Dipartimento di Informatica e
Scienze dell'Informazione

Extending the ODMG Object Model with Temporal and Active Capabilities

by

Isabella Merlo

Theses Series **DISI-TH-2001-02**

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

Extending the ODMG Object Model with Temporal and Active Capabilities

by

Isabella Merlo

April, 2001

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università 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 Isabella Merlo
Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova
`merloisa@disi.unige.it`

Date of submission: December 2000

Title: Extending the ODMG Object Model with Temporal and Active Capabilities

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

Dott. Giovanna Guerrini
Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova
`guerrini@disi.unige.it`

Supervisor: Prof. Eugenio Moggi
Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova
`moggi@disi.unige.it`

Ext. Reviewers: Prof. Shashi Gadia
Department of Computer Science,
Iowa State University
`gadia@cs.iastate.edu`

Prof. Barbara Pernici
Dipartimento di Elettronica e Informazione,
Politecnico di Milano
`barbara.pernici@polimi.it`

Abstract

Object database systems (both the pure object-oriented systems and the object-relational one) are the systems that in the next few years will replace conventional relational databases systems, or even older generations systems (such the hierarchical and the network ones). On the other hand, temporal and active database systems, both relational and object-oriented, provide several innovative features that are critical for many advanced data management applications. Temporal database systems efficiently manage not only the current values of data, but also the entire history of data over time. Such systems are particularly useful in many application domains, such as the business and financial ones. An active database system is a system in which some operations are automatically executed when specified events happen and particular conditions are met. Examples of use of active capabilities are integrity constraint enforcement, authorization and monitoring. Although many approaches have been proposed in the past to extend object database systems with such features and interesting results have been achieved, there is a lack of uniformity and standardization across those approaches. One of the reason is that, the standard for object-oriented databases, ODMG, is recent and not well-established. However, ODMG provides the basis for introducing new capabilities to object-oriented databases that could be regarded as the standard for future object database management systems. Among them, the introduction of temporal and active capabilities to ODMG is an important issue that research in the database research area has to address. The contribution of the thesis is the introduction of these two features into the ODMG standard. First, the ODMG object model is formalized, then the introduction of the time dimension and of active rules is addressed. The thesis is organized as follows. The first chapter surveys related works. In Chapter 2 the ODMG object model is formalized. Such formalization is the basis on which temporal and active features are introduced in ODMG. In Chapter 3 the object model is extended to properly handle temporal data expressed with respect to different temporal granularities. In Chapter 4 the impact of temporal data and temporal granularities on the query language is investigated. In particular, comparisons of values related to different time granularities and object navigation are addressed. In Chapter 5 a language for the introduction of triggers in the ODMG object model is proposed. As part of such language, we propose a data manipulation language for ODMG composed by a set of predefined manipulation primitives.

To Giovanna, for still believing in her ideals.

There is nothing in life to be scared of.
There are just things to be understood. (M. Curie)

Acknowledgements

To my advisors, Elisa Bertino and Giovanna Guerrini.

To Elisa, for providing me the subject of my research and for always giving me great opportunities during these years.

I read once that research is mainly playing with ideas. Then, I would like to thank Giovanna, to whom the thesis is dedicated, for playing with me the game of research, I had fun! Even though sometimes it was hard work! Her help has been fundamental, in particular in the last months.

To Elena Ferrari, who collaborated to the research work of this thesis and who taught me to give the right importance to the right things.

To Shashi Gadia, who gave me very important suggestions and actively participated to the research concerning temporal object-oriented databases of our group.

To all those provided me helpful comments on the thesis, among them: Suad Alagić, Claudio Bettini, Eugenio Moggi, and Elena Zucca.

To the students who have developed a thesis on T_*ODMG*, who shared with me their ideas and who taught me many things...among them let me mention: Fabio, Gabriele, Daniela, Elena, Stefano, Alessio, and Milena.

