')$ is called, where:
  - $q_2.ord = \{\mathtt{m}^H\}\{\mathtt{m}^G\}$, and consequently $\mathbf{T'} = \tau_2(domain(q_2.ord)) = \tau_2(\{\mathtt{m}^H, \mathtt{m}^G\}) = \{B, E\}$ and $\mathbf{T}^* = \mathbf{T'}$;
  - thus, the loop body of first `foreach` is executed twice: the first time, with $T = B$, $ord = \mu_2(B, \{B, E\}) \sqcap q_2.ord = \tau_2^{-1}(B) \sqcap q_2.ord = \{\mathtt{m}^H, \mathtt{m}^F\} \sqcap \{\mathtt{m}^H\}\{\mathtt{m}^G\} = \{\mathtt{m}^H\}$; therefore a new state q_6 with precedence order $\{\mathtt{m}^H\}$ is created, $\delta(q_2, B) = q_6$ and now $Q'' = \{q_5, q_6\}$. The second time, with $T = E$, $ord = \mu_2(E, \{B, E\}) \sqcap q_2.ord = \tau_2^{-1}(E)\ \tau_2^{-1}(B) \sqcap q_2.ord = \{\mathtt{m}^A, \mathtt{m}^G\}\ \{\mathtt{m}^H, \mathtt{m}^F\} \sqcap \{\mathtt{m}^H\}\{\mathtt{m}^G\} = \{\mathtt{m}^G\}\{\mathtt{m}^H\}$; therefore a new state q_7 with precedence order $\{\mathtt{m}^G\}\{\mathtt{m}^H\}$ is created, $\delta(q_2, E) = q_7$ and now $Q'' = \{q_5, q_6, q_7\}$.

Then, procedures *Build_Next_State*$(2, q_3, Q'')$ and *Build_Next_State*$(2, q_4, Q'')$ are always called with $Q'' = \{q_5, q_6, q_7\}$ (because no new state has been created). At the end of the creation of states and transitions of second level of the LUA, the new states assigned to Q'' are $\{q_5, q_6, q_7, q_8, q_9\}$.

6.4.2.2 LUA Simulation

Till now, we have described our approach to the construction of the LUA for a determined set of methods $\mathbf{M}$. Now, given an invocation $o_1.\mathtt{m}(o_2, \ldots, o_n)$, we must specify how to implement the mechanism simulating the LUA itself. In order to reduce the space-complexity, the LUA does not identify *all* possible dispatching cases. Therefore, given an invocation $o_1.\mathtt{m}(o_2, \ldots, o_n)$ and the relative LUA, it is not possible to directly use a standard automaton simulation technique. Rather, a special algorithm to simulate the LUA is necessary, where the knowledge of the total order relation $\preceq_T$ is used.

The task of the LUA simulation is to find a final state q such that $q.ord$ is equal to the first factor of method precedence order (in $\mathbf{M}$) whose n arguments are *supertypes*, respectively, of *at least* a type belonging to $msc(o_i)$ $(i \in \{1, \ldots, n\})$. We can accomplish this task by tracing a list of states $q_{i_0}, \ldots, q_{i_n}$ where $q_{i_0} = q_0$ and $q_{i_k}.ord$ denotes the method precedence order (always in $\mathbf{M}$) whose first k arguments are *supertypes*, respectively, of *at least* a type belonging to $msc(o_i)$ $(i \in \{1, \ldots, k\})$.

In the algorithm, the following relation is used :

$$q_{i_{k+1}} = \delta(q_{i_k}, T) \quad \text{where } T \in lst(T^*, \Sigma(q_{i_k})) \text{ and } T^* \in msc(o_{k+1}).$$

The algorithm *Simulate* (shown in Figure 6.7) takes as input the arguments $o_1, \ldots, o_n$ and then returns either the most specific applicable method (if exists any) in $\mathbf{M}$ or the

```
function $Simulate(o_1, \ldots, o_n)$ return $\mathbf{M}^+$;
  begin
    $q \leftarrow q_0$;
    $\mathbf{T}' \leftarrow ?$;
    foreach $T^* \in msc(o_1)$ do $\mathbf{T}' \leftarrow lst(T^*, \Sigma(q))$;
    if $\mathbf{T}' = ?$ then  return $dummy$;
    else
      begin
        $T \leftarrow min_{\prec_T} \mathbf{T}'$;
        $q \leftarrow \delta(q, T)$;
      end;
    for $k := 1$ to $n$-1 do
      begin
        $\mathbf{T}' \leftarrow ?$;
        foreach $T^* \in msc(o_{k+1})$ do $\mathbf{T}' \leftarrow lst(T^*, \Sigma(q))$;
        if $\mathbf{T}' = ?$ then  return $dummy$;
        else
          begin
            if $\mathbf{T}' = \{T\}$ then $q \leftarrow \delta(q, T)$;
            else
              begin
                $T \leftarrow \tau_{k+1}(head(q.ord))$;
                if $\exists\ T' \in msc(o_{k+1})$ s.t. $T' \leq_T T$ then $q \leftarrow \delta(q, T)$;
                else
                  begin
                    $T'' \leftarrow min_{\prec_T} \{\ T^\# \mid T^\# = \tau_1(tail(q.ord))\ \}$;
                    /* $T''$ is a "marked" type */
                    $q \leftarrow \delta(q, T'')$;
                  end;
              end;
          end;
      end;
  end
```

Figure 6.7: LUA simulation algorithm

predefined error-handling symbol, denoted by *dummy*, if there is no applicable method in **M**. Besides, we denote with $\mathbf{M}^+$ the set $\mathbf{M} \cup \{dummy\}$.

Let us explain the simulation algorithm referring to the invocation of the running example. The transitions to reach final state, that is, the most specific applicable method, are four (because the arity of method `m` is four). They are as follows:

1. $msc(o_1) = \{F, G\}$; therefore $lst(F, \Sigma(q_0)) = lst(F,\{A,H, G,F\}) = \{F\}$ and $lst(G, \Sigma(q_0)) = lst(G,\{A,H, G,F\}) = \{G\}$ are computed. Then the values $\{F, G\}$ are assigned to $\mathbf{T}'$ and because $G \prec_T F$ at the first step the transition $q = \delta(q_0, G) = q_3$ is selected;

2. the second `for` is iterated three times:

   - $msc(o_2) = \{F, G\}$; therefore $lst(F, \Sigma(q_3)) = lst(F, \{B, E\}) = ?$ and $lst(G, \Sigma(q_3)) = lst(G, \{B, E\}) = \{E\}$ are computed. Then the value $\{E\}$ is assigned to $\mathbf{T}'$ and thus, at the second step, the transition $q = \delta(q_3, E) = q_7$ is selected;
   - $msc(o_3) = \{F\}$ and thus $lst(F, \Sigma(q_7)) = lst(F, \{C, D\}) = \{C, D\}$ is computed. Then the values $\{C, D\}$ are assigned to $\mathbf{T}'$; because C and D are not comparable under $<_T$, the value $\tau_3(head(q_7.ord)) = \tau_3(head(\{\mathtt{m}^G\}\{\mathtt{m}^H\})) = \tau_3(\{\mathtt{m}^G\}) = D$ is assigned to T; besides in $msc(o_3)$ there is a type (F) which is a subtype of D and therefore, at the third step, the transition $q = \delta(q_7, D) = q_{12}$ is selected;
   - finally, $msc(o_4) = \{F, G\}$; therefore $lst(F, \Sigma(q_{12})) = lst(F, \{C, D\}) = \{C, D\}$ and $lst(G, \Sigma(q_{12})) = lst(G, \{C, D\}) = \{C, D\}$ are computed. Then the values $\{C, D\}$ are assigned to $\mathbf{T}'$; because C and D are not comparable under $<_T$, the value $\tau_4(head(q_{12}.ord)) = \tau_4(head(\{\mathtt{m}^G\}\{\mathtt{m}^H\})) = \tau_4(\{\mathtt{m}^G\}) = C$ is assigned to T; besides in $msc(o_4)$ there is a type (in this case F) which is a subtype of C and therefore, at the fourth and last step, the transition $q = \delta(q_{12}, C) = q_{18}$ is selected.

For these reasons the method $\mathtt{m}^G(E, D, C)$ is selected, as we have already verified during the informal description of the algorithm and coherently on what argument specificity dispatching rule establishes (see Example 6.5).