To all my collegues in Genoa and Milano. In particular to people "living" (because they spend most part of their lives there!) in 309: Marco, Viviana, Walter, and Francesca. Their patience, especially in the last period, has been heavily tested by me...they listened to me even when I would not have listened to me by myself!

To Davide who is one of my oldest friends...(not referring to his age!) and whose jokes and humor have always made me laugh...even though I was frequently the target of the jokes...

To Eva: commuting is never a pleasure...but thanks to her it has been less hard.

To all my friends who decided to live in Milano and who created a hotel chain which is one of the best you can find in the city...among them let me mention Paola, Carlo, Alessia G., and Alessia V.. Kindness, good food, and a smile are never missed in those places!

To my and Roberto's wonderful families, who have always believed in me and in my ideas. Their help and support have been fundamental during these years...I cannot wait to be close to them again...especially to Stefania, my sister, who will give me soon the great joy of seeing the family growing.

Finally, I would like to thank Roberto, my husband. Sharing my life with him makes it special. He is the most precious thing I have and I hope we will take care of each other for a long long time...

Table of Contents

List of Figures

List of Tables

Introduction

Since its appearance, about thirty years ago, the object-oriented paradigm has been regarded as one of the most promising approaches to the development of information systems. The main concepts of the object-oriented paradigm include the notions of object, class and inheritance. Objects, belonging to classes, are characterized by an identity and a state (attribute values) and a behavior (operations). Classes are templates for classifying objects, they are characterized by a set of attributes and a set of operations. Inheritance allows one to define a class in terms of an existing class inheriting all its attributes and operations. Moreover, object-orientation introduced several interesting novel features, such as encapsulation.

Most recently, the object-oriented technology has been successful applied to several applications. Let us mention distributed systems, telecommunications, geographical information systems, and software engineering. Finally, Java [AG98], one of the most popular new generation programming languages, is an object-oriented language.

Object-oriented database management systems (OODBMSs) [BM92, CZ00] result from the integration of database technology with the object-oriented paradigm developed in the programming language and software engineering areas. From the point of view of the management of data, the object-oriented approach provides the required flexibility because it is not limited to the data types and query languages available in traditional database systems. One of the most important features of OODBMSs is the ability to specify both the structure of complex application objects and the operations to manipulate those structures. Commercial OODBMS products appeared more than a decade ago. Since then OODBMSs were being significantly improved from their earliest releases. We believe that OODBMSs will replace traditional database systems in a few years. Object-relational database management systems (ORDBMSs), which can be regarded as the last generation of DBMSs, are a concrete proof of that. The last release of the SQL standard, SQL:1999 [GP99, Mel99], is based on an object-relational data model. In SQL:1999 relations are still the fundamental data structuring concepts, but objects and classes are introduced in the relational model, extended by supporting reference types and complex types for the tuples of relations and for the domains of relation attributes.

To date, the use of OODBMSs has been hindered by the lack of a standard. Indeed, the success of relational database systems did not simply result from a higher level of data independence and a simpler data model than previous systems. Much of their success has been due to the standardization they offer. The acceptance of the SQL standard resulted in a high degree of portability and interoperability among systems, simplified learning new relational DBMSs, and represented a wide endorsement of the relational approach. The lack of any common standard led the major OODBMS companies to establish in 1991 a consortium, ODMG (Object Database Management Group), with the goal of developing suitable standards for OODBMSs. The goal of ODMG was to develop a standard specification for the object model that could allow one to write *portable* schemas and applications for object-oriented databases. The intense ODMG effort has given the object database industry a "jump start" towards standards that would otherwise have taken many years. ODMG enables many vendors to support and endorse a common object database interface with respect to which customers can write their applications.

ODMG 3.0 [CBB$^+$99] is the recent release of the object database standard. It follows ODMG-93 [Cat93] and its subsequent releases: ODMG 1.1 [Cat94], ODMG 1.2 [Cat96], and ODMG 2.0 [CBB$^+$97]. The ODMG standard includes a reference object model (ODMG object model), an object definition language (ODL) and an object query language (OQL). Moreover, the C++, Smalltalk, and Java programming language bindings for ODL and for object manipulation have been specified [CBB$^+$99]. The task undertaken by ODMG was obviously difficult. The systems manufactured by the member companies were (and are) very different from each other in many ways. We want to underline the fact that ODMG offers a starting point to further standardize OODBMSs and it is not the arrival point. Indeed, the research in the object-oriented database area is still active in investigating open problems and extensions to the ODMG standard. Among the possible extensions, we believe that temporal and active capabilities should be added to the ODMG standard.

Temporal database systems manage not only the current value of data, but also the entire history of data over time. Indeed, conventional database systems do not offer appropriate functionality to deal with time-varying data. The content of a database represents a *snapshot* of the reality in that only the current data are recorded, without the possibility of maintaining the complete history of data over time. If such a need arises, data histories must be managed at application program level, thus, making the management of data very difficult, if at all possible. Temporal database systems overcomes this lack. Several temporal extensions to the relational and the object-oriented data models and the associated query languages have been proposed [WJW98]. An important requirement, when dealing with temporal aspects, is the support for *multiple temporal granularities* [BJW00]. Temporal granularities are the units of measure for temporal data. For instance, birth dates are typically considered to have granularity in days and train schedules to that in minutes. Multiple temporal granularities allow one to store temporal data according to different temporal units, depending on the needs of the application domain. The choice of the correct temporal granularity not only allows the system to store the minimal amount of data, but also establishes an integrity constraint: data cannot be changed more often than the specified granularity. Since many different granularities exist and no granularity is inherently "better" than another, a temporal database system should support a wide range of temporal granularities and should allow the user to define his/her own application-specific granularities. The introduction of temporal granularities in object-oriented data models raises many relevant issues. For instance, one has to consider the impact of inheritance when multiple granularities are supported, and address issues concerning consistency, type refinement and substitutability.

An *active database system* is a system in which some operations are automatically executed when specified events happen and particular conditions are met. Active features are critical for many advanced data management applications. Examples of use of active capabilities are integrity constraint enforcement, authorization and monitoring. 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 to do when the rule is triggered and its condition is true. Most recent relational and object-relational DBMSs and the recent standard SQL:1999, provide capabilities to define triggers. Moreover, several proposals for adding triggers to object-oriented database systems have been presented [CW96, Pat99]. In both the relational and the object frameworks active rules provide a comprehensive means to formally state the semantics of data, the high-level semantic operations on data and the integrity constraints. Even though current relational database systems provide stored procedures, and object-relational and object-oriented database systems provide methods, as a means to express behavior of data, specifying the semantics of data through rules has important advantages over coding it into methods (or stored procedures). For instance, the behavior represented by methods must be explicitly invoked by the user or by applications, whereas active rules are autonomously activated. Moreover, 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.

Although the ODMG data model provides a very rich set of modelling primitives; it does not support the management of time-varying data and does not provide any active capability. We believe that such features, whose importance has been widely recognized by the database community, should be integrated in the ODMG standard. Aim of this thesis is the extension of the ODMG object model to support those features. Thus, the main contributions of this thesis can be summarized as follows. (*i*) The formal definition of the ODMG object model. (*ii*) The extension of the ODMG object model to support time varying data with multiple granularities and the definition of some primitives for querying them. (*ii*) The extension of the ODMG object model with triggers. In what follows we illustrate each contribution separately.