6.4.3 Efficiency

In this subsection we evaluate the temporal and spatial complexity of the algorithms presented in the previous subsection.

6.4.3.1 Temporal Complexity

We have already stated that, given a method invocation $o_1.\mathtt{m}(o_2, \ldots, o_n)$, the number of transitions executed is never greater than n. Besides, during simulation, the choice of the transition to be executed depends on the search, among the most specific types to which each o_i ($i \in \{1, \ldots, n\}$) belongs, of a type for which there is a transition outgoing from the current state. Therefore, the total cost is the following:

$$O(n \cdot h \cdot d)$$

where n = arity of invoked method;
h = number of most specific classes to which an object o_i can belong;
d = number of definitions of the invoked method.

The operation $lst(T^*, \Sigma(q))$ has a cost equal to $|\Sigma(q)|$ and for each q, $|\Sigma(q)| \leq d$ holds. Such an operation is executed for each $T^* \in msc(o_i)$, i.e. $|msc(o_i)|$ times. Besides, for each o_i, $|msc(o_i)| \leq h_i$ holds, where h_i is the maximum number of types that are not comparable under $<_T$ in the hierarchy $\mathcal{H}_i$ to which o_i belongs. Since $h = max_{\mathcal{H}_i}\{h_i\}$, then $h_i \leq h$. Therefore, since the previous operations are executed for each parameter of the operation, the total cost of simulation results $O(n \cdot h \cdot d)$.

In general, however, we can notice that in most applications, a large number of objects will not belong to many most specific classes, and thus $|msc(o_i)|$ is in general very low ($\simeq 1$). Therefore the given evaluation (where $|msc(o_i)|$ is bound by h) is a pessimistic evaluation.

6.4.3.2 Spatial Complexity

For what concerns space-complexity, our algorithm is also very efficient. The size of the LUA is kept minimal during LUA construction itself, by merging together the states (at the same level) with the same precedence order. Besides, the size of the LUA heavily depends on the structure of the type hierarchy $\mathcal{T}$ and on the number of methods which are in $\mathbf{M}$ (that is, defined on $\mathcal{T}$). These factors make it difficult to properly evaluate the average space-complexity of the LUA.

However, given $\mathcal{T}$ and $\mathbf{M}$, we can express the size of the corresponding LUA as:

$$|Q| + \prod_{q \in Q-F} |\Sigma(q)|$$

which is the sum of the number of states and the number of all state transitions. As we have anticipated earlier it is difficult to relate this dimension with parameters depending on the hierarchy. A possibility, which is left for future investigation, is to experimentally evaluate how the dimension of the automaton varies depending on the number of types in the hierarchy (i.e., $|\mathcal{T}|$) and on the number of defined methods (i.e., $|\mathbf{M}|$).

6.5 Discussion and Comparison

In Section 6.3 we have proposed two different techniques for dispatching methods in object models with multiple class direct membership. The main differences between the two approaches are summarized in Table 6.1.

	Information exploited	Context-dependence	Behavior identity	Applicability	Static typing	Cost
Preferred class	static type of the distinguished parameter	YES	NO	Total	YES	$O(h \cdot (n + height(\mathcal{T})))$
Argument specificity	dynamic types of undistinguished parameters	NO	YES	Total	NO	$O(n \cdot h \cdot d)$

Legenda:
n = arity of invoked method;
h = number of most specific classes to which an object can belong;
d = number of definitions of the invoked method;
$height(\mathcal{T})$ = maximum among the lengths of paths with origin in the root of $\mathcal{T}$.

Table 6.1: Comparison between the two proposed dispatching approaches.

For what concerns the information exploited to dispatch the method invocation, both the dispatching techniques use the dynamic types of the method receiver (which are, obviously, known only at run-time) and the total order on classes $\preceq_T$, which can be fixed (and thus known statically), or object-dependent (and thus known at run-time). In addition, the preferred class dispatching rule use *static* information, that is, the static type of the object which receives the message, whereas the argument specificity dispatching rule use *dynamic* information, that is, the dynamic types of other actual arguments of the invocation.

As we have already pointed out, the preferred class dispatching approach models context-dependent behavior. Indeed, a given method invocation with a fixed set of parameters may produce different results (both in terms of results and data updates), though executed on the same state, depending on the context of the receiver object reference in the expression containing the invocation. By contrast, the argument specificity approach ensures the identity of behavior of a method invocation. This means that a given method invocation with a fixed set of parameters always produces the same results (both in terms of results and data updates) when executed on the same state.

The term *applicability* denotes whether or not the technique allows to dispatch any type correct method invocation. This result holds for both the techniques. Note, moreover, that the use of lookup automata for implementing the argument specificity dispatching rule, correctly works for type correct method invocation, but it can return an (incorrect) method when no applicable methods exist. Consider indeed the following example.

Example 6.6 *Referring to the type hierarchy $\mathcal{T}$ and to the set of methods* $\mathbf{M}$ *of the running example, consider the invocation* $X.\mathtt{m}(X_1, X_2, X_3)$ *where*

- *static_type*$(X) = F$, *static_type*$(X_1) = E$, *static_type*$(X_2) = D$ *and static_type*$(X_3) = D$;
- $msc(o) = \{F\}$, $msc(o_1) = \{E\}$, $msc(o_2) = \{D\}$ *and* $msc(o_3) = \{D\}$ *(being* o, o_1, o_2 *and* o_3 *the identifiers of objects to which* X, X_1, X_2 *and* X_3 *are instantiated, respectively).*

There is no applicable method for that invocation, indeed only $\mathtt{m}^F$ *meets the first applicability condition, but it does not meet the second condition since* $D \leq_T C$ *does not hold.*

If we simulate the LUA, however, we get as most specific applicable method the method associated with the final state q_{19}*, that is,* $\mathtt{m}^F$*. This result is however incorrect because method* $\mathtt{m}^F$ *is not applicable for the invocation.* ◇

If, instead of statically type checking the invocation as we do, we had decided to check the existence of an applicable method by means of the lookup automaton, we would have increased the complexity of dispatching. Indeed, we should have modified the automaton by adding to it all possible transitions so that all possible cases were considered (by labeling transitions with sets of types rather than with a single type). This would lead to an exponential growth in the LUA size.

As far as static typing is concerned, the preferred class dispatching rule always selects a method in a subtype of the static type of the distinguished parameter, thus the type correctness of method result is ensured. This does not hold for the argument specificity rule. In the remainder of this section, we discuss typing aspects, and compare our dispatching techniques with other existing dispatching approaches.

6.5.1 Typing Aspects

In this section we focus on the conditions to ensuring that the type of a method result is a subtype of the static type T of the invocation $X.\mathtt{m}(X_1, \ldots, X_n)$. Let c_{st} =

$s(\mathtt{m}, static\_type(X))$ be the class from which $static\_type(X)$ inherits method m, that is, let $\mathtt{m}^{c_{st}}$ be the method selected according to static dispatching. The covariant redefinition of method result in the redefinition of method signatures in selfish method based systems, ensure the type correctness of method result. That is, it ensures that the returned value has type T such that $T \leq_T T_r^{c_{st}}$, being $T_r^{c_{st}}$ the type of the result in the signature for m in class c_{st}.

The problem is thus how to ensure that, for any method which can be applied to a given invocation, the returned value has a type which is a subtype of the static type of the invocation. Thus, given an invocation $X.\mathtt{m}(X_1, \ldots, X_n)$, we must consider the possible implementations of method m that could be executed, and compare the return types of those implementations with the static type of the invocation.

For the preferred class dispatching rule the following result holds.

Proposition 6.2 *Let $X.\mathtt{m}(X_1, \ldots, X_n)$ be a method invocation. Let $c_{st} = s(\mathtt{m}, static\_type(X))$ be the class from which $static\_type(X)$ inherits method* m*. Let $\mathtt{m}^{c^*}$ be the method selected for the invocation, according to the preferred class dispatching rule. Let $sign(\mathtt{m}, c^*)$ be $T_1^{c^*} \times \cdots \times T_n^{c^*} \rightarrow T_r^{c^*}$, and let $sign(\mathtt{m}, c_{st})$ be $T_1^{c_{st}} \times \cdots \times T_n^{c_{st}} \rightarrow T_r^{c_{st}}$. Then, the method produces a type correct result, that is, $T_r^{c^*} \leq_T T_r^{c_{st}}$.* □

Proof Sketch: According to the preferred class dispatching rule, if a method $\mathtt{m}^{c^*}$ is selected for execution, certainly $c^* \leq_T c_{st}$ holds. Then, because $sign(\mathtt{m}, c^*)$ is a correct refinement of $sign(\mathtt{m}, c_{st})$, the output argument is covariantly specialized, that is, $T_r^{c^*} \leq_T T_r^{c_{st}}$. □

The same result does not hold for the argument specificity rule, as shown by the following example.