Although ODMG has been an important step in the area of OODBMSs, it has a major drawback: the lack of a formal basis. To date, there is no comprehensive formalization of the ODMG object model. Thus, the first aim of the thesis has been to address such a need by developing a formalization of the ODMG standard object data model. The ODMG object model provides a very rich set of modeling primitives. The formalization of a semantically rich data model, like the one proposed by ODMG, entails several interesting issues that we address in this thesis. For instance, we investigate relevant issues related to inheritance in the ODMG object model, that, to the best of our knowledge, have not been addressed so far. Such a definition is crucial for providing a formal basis on which the temporal and active extensions to the model can be developed. In addition, a formal definition, like ours, helps to better understand the model. The formal model can also be useful to everyone involved in teaching the ODMG model and it can be used as a basis for assessing whether theoretical and practical results concerning type systems can be applied to the ODMG model. Applications of those results may in turn lead to significant extensions to the ODMG model, such as the temporal and active ones we propose. Another relevant use of our formal model is to clarify to what extent the bindings towards various object-oriented programming languages, such as C++ and Java, have influenced the ODMG data model. We believe that a better understanding of issues concerning object-oriented programming language bindings has value not only for object-oriented database management systems, but also for object-relational database management systems that need to support such bindings as well. Finally, such a formal model can also be used as a basis for developing a foundation for object-relational data models and for XML [Wor98]. Such a foundation is relevant for example in developing systems and language bindings able to support applications for managing XML data and documents.

The second contribution of the thesis is the extension of the ODMG object model to support time varying data related to different granularities and how to query them. We propose in the thesis a comprehensive temporal object model supporting multiple temporal granularities. In particular, we first formally define the notion of temporal type with respect to a given granularity. Then we provide a formal definition for the set of legal values of our temporal type system. Then, the notions of class and object are formally defined. Subsequently, the notions of inheritance, type refinement and substitutability are revisited taking multiple granularities into account. Substitutability ensures that each instance of a given class can be used whenever an instance of one of its superclasses is acceptable. To enforce substitutability, the idea is that attribute domains can be replaced in a subclass by refined types. For instance, an attribute with granularity month can be refined in an attribute with granularity day. This means that the history of the daily changes of the attribute is kept in the subclass. Clearly, however, the attribute may not maintain the same value for each day of the month. Then, the question arises: how an instance of the subclass can be seen as an instance of the superclass? To overcome this problem, we introduce the notion of *coercion function*. Coercion functions are used to compute, given the values of the attribute in the subclass, the value to be considered in the superclass. Finally, all the introduced notions are "put together" leading to the definition of database schema and database. This requires us to develop the notion of *consistency* of a set of objects, which in turn encompasses several notions, such as temporal referential integrity and temporal keys.

In extending the ODMG object model to manage temporal data, we also investigate how temporal multi-granularity objects can be queried. Note that, though several temporal object-oriented query languages have been proposed in the literature, most of them do not consider multiple granularity data. Moreover, most of them are extensions of relational query languages rather than of existing nontemporal object-oriented query languages. Object-oriented database systems provide, in addition to traditional query languages, a *navigational* approach to data access. The navigational approach is based on object identifiers and aggregation relationships: given an oid, the system directly accesses the corresponding object and navigates through objects referred to by its components. This access modality can be combined with the query-based (e.g., SQL-like) access. Thus, conditions in a query can be imposed on the nested attributes of the queried objects; such nested attributes are in most cases denoted by *path expressions*, or similar syntactic notations. In this thesis, we investigate navigational accesses in a temporal multi-granularity context. This can be seen as the *core* of a query language for the proposed data model.

The last contribution of the thesis is the extension of the ODMG object model with triggers. Although several approaches have been proposed in the past to extend object database systems with triggers and interesting results have been obtained, there is a lack of uniformity and standardization across those approaches. ODMG provides the basis for extending object-oriented databases with new capabilities. Thus, we propose extending the ODMG standard with triggers. Our aim, however, is not to propose a new active rule language. Rather, we adapt the SQL:1999 trigger language to ODMG. This choice is mainly motivated by the convergence between object-oriented and object-relational approaches. Of course the shift from the SQL:1999 data model to a pure object-oriented data model requires a re-examination of the trigger language proposed for SQL:1999.

In particular, the development of a trigger model for ODMG entails addressing several issues. First of all, triggers are usually associated with data manipulation operations. However, no ad-hoc data manipulation language has been proposed for ODMG. While standards for data definition and query languages have been developed for the model, the issue of updates has been largely neglected. Data manipulation has been considered only when defining the bindings to C++, Smalltalk, and Java. A number of ad-hoc data manipulation primitives, resembling the ones found in SQL, should in our opinion be provided as part of the ODMG model. Indeed, object-oriented programming languages have not been designed to manipulate large amounts of data. Predefined primitives, well-suited for data manipulation, implemented at system-level, are missing. Therefore, as part of our work, we propose a set of primitives to facilitate ad-hoc manipulation of data. Another issue to be addressed is set-orientedness. In the database context, the common case is to execute a given update operation on a set of objects rather than on a single object. In particular, in databases DMLs support *sets* of instances as logical units of computation, whereas conventional programming languages reason on a single instance (*record*) at-a-time. Data manipulation primitives in SQL also have a set-oriented semantics, that is, they work on a set of objects at-a-time. We thus also introduce *set-oriented* data manipulation in ODMG, still retaining the *instance-oriented* approach of the current standard. Thus, data manipulation primitives in our approach can be executed both with a set-oriented or with an instance-oriented semantics, and the same applies also to trigger execution (this corresponds to *row-level* and *statement-level* triggers of SQL:1999). Method execution, by contrast, can only be carried on in an instance-oriented way. A set-oriented method execution would indeed require to change the semantics and the implementation of object-oriented programming languages to accommodate set-orientedness. In this thesis we discuss and compare instance-oriented and set-oriented executions, by taking into account that in our proposal both these executions are possible for data manipulation primitives as well as triggers. In SQL:1999, by contrast, the two possibilities are supported only for triggers, whereas data manipulation is always set-oriented. In adapting the SQL:1999 trigger definition language to the ODMG data model, we also investigate some issues concerning trigger *inheritance* and *overriding* which have not been considered in the context of SQL:1999. Such issues have not been thoroughly addressed by other proposals of active object-oriented data models; however, they are crucial for a proper integration of reactive capabilities with object-oriented modelling primitives [BGM97]. In most active object-oriented database systems triggers are always inherited and can never be overridden nor refined. Such an approach, referred to as *full trigger inheritance*, simply means that event types are propagated across the class inheritance hierarchy. 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. Full trigger inheritance is, however, not appropriate in all the situations. Trigger overriding is as crucial for behavior specialization as method overiding. Thus, in our approach, we consider the possibility of redefining triggers in subclasses instead of simply inheriting them. In the thesis, we also formally define the semantics of the proposed language. This is another contribution of the thesis.

The thesis is organized as follows.

Chapter 1 presents the existing literature related to the thesis work. The aim of this chapter is twofold: to illustrate the historical background which led to the definition of a standard for database management systems ODMG 3.0 and to present the most significant literature in the fields of temporal and active OODBMSs.

In Chapter 2 the object model of ODMG 3.0 is formally defined. The proposed framework is the formal basis on which the temporal and the active model will be built.

In Chapter 3 we present T_ODMG, a temporal extension of the ODMG object model, which allows one to manage temporal data expressed with respect to different temporal granularities. In this chapter, we formally define the main notions of the temporal data model and address issues related to inheritance, type refinement, and substitutability. Finally, we describe the implementation of the presented model on top of an ODMG compliant DBMS.

In Chapter 4 we investigate how temporal data expressed with respect to different granularities can be queried. In particular issues related to the combination and the comparison of values with different granularities and navigation through objects are addressed.

Chapter 5 extends ODMG with reactive features, by proposing a language for specifying triggers and defining its semantics. Data manipulation in ODMG is also addressed.

Finally some concluding remarks and topics for future work are presented in the conclusions.