Example 6.7 *Referring to the running example, consider $sign(\mathtt{m}, F) = B \times C \times D \rightarrow H$ and $sign(\mathtt{m}, G) = E \times D \times C \rightarrow G$. Consider now the invocation $X.\mathtt{m}(X_1, X_2, X_3)$ where $static\_type(X) = F$, $static\_type(X_1) = E$, $static\_type(X_2) = C$ and $static\_type(X_3) = D$. The static type of the invocation is H. However if, at run-time, the invocation is the one of our running example (which is correct, since the required relationships between static and dynamic types are according to Rule 6.1), method $\mathtt{m}^G$ is selected (see Example 6.5) and the result value has type G, which is not a subtype of H, and is thus type incorrect.* ◇

The problem is that the argument specificity dispatching rule does not ensure that the selected method belongs to a class which is a subtype of $static\_type(X)$, thus the result type is not necessarily a subtype of $T_r^{c_{st}}$.

We now characterize the set of classes whose method m could be executed for the invocation. Surely, they all belong to the same hierarchy $\mathcal{H}_i$ of $static\_type(X)$ and c_{st}. Moreover, they cannot be more general than (that is, superclasses of) c_{st}, otherwise late binding would be violated. However, we cannot exclude all classes c such that $\mathtt{m}^{c_{st}} \prec_{\mathtt{m}} \mathtt{m}^c$, because $\prec_{\mathtt{m}}$ is not necessarily coherent with $<_T$ on the first argument (that is, in general, $c_i <_T c_j$ and $\mathtt{m}^{c_j} \prec_{\mathtt{m}} \mathtt{m}^{c_i}$ may hold), and thus those methods could be executed [39]. The methods that can be executed are thus all methods $\mathtt{m}^c$ that verify the following three conditions:

1. $c \in \mathcal{H}_i$ where $\mathcal{H}_i$ is the hierarchy to which $static\_type(X)$ belongs;
2. $\mathtt{m}^c \prec_{\mathtt{m}} \mathtt{m}^{c_{st}}$ or $\exists\ c^*$ such that $c^* <_T c_{st}$ and $\mathtt{m}^c \prec_{\mathtt{m}} \mathtt{m}^{c^*}$;
3. $c_{st} <_T c$ does not hold.

Thus, given an invocation $X.\mathtt{m}(X_1, \ldots, X_n)$ with static type T, a possible approach to statically check the type correctness of method result under the argument specificity rule is the following: at compile-time, it is checked whether for any class c satisfying the above conditions, the result type of method m in class c is a subtype of T, that is $T_r^c \leq_T T$.

An alternative approach is to impose some restrictions on method definitions, similarly to what is proposed in [10] for static type checking of multi-methods. This second approach is more efficient, but it is less flexible, because some constraints are imposed on method definitions. The following definition restricts legal method definitions.

Definition 6.10 *(Method Consistency). A method* m *is said consistent if* $\forall\ c_1, c_2$ *satisfying the following conditions:*

- $\mathtt{m} \in M(c_1), \mathtt{m} \in M(c_2)$,
- *a method invocation exists to which* $\mathtt{m}^{c_1}$ *and* $\mathtt{m}^{c_2}$ *are both applicable,*
- $\mathtt{m}^{c_1} \prec_{\mathtt{m}} \mathtt{m}^{c_2}$

$T_r^{c_1} \leq_T T_r^{c_2}$ *holds.* □

Example 6.8 *Method* m *of Example 6.7 is not consistent because* $\mathtt{m}^F$ *and* $\mathtt{m}^G$ *are both applicable to the invocation of Example 6.7,* $\mathtt{m}^G \prec_{\mathtt{m}} \mathtt{m}^F$, *but* $T_r^G = G \leq_T T_r^F = H$ *does not hold.* ◇

The notion of consistency, together with a covariant redefinition of method results, ensures type correctness of method results, as stated by the following proposition.

Proposition 6.3 *Let* $\mathtt{m}$ *be a consistent method (according to Definition 6.10). Let* $X.\mathtt{m}(X_1, \ldots, X_n)$ *be a method invocation. Let* $c_{st} = s(\mathtt{m}, static\_type(X))$ *be the class from which* $static\_type(X)$ *inherits method* $\mathtt{m}$. *Let* $\mathtt{m}^{c^*}$ *be the method selected for the invocation, according to the argument specificity dispatching rule. Let finally* $sign(\mathtt{m}, c^*)$ *be* $T_1^{c^*} \times \cdots \times T_n^{c^*} \rightarrow T_r^{c^*}$, *and* $sign(\mathtt{m}, c_{st})$ *be* $T_1^{c_{st}} \times \cdots \times T_n^{c_{st}} \rightarrow T_r^{c_{st}}$. *Then,* $X.\mathtt{m}(X_1, \ldots, X_n)$ *produces a type correct result, that is,* $T_r^{c^*} \leq_T T_r^{c_{st}}$. □

Proof Sketch: The following cases should be considered:

- $c_{st} \in msc(o)$; there are two further cases to be considered:
  - $c^* = c_{st}$, then, trivially, $T_r^{c^*} \leq_T T_r^{c_{st}}$;
  - $c^* \neq c_{st}$: since c_{st} is applicable, if c^* has been selected instead of c_{st}, surely $\mathtt{m}^{c^*} \prec_{\mathtt{m}} \mathtt{m}^{c_{st}}$ must hold; then, for the consistency of $\mathtt{m}$, $T_r^{c^*} \leq_T T_r^{c_{st}}$;
- $c_{st} \notin msc(o)$; let $\hat{c} \in msc(o)$ such that $\hat{c} <_T c_{st}$. Such a class $\hat{c}$ certainly exists because of the relationships among static and dynamic types stated by Rule 6.1. The rule ensures the existence of a class $\hat{c} \in msc(o)$ such that $\hat{c} \leq_T static\_type(X)$; but, by definition of c_{st} $static\_type(X) \leq_T c_{st}$, and $\hat{c} \leq_T c_{st}$ follows by transitivity of $\leq_T$ (and $\hat{c} <_T c_{st}$ because $\hat{c} \neq c_{st}$). There are two possible cases to consider:
  - $c^* = \hat{c}$: then, since $\hat{c} <_T c_{st}$, for the covariant redefinition of method result, $T_r^{\hat{c}} \leq_T T_r^{c_{st}}$ holds;
  - $c^* \neq \hat{c}$: since $\hat{c}$ is applicable, if c^* has been selected instead of $\hat{c}$, certainly $\mathtt{m}^{c^*} \prec_{\mathtt{m}} \mathtt{m}^{\hat{c}}$ and thus, for the consistency of $\mathtt{m}$, $T_r^{c^*} \leq_T T_r^{\hat{c}}$, and $T_r^{c^*} \leq_T T_r^{c_{st}}$ follows. □

Under the approach above, the consistency check for a method definition is performed at method definition time, rather than when the invocation is compiled.

6.5.2 Comparison

We start by comparing dispatching rules that can be applied to models supporting multiple class direct membership (described in Section 6.3) with existing approaches for method dispatching in systems that support predicate classes and in those that support roles. Method dispatching for predicate classes is quite similar to argument specificity dispatching rule. There are however several differences between our argument specificity dispatching approach and dispatching in Cecil:

1. In Cecil, type correct method dispatching is not total (indeed, certain method invocations are not dispatched as ambiguous), while our argument specificity dispatching rule is total.

2. In Cecil the first formal argument of a method does not have any privilege; under the argument specificity it has such a privilege. In our approach, indeed, only methods in the most specific types of the first argument can be selected.

3. Finally, in Cecil if an object inherits simultaneously from two objects o and o' (multiple inheritance) then either these objects do not have common methods, or they must be one a proper subtype of the other; this constraint is not imposed in our model.

In method dispatching for systems supporting roles the notion of context is present and therefore dispatching in those systems is more similar to preferred class dispatching. The main differences between our preferred class dispatching rule and the one adopted in Fibonacci are the following:

1. The preferred class dispatching rule is more general than the one in Fibonacci. In the preferred class approach the total order on classes can be arbitrary, while in Fibonacci it is always based on the acquisition order of roles. Thus, our dispatching approach can model that of Fibonacci as a particular case.