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

Chapter 1

Related Work

In this chapter we present the existing literature related to this thesis. The aim of this chapter is twofold: to illustrate the historical background which led to the definition of a standard for database management systems ODMG 3.0, and to present the most significant literature in the fields of temporal and active OODBMSs.

In addition, some notions that will be extensively used in the thesis, such as the different time dimensions which can be considered in a temporal database and the notion of ECA rule, are introduced.

We do not provide here a detailed description of the ODMG object model since this chapter is mainly a non-technical overview. Such a description can be found in the next chapter in Section 2.1. Moreover, OQL is described in Section 4.1 and data manipulation in ODMG is described in Section 5.2.

This chapter is organized as follows. Section 1.1 is devoted to the presentation of the historical background which led to the definition of the ODMG standard. In addition, in this section the literature dealing with the ODMG standard is proposed. In Section 1.2 the most relevant works on temporal object-oriented databases and temporal granularities is illustrated. Finally, Section 1.3 describes the work on active object-oriented databases and presents SQL triggers which have strongly influenced our work on the active extension of the model.

1.1 Object-Oriented Databases and ODMG

In this section we first present the historical background which led to the definition of a standard for OODBMSs. Then, in Subsection 1.1.2, the existing work dealing with the

ODMG standard is presented. In particular, we first describe work dealing with various analysis of the ODMG standard and then extensions to it.

One of the most recent publications on OODBMSs is a book edited by Chaudhri and Zicari [CZ00]. In this book, after a comprehensive introduction on OODBMs, Chaudhri and Zicari provide a collection of real-world case studies and examples that demonstrate how some of the world's leading companies and research institutions are promoting the use of OODBMSs to build robust databases. The content of the book ranges from the state of the art, to the use of new technologies, such as XML, and performance evaluation. Each chapter of this book is written by one or more experts in the field of OODBMSs from academia or from industry. Among the chapters we refer to [BG00] for the historical perspective presented in the first part of this section.

1.1.1 Historical Background

Strong experimental work and the development of several prototype systems have characterized research in the area of object-oriented databases, whereas theoretical foundations were investigated and then the standards developed. These trends represent, in some way, the reverse of what happened for relational DBMSs (RDBMSs), for which the development of systems began after the formal definition of the data model.

The late 1980s saw the birth of OODBMSs. Since then, such systems has undergone intense industrial development. In the space of a few years, different generations of OODBMSs were developed. In [BG00] three generation of OODBMSs have been devised.

The first generation of OODBMSs dates back to 1986 when G-Base, the first OODBMS appeared on the market. From 1986 to 1989 several different software products that can be classified as first generation OODBMSs have been produced. The common goal of this group of products was to support persistence for object languages, in particular languages related to artificial intelligence. The distinguishing feature of these systems was that they were stand-alone, based on proprietary languages and did not use standard industrial platforms. In 1990, the total number of systems installed by those companies was perhaps between 400 and 500, and the systems were mainly located in the research departments of large companies.

The release of Ontos (from the company with the same name) in 1989 marked the beginning of the second generation in the development of OODBMSs. Products from Object Design, Objectivity and Versant Object Technology (ObjectStore, Objectivity/DB and Versant ODBMS, respectively) followed soon after. Compared to the first generation of OODBMSs, all the second generation systems used a client/server architecture and a common platform: C++, X Windows and UNIX workstations. Those companies are still the major producers of OODMBSs nowadays.

Only a few months after the second generation OODBMSs, the first product of the third generation, Itasca, was released in August 1990. Among the other third generation OODBMS we would like to cite O_2 which has strongly influenced the ODMG query language.

Whilst first generation OODBMSs can be considered object-oriented languages with persistence, the third generation ones can be defined as DBMSs with advanced characteristics (for example, version support) and with data definition and data manipulation languages which are object-oriented and computationally complete. Beyond the technical differences (in architectures and functionalities), third generation DBMSs are the result of long-term research projects run by large organizations seeking to capitalize on their investments. Therefore, they are very advanced systems from the viewpoint of both database technology and software development environments.