2. The preferred class dispatching rule does not provide strict binding, while Fibonacci supports such alternative dispatching mechanism.

Now, we point out similarities and differences between our technique for dispatching methods according to the argument specificity approach and existing techniques for multi-methods dispatching. In particular, we compare our technique and the one proposed by Chen, Turau and Klas [71]. As we have anticipated in Section 6.1, the approach proposed in [71] performs dispatching for multi-methods through the use of lookup automata (LUA). The main differences can be summarized as follows:

1. Lookup automata for multi-methods only use the total order $\preceq_T$ during construction (at compile-time), whereas during simulation (at run-time) they only use the partial order $\leq_T$, exactly the opposite of what is done in our approach[14]. Our dispatching technique, indeed, makes use of the partial order during construction and of the total

[14]Note that both the dispatching techniques (lookup automata and compressed dispatch tables) we have presented in Section 6.1 use the total order $\mathtt{E}$ only during the construction (at compile-time), whereas during simulation (at run-time) they only make use of the partial order $<_{\mathtt{m}}$.

one only during simulation. Our choice is crucial in our context, because using the total order only at run-time enables the use of an object-dependent total order, like the acquisition order of classes, that is known only at run-time.

2. Lookup automata for multi-methods have in general a size greater than ours. There are two reasons for that:

   (i) in LUAs for multi-methods additional transitions, because of multiple inheritance, must be considered in order to simulate all possible cases (otherwise the results of every simulation would not be correct and backtracking would be required);

   (ii) since our approach does not support "pure" multi-methods, in our model only a single definition of the same method exists in the same class (therefore $d \leq |\mathcal{T}|$); by contrast, in CLOS a method could have a different definition for each possible combination of argument types (therefore $d \leq |\mathcal{T}|^n$).

   Consequently, with respect spatial complexity our approach is better.

3. Lookup automata for multi-methods check also whether a function invocation is type correct, while in our approach this check is not performed because it is performed statically. Our technique can perform such a check at compile-time because in our model method redefinition must verify the contravariance rule for input parameters.

4. Method invocations for our approach are sets of types, while in lookup automata for multi-methods each argument of an invocation has only a single type. This is the reason why in our case the temporal complexity is in $O(n \cdot h \cdot d)$, while in the other case it is in $O(n \cdot d)$.

Conclusions

If we reason on the evolution of database technology, we observe a progressive attempt of subtracting semantics from applications and adding it to the generic, system-controlled component of the database. First, applications were decoupled from the physical data representation on files (physical independence); next, applications were decoupled from the actual schema structure used in the database (logical independence). The next step in this evolution can be considered as the achievement of knowledge independence, that is, the decoupling of applications from generic, application-independent, shared knowledge. Object-oriented data models, deductive data models, and active data models can be seen, though to different extents, going towards this direction. However, the areas of active, deductive and object-oriented database systems have primarily developed as separate research areas. Although significant results have been achieved in each area individually, there are weaknesses in each type of database system that can be overcome by combining the strengths of each. The successful integration of these areas has the potential of providing the kind of powerful database processing environment that will be required by database applications of the future.

In this thesis we have formally defined an active and deductive object-oriented data model. The proposed model exploits rule-based query and method definition languages. Issues related to a clean integration of rules with object-oriented modeling capabilities, such as methods and inheritance, have been clarified. In the thesis we have specified the semantics of the method and trigger definition languages, and we have investigated typing aspects. Issues that have been addressed include: the formal definition of a reference object-oriented data model, typing of object-oriented rule languages, behavior refinement for methods expressed by rules, integration of triggers with inheritance hierarchies, efficient method dispatching, views.

In particular, the formal definition of a rich object-oriented data model supporting composite objects, object migration, and multiple class direct membership has been given. A set-oriented rule-based method definition language has been defined for that model; its syntax and semantics have been formally specified and method semantic properties, such as behavioral refinement, have been investigated. An active rule language, supporting trigger inheritance and overriding, has been defined. The syntax and the semantics of the language has been formally specified, and some static conditions ensuring a correct refinement of triggers in subclasses have been stated. A view mechanism, supporting set-tuple, object-preserving and object-generating views has been defined for the model. Two view levels have been discussed for the model: views (that is, virtual classes) and schema views (that is, virtual schemas). How the database can be *seen* from these different schema views is also discussed. Finally, method dispatching in object data models supporting multiple class direct membership has been addressed. Two different dispatching techniques have been proposed and compared.

The model presented in the thesis has also been extended with temporal capabilities, and it has then been used as a basis for the formal definition of a temporal object-oriented data model [30, 29]. Temporal databases [166] support modeling of time-varying data, in that, while the content of a conventional database represents a snapshot of the reality at the current time, temporal databases maintain the complete history of data over time. Some architectural issues, mainly concerning efficient trigger support, have been also investigated with reference to the model presented in the thesis [57, 100].

The work reported in the thesis could be extended along different directions. A first extension could be the support of recursion in the method definition language and in the view-query specifications.

For what concerns the view mechanism an important possible topic of future work concerns update propagation. The view update problem has been widely investigated in the relational context. As noted in [156], the existence of object identifiers makes easier updating object-oriented views than relational views, because it is possible to establish a mapping among a view instance and its base object(s). In analyzing object-oriented view updates, it is necessary to distinguish between updating object-preserving views and object-generating views. To our knowledge, all the proposed approaches [14, 156] have only considered object-preserving views. View updates are quite straightforward when a view model only includes object-preserving views, because they are automatically propagated to the base objects: both the view instance and its base object have the same object identifier. Since our model supports multiple class instantiation, the approach to view update presented in [156] is applicable to the object-preserving views considered in our view model. In the case of object-generating views, the Derived By table can be employed for holding the correspondence between each generated view instance and the base objects from which it is derived. Through this table, the update operations (e.g. insert, delete, modify) can be propagated to the base objects. Moreover, object-oriented data models offer the possibility of specifying in methods how to propagate ambiguous updates, such as deletions on views defined as joins, which are forbidden in the relational context.

Issues related to schema validation and resolution of conflicts among different external schemas could be investigated too. These issues are particularly relevant in a data model allowing the specification of arbitrary constraints for views.

For what concerns multiple class direct membership, it seems reasonable, from a modeling perspective, that some classes, though belonging to the same inheritance hierarchy, should not have common members. An example can be given by the classes `male` and `female`. A possible extension of this work is to investigate how such kinds of *exclusiveness* constraints can be expressed and how these constraints, stating that two classes always have disjoint extents, can influence dispatching and type checking.

Our data model can also be extended to support schema evolution, investigating how the taxonomy of object-oriented schema changes proposed in [24, 177] can be extended to our model, and how the proposed view mechanism can be exploited to support all the possible changes.

Another possible direction along which the work reported in the thesis can be extended concerns implementation aspects. Possible interesting aspects to be investigated include: how to provide an ad-hoc efficient support for composite objects, how to efficiently support set-oriented method execution, how to adapt incremental view materialization techniques to our view model, how to efficiently determine the set of triggered rules in active object model supporting rule overriding, possibly combining method look-up and trigger look-up.

Bibliography

[1] S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Objects. *VLDB Journal*, 4(4):727–794, 1995.

[2] S. Abiteboul and A. Bonner. Objects and Views. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 238–247, 1991.

[3] S. Abiteboul and S. Grumbach. COL: A Logic Based Language for Complex Objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programing Languages*, pages 347–377. ACM Press, 1990.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[5] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 159–173, 1989.

[6] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and Rules. In P. Buneman and S. Jajodia, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 32–41, 1993.

[7] A.D.B. S.A, Paris. *Matisse 2.3 Tutorial*, 1995.

[8] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1989.

[9] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 36–45, 1989.

[10] R. Agrawal, L. G. De Michiel, and B. C. Lindsay. Static Type Checking of Multi-Methods. In A. Paepcke, editor, *Proc. Sixth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 113–128, 1991.

[11] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice Hall, 1972.

[12] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Object Data Model with Roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 39–51, 1993.

[13] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[14] S. Amer-Yahia, P. Breche, and C. Souza dos Santos. Object Views and Updates. In *Proc. of Journées Bases de Données Avancées - BDA'96*, 1996.

[15] E. Amiel, O. Gruber, and E. Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *Proc. Ninth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 244–258, 1994.

[16] M. Andries, L. Cabibbo, J. Paredaens, and J. van den Bussche. Applying an Update Method to a Set of Receivers. In *Proc. of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.

[17] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In P. Buneman and S. Jajodia, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 99–108, 1993.

[18] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-whole Relations in Object-Centered Systems: An Overview. *Data and Knowledge Engineering*, 20(3):347–383, 1996.

[19] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim et al., editors, *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[20] R. Bal and H. Balsters. A Deductive and Typed Object-Oriented Language. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, number 760 in Lecture Notes in Computer Science, pages 340–359, 1993.

[21] H. Balsters, R. A. de By, and R. Zicari. Typed Sets as a Basis for Object-Oriented Database Schemas. In *Proc. Seventh European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, 1993.

[22] H. Balsters and M. Fokkinga. Subtyping can have a Simple Semantics. *Theoretical Computer Science*, 87:81–96, September 1991.

[23] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan-Kaufmann, 1992.

[24] J. Banerjee, W. Kim, H.K. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1987.

[25] C. Beeri. Formal Models for Object Oriented Databases. In W. Kim et al., editors, *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395, 1989.

[26] C. Beeri and T. Milo. A Model for Active Object Oriented Database. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 337–349, 1991.

[27] V. Benzaken and A. Doucet. Thémis: a Database Programming Language with Integrity Constraints. In *Proc. Fourth Int'l Workshop on Database Programming Languages*, Springer WICS, pages 243–262, 1993.

[28] E. Bertino. A View Mechanism for Object-Oriented Databases. In M. L. Brodie and S. Ceri, editors, *Proc. Third Int'l Conf. on Extending Database Technology*, pages 136–151, 1992.

[29] E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera: A Temporal Object-Oriented Data Model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.

[30] E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, number 1057 in Lecture Notes in Computer Science, pages 342–356, Avignon (France), March 1996.

[31] E. Bertino, R. Ferrari, and G.Guerrini. Containment among Chimera Positive Conjunctive Queries. Technical Report in preparation, 1997.

[32] E. Bertino and G. Guerrini. Viewpoints in Object Database Systems. In A. Finkelstein, A. Wolf, and G. Spanoudakis, editors, *Joint Proc. of the SIGSOFT '96 Workshops*, pages 289–293, 1996.

[33] E. Bertino and G. Guerrini. Object-Oriented Databases. In J. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1997. To Appear.

[34] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 102–126, Aarhus (Denmark), August 1995.

[35] E. Bertino, G. Guerrini, and I. Merlo. A Set-Oriented Method Definition Language for Object Databases. Technical Report DISI-TR-97-11, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1997.

[36] E. Bertino, G. Guerrini, and I. Merlo. Trigger Inheritance and Overriding in an Active Object Database System. In *Proc. Fifth Int'l Conf. on Deductive and Object-Oriented Databases*, 1997. To Appear.

[37] E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 213–235, Bologna (Italy), July 1994.

[38] E. Bertino, G. Guerrini, and D. Montesi. Towards Deductive Object Databases. *Theory and Practice of Object Systems*, 1(1):19–39, Spring 1995. Special Issue: Selected Papers from ECOOP '94.

[39] E. Bertino, G. Guerrini, and L. Rusca. Method Dispatching in Object Data Models with Multiple Class Direct Membership. Technical Report DISI-TR-96-17, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1996.

[40] E. Bertino, G. Guerrini, and L. Rusca. Object Evolution in Object Databases. In B. Franhoefer and R. Pareschi, editors, *Theoretical Approaches to Dynamic Worlds: from the Frame Problem to the Theory and Practice of Knowledge Management*. Kluwer, 1997. To Appear.

[41] E. Bertino and S. Jajodia. Modeling Multilevel Entities using Single-Level Objects. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, number 760 in Lecture Notes in Computer Science, pages 415–428, 1993.

[42] E. Bertino and L.D. Martino. *Object-Oriented Database Systems - Concepts and Architecture*. Addison-Wesley, 1993.

[43] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-Oriented Query Languages: the Notion and the Issues. *IEEE Transactions on Knowledge and Data Engineering*, 4:223–237, 1992.

[44] G. Birkhoff. Lattice Theory. *American Mathematical Society Colloquium Publications*, 25, 1973.

[45] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. *SIGPLAN Notices*, 23, September 1988.

[46] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmerman. Rules in an Open System: the REACH Rule System. In N. Paton and M. Williams, editors, *Proc. First International Workshop on Rules in Database Systems*, Workshops in Computer Science, pages 111–126, 1993.

[47] P. Breche, F. Ferrandina, and M. Kuklok. Simulation of Schema Changes using Views. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, number 978 in Lecture Notes in Computer Science, pages 247–258, 1995.

[48] R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databasases, and Applications*, pages 283–308. Addison-Wesley, 1989.

[49] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[50] M. Buchheit, M. Jeusfeld, W. Nutt, and M. Staudt. Subsumption Between Queries to Object-Oriented Databases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 15–22, 1994. Extended version in *Information Systems*, 19(1).

[51] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. The Logres project: Integrating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm. Technical Report TR 89-039, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1989.

[52] S.J. Cannan and G.A.M. Otten. *SQL - The Standard Handbook*. McGraw-Hill, 1992.

[53] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[54] L. Cardelli. Types for Data Oriented Languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. First Int'l Conf. on Extending Database Technology*, Lecture Notes in Computer Science, pages 1–15, 1988.

[55] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polimorphism. *Computing Surveys*, 17:471–522, 1985.

[56] G. Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[57] S. Castangia, G. Guerrini, D. Montesi, and G. Rodriguez. Design and Implementation for the Active Rule Language of Chimera. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, pages 45–54, London (UK), September 1995.

[58] R. Cattel. *Object Data Management – Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[59] R. Cattel. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, 1996.

[60] S. Ceri, P. Fraternali, and S. Paraboschi. Constraint Management in Chimera. *IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management*, 17(2):4–8, June 1994.

[61] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active Rule Management in Chimera. In S. Ceri and J. Widom, editors, *Active Database Systems.* Morgan-Kaufmann, 1996.

[62] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[63] S. Ceri and R. Manthey. Chimera: A Model and Language for active DOOD Systems. In J. Eder and L. Kalinichenko, editors, *Extending Information System Technology, Proc. Second International East/West Database Workshop*, pages 9–21, Klagenfurt, 1994.

[64] S. Ceri and R. Manthey. Consolidated Specification of Chimera. Technical Report IDEA.DE.2P.006.01, ESPRIT Project 6333, November 1993.

[65] S. Ceri and J. Widom. *Active Database Systems - Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, 1996.

[66] D. Chamberlin. *Using the New DB2 - IBM's Object-Relational Database System.* Morgan-Kaufmann, 1996.

[67] C. Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proc. Sixth European Conference on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science, pages 33–56. Springer-Verlag, Berlin, 1992.

[68] C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proc. Seventh European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 268–296, 1993.

[69] E. Chan. Containment and Minimization of Positive Conjunctive Queries in OODBs. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 202–211, 1992.

[70] P. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[71] W. Chen, V. Turau, and W. Klas. Efficient Dynamic Look-Up Strategy for Multi-Methods. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 408–431, 1994.

[72] W. Chen and D. S. Warren. C-Logic of Complex Objects. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378, 1989.

[73] F. Civello. Roles for Composite Objects in Object-Oriented Analysis and Design. In *Proc. Eighth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 376–385, 1993.

[74] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), July 1970.

[75] C. Collet, T. Coupaye, and T. Svensen. Naos: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Proc. Twentieth Int'l Conf. on Very Large Data Bases*, pages 132–143, 1994.

[76] R. Connor and R. Morrison. Subtyping without Tears. In *Proc. Fifteenth Australian Computer Science Conference*, pages 209–225, 1992.

[77] C.J. Date. Referential Integrity. In *Proc. Seventh Int'l Conf. on Very Large Data Bases*, pages 2–12, 1981.

[78] K. Davis, G. Dong, and A. Heuer. Discussion Report: Object Migration and Classification. In U. Lipeck and B. Thalheim, editors, *Proc. Fourth International Workshop*

on Foundations of Models and Languages for Data and Objects - Modelling Database Dynamics, Workshops in Computer Science, pages 223–227, 1992.

[79] U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC project. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.

[80] O. Deux et al. The Story of o_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[81] O. Diaz, N. Paton, and P. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 317–326, 1991.

[82] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proc. Fourth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 211–214, 1989.

[83] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. Addison-Wesley, 1989.

[84] P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Rule Generation for Constraint Enforcement in Active Databases. In U. Lipeck and B. Thalheim, editors, *Proc. Fourth International Workshop on Foundations of Models and Languages for Data and Objects - Modelling Database Dynamics*, Workshops in Computer Science, pages 153–173, 1992.

[85] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems*, 20(4):414–439, 1995.

[86] R. Gabriel, J. White, and D. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–38, September 1991.

[87] J.H. Gallier. *Logic for Computer Science.* Harper and Row, 1986.

[88] S. Gatziu and K. Dittrich. SAMOS: an Active Object-Oriented Database System. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):23–26, December 1992.

[89] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 327–336, 1991.

[90] N. Gehani, H. Jagadish, and O. Shmueli. Event Specification in Active Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 81–90, 1992.

[91] N. Gehani and H. V. Jagadish. Active Database Facilities in Ode. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.

[92] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[93] G. Gottlob, G. Kappel, and M. Schrefl. Semantics of Object-Oriented Data Models - The Evolving Algebra Approach. In *Proc. Int'l Workshop on Information Systems*, Lecture Notes in Computer Science, pages 144–160, 1991.

[94] G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–284, July 1996.

[95] S. Greco, N. Leone, and P. Rullo. COMPLEX: An Object-Oriented Logic Programming System. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359, August 1990.

[96] P. Grefen, R. de By, and P. Apers. Integrity Control in Advanced Database Systems. *IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management*, 17(2):9–13, June 1994.

[97] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. *Journal of Intelligent Information Systems*, 11(1):5–40, 1998.

[98] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. Technical Report IDEA.DE.2P.011.01, ESPRIT Project 6333, May 1994.

[99] G. Guerrini, E. Bertino, B. Catania, and J. Garcia-Molina. A Formal Model of Views for Object-Oriented Database Systems. *Theory and Practice of Object Systems*, 3(3):157–183, 1997.

[100] G. Guerrini and D. Montesi. Design and Implementation of Chimera Active Rule Language. *Data and Knowledge Engineering*, 24:39–67, 1997.

[101] E. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 49–58, 1992.

[102] Illustra Information Technologies, Oakland, California. *Illustra User's Guide*. Release 2.1.

[103] H. Ishikawa, Y. Izumida, N. Kawato, and T. Hayashi. An Object-Oriented Database System and its View Mechanism for Schema Integration. In *Proc. of the Second Far-East Workshop on Future Databases*, pages 194–200, 1992.

[104] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. Beyond Coupling Modes: Implementing Active Concepts on Top of a Commercial ooDBMS. In E. Bertino and S. Urban, editors, *Proc. Int'l Symp. on Object-Oriented Methodologies and Systems*, number 858 in Lecture Notes in Computer Science, 1994.

[105] A. Karadimce and S. Urban. Diagnosis Anomalous Rule Behaviour in Databases with Integrity Maintenance. In *Proc. Third International Workshop on Foundations of Models and Languages for Data and Objects*, pages 77–102, 1991.

[106] A. Kemper and G. Moerkotte. A Formal Model for Object-Oriented Databases: The First Step. Technical report, University of Karlsruhe, 1991.

[107] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 393–402, 1992.

[108] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 134–146, 1990.

[109] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited). In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–393, 1989.

[110] W. Kim. *Introduction to Object-Oriented Databases*. The MIT Press, Cambridge, Mass., 1990.

[111] W. Kim. Object-Oriented Databases Systems: Promises, Reality and Future. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 676–687, 1993.

[112] W. Kim. UniSQL/X Unified Relational and Object-Oriented Database System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, page 481, 1994.

[113] W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk. Composite object support in a Object-Oriented Database System. In N. Meyrowitz, editor, *Proc. Second Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 118–125, 1987.

[114] W. Kim, E. Bertino, and J. Garza. Composite Objects Revisited. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 337–347, 1989.

[115] W. Kim and T. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. Fourteenth Int'l Conf. on Very Large Data Bases*, pages 148–159, 1988.

[116] W. Kim et al. Features of the ORION Object-Oriented Database System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databasases, and Applications*, pages 251–282. Addison-Wesley, 1989.

[117] W. Kim and W. Kelley. On View Support in Object-Oriented Database Systems. In W. Kim, editor, *Modern Database System*, pages 108–129. ACM Press, 1995.

[118] W. Kim and F. H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley, 1989.

[119] G. Kniesel. Implementation of Dynamic Delegation in Srongly Typed Inheritance-Based Systems. Technical Report IAI-TR-94-3, Institut für Informatik, Universität Bonn, 1994.

[120] R. Kowalsky. Database Updates in Event Calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[121] H. Kuno and E.A. Rundensteiner. The MultiView OODB View System: Design and Implementation. *Theory and Practice of Object Systems*, 2(3):203–225, 1996.

[122] C. Laasch and M. H. Scholl. Determinstic Semantics of Set-Oriented Update Sequences. In *Proc. Ninth IEEE Int'l Conf. on Data Engineering*, 1993.

[123] G. Leavens and W. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *Proc. Fifth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications joint with Fourth European Conference on Object-Oriented Programming*, pages 212–223, 1990.

[124] C. Lecluse, P. Richard, and F. Velez. 0_2, an Object-Oriented Data Model. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 257–276. Addison-Wesley, 1990.

[125] D. Lieuwen, N. Gehani, and R. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proc. Twelfth IEEE Int'l Conf. on Data Engineering*, 1996.

[126] L. Liu. *A Formal Approach to Structure, Algebra and Communication of Complex Objects.* PhD thesis, Katholieke Universiteit Brabant Tilburg, November 1992.

[127] Y. Lou and Z. M. Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Methods Inheritance. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 198–207, 1991.

[128] D. Maier. A Logic for Objects. In *Proc. of the Workshop on Foundation of Deductive Database and Logic Programming*, pages 6–26, 1986.

[129] V.M. Markowitz. Safe Referential Structures in Relational Databases. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 123–132, 1991.

[130] J. Martin and J. J. Odell. *Object-Oriented Analysis and Design*. Prentice Hall, 1992.

[131] D. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 215–223, 1989.

[132] C. Medeiros and P. Pfeffer. Object Integrity Using Rules. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 219–230, 1991.

[133] B. Meier. *Eiffel: The Language*. Prenctice Hall - Object-Oriented Series, Second Edition, 1992.

[134] R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, 1996.

[135] R. Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems*, 21(3):229–252, 1996.

[136] W. Mugridge, J. Hamer, and J. Hosking. Multi-Methods in a Statically Typed Programming Language. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 307–324, 1991.

[137] Object Design, Burlington (Mass.). *ObjectStore Reference Manual*, 1990. Object Design Inc.

[138] A. Ohori and K. Tajima. A Polimorphic Calculus for Views and Object Sharing. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266, 1994.

[139] M. P. Papazoglou. Roles: A Methodology for Representing Multifaced Objects. In *Proc. of the International Conference on Database and Expert Systems Applications*, pages 7–12, 1991.

[140] W. Paton, J. Campin, A.A.A. Fernandes, and M. Howard Williams. Formal Specification of Active Database Functionality: A Survey. In T . Sellis, editor, *Rules in Database Systems, Proc. of the 2nd International Workshop RIDS'95*, number 985 in Lecture Notes in Computer Science, pages 21–35, Greece, September 1995.

[141] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK System Revisited. Technical Report KIT - Report 75, Technische Universitat Berlin, 1989.

[142] D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. Second Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, 1987.

[143] B. Pernici. Objects with Roles. In *Proc. of the ACM Conference on Office Information Systems*, pages 205–215, 1990.

[144] R. J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Managent System.* PhD thesis, Department of Computing Science, University of Alberta, April 1994.

[145] A. Coen Porisini, L. Lavazza, and R. Zicari. Static Type Checking of Object-Oriented Databases. Technical Report 91-060, Dipartimento di Elettronica e Informazione, Politecnico di Milano, October 1991.

[146] Y.G. Ra, A. Kuno, and E.A. Rundensteiner. A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views. Technical report, Department of Electronic Engineering and Computer Science, University of Michigan, April 1994.

[147] Y.G. Ra and E.A. Rundensteiner. A Transparent Schema Evolution System Based on Object-Oriented View Technology. In *Proc. Eleventh IEEE Int'l Conf. on Data Engineering*, pages 165–172, 1995.

[148] J. Richardson and P. Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 298–307, 1991.

[149] G. A. Rose. Object-Z. In S. Stepheney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, pages 59–77. Springer-Verlag, Berlin, 1992.

[150] E.A. Rundensteiner. A Methodology for Supporting Multiples Views in Object-Oriented Databases. In *Proc. Eighteenth Int'l Conf. on Very Large Data Bases*, pages 187–198, 1992.

[151] E.A. Rundensteiner. The MultiView Approach to the Object-Oriented View Classification Problem. Technical report, Department of Electronic Engineering and Computer Science, University of Michigan, 1995.

[152] K. D. Schewe and B. Thalheim. Fundamental Concepts of Object Oriented Databases. *Acta Cybernetica*, 11(1-2):49–83, 1993.

[153] K.-D. Schewe, B. Thalheim, J. Schmidt, and I. Wetzel. Integrity Enforcement in Object-Oriented Databases. In U. Lipeck and B. Thalheim, editors, *Proc. Fourth International Workshop on Foundations of Models and Languages for Data and Objects - Modelling Database Dynamics*, Workshops in Computer Science, pages 174–195, 1992.

[154] J. Schilling and P. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In *Proc. Fourth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 353–361, 1989.

[155] M. Scholl, C. Laasch, and M. Tresch. Views in Object-Oriented Databases. In *Proc. Second International Workshop on Foundations of Models and Languages for Data and Objects*, pages 37–58, 1990.

[156] M. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In M. Kifer et al., editors, *Proc. Second Int'l Conf. on Deductive and Object-Oriented Databases*, number 566 in Lecture Notes in Computer Science, pages 189–207, 1991.

[157] M. Scholl and H. Schek. A Relational Object Model. In S. Abiteboul and P.C. Kanellakis, editors, *Proc. Third Int'l Conf. on Database Theory*, number 470 in Lecture Notes in Computer Science, pages 89–105, 1990.

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

[159] P. Shaw. Database Language Standards: Past, Present, Future. In A. Blaser, editor, *Database Systems of the 90s*, number 466 in Lecture Notes in Computer Science, pages 50–88. Springer-Verlag, Berlin, 1990.

[160] Y. Shyy and S. Su. Refinement Preservation for Rule Selection in Active Object-Oriented Database Systems. In J. Widom and S. Chakravarthy, editors, *Proc. Fourth International IEEE Workshop on Research Issues in Data Engineering - Active Database Systems*, pages 115–123, 1994.

[161] C. Souza dos Santos. Design and Implementation of Object-Oriented Views. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, number 978 in Lecture Notes in Computer Science, pages 91–102, 1995.

[162] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 81–94, 1994.

[163] M. Staudt, M. Jarke, M. Jeusfeld, and H. Nissen. Query Classes. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, number 760 in Lecture Notes in Computer Science, pages 283–295, 1993.

[164] L. A. Stein. A Unified Methodology for Object-Oriented Programming. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 211–222. John Wiley & Sons, 1991.

[165] D. Straube and M. Ozsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[166] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Database Systems and Applications Series. Benjamin/Cummings, 1993.

[167] S. Urban and L. Delcambre. Constraint Analysis: a Design Process for Specifying Operations on Objects. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December, 1990.

[168] G. Vossen and K. U. Witt. FASTFOOD: A Formal Algebra over Sets and Tuples for the FOOD Object-Oriented Data Model. Technical report, Arbeitsgruppe Informatik, Justus Liebig Universitat, Giessen, August 1991. TR n. 9103.

[169] G. Vossen and K. U. Witt. Objectbase Schemata and Objectbases in the FOOD Model. Technical report, Arbeitsgruppe Informatik, Justus Liebig Universitat, Giessen, June 1991. TR n. 9101.

[170] J. Widom. A Denotational Semantics for the Starbust Production Rule Language. *SIGMOD Record*, 21(3):4–9, September 1992.

[171] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 275–285, 1991.

[172] J. Widom and S. J. Finkelstein. Set-Oriented Production Rule in Relational Database Systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 259–270, 1990.

[173] R. Wieringa, W. de Jonge, and P. Spruit. Roles and Dynamic Subclasses: a Modal Logic Approach. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, 1994.

[174] M. E. Winston, R. Chaffin, and D. Herrmann. A Taxonomy of Part-whole Relations. *Cognitive Science: a Multidisciplinary Journal of Artificial Intelligence, Linguistics, Neuroscience, Philosophy, Psychology*, 11(4):417–442, 1987.

[175] C. Zaniolo. A Unified Semantics for Active and Deductive Databases. In N. W. Paton and M. H. Williams, editors, *Rules in Database Systems*. Springer-Verlag, Berlin, 1994.

[176] Y. Zhou and M. Hsu. A Theory of Rule Triggering Systems. In *Proc. Second Int'l Conf. on Extending Database Technology*, pages 407–421, 1990.

[177] R. Zicari. A Framework for Schema Udates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, pages 146–182. Morgan-Kaufmann, 1992.

Appendix A

Notations used in the Thesis

Symbol	Meaning
$D_1, \ldots, D_n$	basic domains
$\mathcal{OI}$, $\mathcal{CI}$	a set of object/class identifiers
$\mathcal{TN}$, $\mathcal{AN}$, $\mathcal{MN}$, $\mathcal{PN}$	a set of type/attribute/method/constraint names
$\mathcal{T}$, $\mathcal{OT}$, $\mathcal{VT}$	the set of types, object types, value types
$\mathcal{BVT}$, $\mathcal{CT}$, $\mathcal{ET}$	the set of basic value types, constrained types, extended types
$\mathcal{V}$	the set of legal values
2^S, $\|S\|$	the powerset of set S, the cardinality of the setv S
$\Pi_i\langle el_1, \ldots, el_n\rangle$	the i-th component el_i of the tuple $\langle el_1, \ldots, el_n\rangle$
$type(T)$	the type obtained from T replacing each named type appearing in it with the type it stands for
$struct(T)$	the structural component of the constrained or extended type T
$constr(T)$	the constraint component of the constrained type T
$ext(T)$	the extent of the extended type T
$ISA(c)$, $ISA^*(c)$	the set of the direct/of all the superclasses of class c
$ISA_V(T)$, $ISA_V^*(T)$	the set of the direct/of all the supertypes of the value type T
$T_2 \leq_T T_1$	T_2 is a subtype of T_1
$T_2 <_T T_1$	T_2 is a proper subtype of T_1
$T_1 \sqcup T_2$	the most specific common supertype (lub) of T_1 and T_2
$\pi(c)$ $(\pi_O(c))$	the (proper) extent of class c
$\pi(T)$ $(\pi_V(T))$	the (proper) extent of the extended type T
$\pi^*(c)$, $\pi^*(T)$	the global extent of class c, of extended type T
$dom(D_i)$	the (postulated) non-empty extension of the basic value type D_i
$v : T$	the value v belongs to type T
$\llbracket T \rrbracket_\pi$	the extension of type T under the assignment π
$stype(c)$	the type of the state of objects of class c (the *structural type* of class c)

Table A.1: Notations and terminology

Symbol	Meaning
$A(c)$, $M(c)$	the set of attributes/methods of class c
$dom(a,c)$	the domain of attribute a in class c
$sign(m,c)$	the signature of method m in class c
$\nu(i)$	the state of the object identified by the oid i
$source(a,c)$	the class from which class c inherits attribute a
$ref(o)$	the set of oids appearing in the state of object o
$comp(o)$	the set of oids appearing in the state of object o as the value of a composite attribute
$o_1 = o_2$	objects o_1 and o_2 are equal by identity
$o_1 == o_2$, $o_1 ==_d o_2$	objects o_1 and o_2 are equal by shallow/deep value equality
$s \leq_S s'$	signature s is a refinement of signature s'
$MC_2 \leq_{MC} MC_1$	metaclass MC_2 is a refinement of metaclass MC_1
$C_2 \leq^i_C C_1$, $C_2 \leq^e_C C_1$	class C_2 intensionally/extensionally refines class C_1
Var, Var_T	the set of variables and variables of type T
$Term$	the set of terms of our language
$Bind$	the set of possible sets of bindings
$State$	the set of possible database states
$Cond$	the set of well-formed declarative expressions of our language
$Update$	the set of well-formed update sequences of our language
$Event$	the set of possible event instances
$Rule$	the set of possible rules of the language
$\mathcal{E}$, $\mathcal{C}$, $\mathcal{U}$	semantic functions for terms, conditions, updates
$\mathcal{R}$, $\mathcal{P}$	semantic functions for active rules, reactive process
$\mathcal{VI}$, $\mathcal{CVI}$	the set of view identifiers, of virtual and base class identifiers
$\mathcal{BOI}$, $\mathcal{VOI}$	the set of base object/view object identifiers
$\mathbf{M}$	the set of method definitions
X_B	the set of values to which substitutions in the set of bindings B bind variable X

Table A.2: Notations and terminology (continued)

Symbol	Meaning
$dom(\theta)$	the domain of substitution θ
$\theta_{\|V}$	the restriction of substituition θ to the variables in the set V
$o \vdash^d o'$, $o \vdash^r o'$	object o' is deleted for dependence/referential integrity from object o
$\mu(m)$, $\mu(r)$	the set of classes manipulated by the execution of method m, trigger r
$\delta_m(c)$, $\delta_r(c)$	the set of objects deleted from class c by the execution of method m, trigger r
$\iota_m(c)$, $\iota_r(c)$	the set of objects inserted into class c by the execution of method m, trigger r
$\nu_m(i).a$, $\nu_r(i).a$	the value assigned to attribute a of object i by the execution of method m, trigger r
$BVar(R)$, $BVar(r)$	the set of variables shared by the condition and action part of the update rule R, of the trigger r
$B \text{-}^{\rho} B'$	the set of bindings B is included in the set of bindings B' to which renaming ρ is applied ($B \subseteq B'\rho$)
$t\ \mathtt{w}^{\rho}\ t'$	the term t is equal to term t' to which renaming ρ is applied ($t = t'\rho$)
$F[c_1/c_2]$	the formula obtained by formula F replacing each class formula $c_1(X)$ with class formula $c_2(X)$
$t_{\|c}$	the restriction of record term t to fields whose labels are attributes of class c
$u \leq_b^{\rho} u'$	update u refines update u' under renaming ρ
$Net(A)$	net effect of the action list A
$A \leq_a^{\rho} A'$	action list A refines action list A' under renaming ρ
$r <_r r'$	trigger r has priority over trigger r'
$c_1 \leq_{ISA} c_2$	class c_1 is a subclass of class c_2
F^{-E}	the formula obtained by F by eliminating event formulas
$C \leq_c^{\rho} C'$	condition C is a refinement of condition C'
$stype\_imp(v)$, $stype\_add(v)$	the imported/additional structural type of view v
$defined\_on(v)$	the set of root classes of view v
$oid(v)$	whether or not the instances of view v are objects
$new\_oid(v)$	whether the view v is object-generating
$derived\_by(i)$	the set of identifiers of objects from which view object i has been derived
$value(i, c)$	the state of the object identified by i seen as an instance of class c
$VISA(v)$, $VISA^*(v)$	the set of the direct/of all the superviews of view v
$v_1 \leq_{VISA} v_2$	view v_1 is a subview of view v_2
$static\_type(e)$, $static\_type(X)$	the static typer of expression e/variable X
$msc(o)$	the set of most specific classes to which object o belongs
$T_1 \preceq_T T_2$	type T_1 precedes type T_2 in the total order on types
$m_1 \prec_m m_2$	method m_1 is more specific than method m_2

Table A.3: Notations and terminology (continued)

Appendix B

Formats for Importing Features in View Specification

In the view definition statement, appropriate clauses denote the lists of features imported from root classes. For each feature, the associated list contains one or more items whose format can be one of the following:

a) `pName [of className]`, indicating that the feature named `pName` is imported from the root class named `className`. The specification of the class name is optional, except if the view is derived from several root classes having a feature of the same kind named `pname`.

b) `- pname [of className]`, indicating that the feature named `pname` is hidden[1]. As in the previous case, it is mandatory to specify the class name if name conflicts arise.

c) `all [of ListofClassNames]`, indicating that the view imports all the features of the class (or classes) specified. If no class is specified, the view imports all the features of all its root classes.

d) `name1 [of className] as name2`, indicating that the feature named `name1` is renamed as `name2` in the view. As in the first case, it is mandatory to specify the class name if name conflicts arise.

e) `attName: typeName derived`, indicating that a view attribute named `attName` is derived, with domain type `typeName`. Its implementation must be given in the view implementation.

[1]When the format `- pname [of className]` is used, all the other features (of the specified kind) of the class are intended to be included, thus a keyword `all [of className]` is implicitly added to the clause.

f) $\texttt{attName} : \texttt{className.a}_1.\cdots.\texttt{a}_n$, specifying that the view attribute `attName` corresponds to the nested attribute $\texttt{a}_1.\cdots.\texttt{a}_n$ of the class `className`. `className` is one of the root classes of the considered view. The expression $\texttt{className.a}_1.\cdots.\texttt{a}_n$ is very similar to commonly used path expressions[2]. Thus, for $1 \leq i \leq n-1$, $\texttt{a}_i$ must be an object valued attribute, while $\texttt{a}_n$ can be either an object valued or a value attribute.

Formats e) and f) are allowed only for attributes.

Table B.1 shows the correspondence between those formats and the corresponding items in the view signature.

[2] A difference with usual path expression is that paths start with a base class identifier instead of with a variable name.

1. for each item $\mathtt{a_i}$ **of class** $\mathtt{c_i}$ in the list of imported attributes, an item (a_i, T_i, at_i) belongs to $V.sign.struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being $C_i.id = c_i$ (that is, if C_i is the class identified by c_i);

2. for each item $\mathtt{a_i}$ in the list of imported attributes, an item (a_i, T_i, at_i) belongs to $V.sign.struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being C_i the only(1) class among the root classes of V, containing in its *struct* component an item whose first component is a_i;

3. for each item $\mathtt{a_i}$ **of class** $\mathtt{c_i}$ **as** $\mathtt{a_j}$ in the list of imported attributes, an item (a_j, T_i, at_i) is added to $V.sign.struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being $C_i.id = c_i$;

4. for each item $\mathtt{a_i}$ **as** $\mathtt{a_j}$ in the list of imported attributes, the item (a_j, T_i, at_i) is added to $V.sign.struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being C_i as in (2);

5. if the clause for specifying imported attributes contains the keyword `all`, then $V.sign.struct = \bigcup_{i=1,\ldots,n} C_i.sign.struct$, being $\{C_1, \ldots, C_n\}$ the root classes of V;

6. if the clause for specifying imported attributes contains the keyword **all of** $\mathtt{c_1}, \ldots, \mathtt{c_n}$, then $V.sign.struct = \bigcup_{i=1,\ldots,n} C_i.sign.struct$, being $C_i.id = c_i$, for each $i, 1 \leq i \leq n$;

7. for each item $-\mathtt{a_i}$ **of class** $\mathtt{c_i}$ in the clause for specifying imported attributes the item (a_i, T_i, at_i) is removed from $V.sign.struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being $C_i.id = c_i$(2);

8. for each item $-\mathtt{a_i}$ in the clause for specifying imported attributes the item (a_i, T_i, at_i) belongs to $V.sign, struct$ if the item $(a_i, T_i, at_i) \in C_i.sign.struct$, being C_i as in (2);

9. for each item $\mathtt{a_i} = \mathtt{c_i.a_1.\cdots.a_n}$ in the list of imported attributes the item $(a_i, T_i, derived)$ belongs to to $V.sign.struct$ if T_i is the type of $c_i.a_1.\cdots.a_n$(3);

10. for each item $\mathtt{a_i}$ $: \mathtt{T_i}$ **derived** in the list of imported attributes the item $(a_i, T_i, derived)$ belongs to $V.sign.struct$(4).

---

(1) If more than one root class of V contains an attribute whose name is a_i, then the specification of the class from which the attribute must be taken is mandatory.

(2) We assume that the keyword `all` implicitly added to the clause has been already considered by applying items 5 and 6.

(3) The type of $c_i.a_1 \cdots .a_n$ is obtained by making use of the following rule

$$\frac{e : c \quad c \in \mathcal{OT} \quad (a, T, at) \in C.sign.struct \quad c = C.id}{e.a : T}$$

starting from the base

$$\frac{}{c : c} \quad c \in \mathcal{OT}.$$

(4) In the case of derived attributes (for example, items (9) and (10)), the deductive rules specifying the attribute implementation must be present in the view implementation.

Table B.1: Correspondence between different formats for importing attributes and components of view signature