From this brief historical background, one can deduce that the beginning of OODBMSs was characterized by the development of a large number of systems, most of which were produced by small vendors, that were revolutionary with respect to previous DBMSs in that they were built from scratch with a different code base and a different data model. In this initial period, the research community felt the need of at least defining what an OODBMS was. Thus, in 1989, the Object-Oriented Database System Manifesto [ABD+89] was published. This document describes the main features and characteristics that a system must have to qualify as an OODBMS. This document can be considered as the ancestor of ODMG.

The OODBMS panorama at the beginning of the 1990s was characterized by a quite large number of systems, lacking a common data model and whose use in applications was still at the experimental stage. One of the reasons for the slow growth in OODBMSs was the resistance by customers and companies in migrating to new technologies. However, a common feeling within the OODBMS community was that the lack of a standard for object databases was the major limitation for their widespread use.

The success of RDBMSs did not simply result from a higher level of data independence and a simpler data model than previous systems, but also from the standardization they offered. The SQL standard offers a high degree of portability and interoperability between systems. It simplifies learning new RDBMSs, and represents a wide endorsement of the relational approach. All these factors are important for OODBMSs as well. Actually, these factors are even more important for OODBMSs, because companies developing most products in this area are young, small companies: portability and endorsement of the approach are thus essential to customers. In addition, the scope of OODBMSs is more far-reaching than that of RDBMSs, because OODBMSs integrate the programming language and the database system, and encompass all of an applications operations and data.

Thus, in the summer of 1991, the ODMG consortium was formed. The primary role of this group was to develop a set of standards allowing an OODBMS customer to write portable applications, that is, applications that could run on more than one OODBMS. ODMG member companies represented almost the entire OODBMS industry. This standardization effort falls within the common effort of promoting the definition of standards for the object-oriented world.

ODMG is affiliated with the Object Management Group (OMG), established in 1989, and tried to benefit as much as possible from the work developed by that group. The main contribution of OMG has been the definition of the CORBA architecture [OPR96] for distributed object systems interoperability. Since OODBMSs provide an architecture for developing applications that is significantly different from RDBMSs, the nature of a standard is also substantially different. Rather than providing only a high-level language such as SQL for data manipulation, an OODBMS transparently integrates database capability with an application programming language. Thus, in contrast to RDBMSs, OODBMSs require standards based on integration with existing programming languages. OODBMSs have been integrated with C++, C, Smalltalk, and LISP. Overlap with the RDBMS standard only concerns the query language component of the standard.

The ODMG standard was first released in 1993, followed by release 1.2 of the standard, published in a book [Cat94]. The ODMG standard included a reference object model (ODMG object model), an Object Definition Language (ODL) and an Object Query Language (OQL). Moreover, it defined the C++ and Smalltalk programming language bindings for ODL and data manipulation. The object model specifies the standard data model for object-oriented databases and ODL is the Data Definition Language (DDL) that is provided to define an object database schema according to that model. OQL is a declarative query language, strongly inspired by the SQL query language, whereas the language bindings specify how the object model concepts are mapped into the considered languages, and how ODMG objects can be accessed and manipulated by the languages. An excellent work presenting the history of OQL is [Clu98]. The ODMG object model was defined as a superset of the OMG object model, and ODL extended the OMG IDL (Interface Definition Language).

To these preliminary releases of the standard two others followed. ODMG 2.0 [CBB+97] was released in 1997. In [CBB+97], as a consequence of the development and increasing popularity of Java, Java programming language binding has been added to the C++ and Smalltalk bindings defined in the first version. The influence of Java in the release 2.0 is also reflected in many changes to the object model, such as the notion of interface and the introduction of two different inheritance hierarchies. Finally, at the end of 1999 the last release of the standard, ODMG 3.0 [CBB+99] has been published. The ODMG 3.0 does not present any relevant different feature with respect to release 2.0. In particular, the object model, the object definition language, and the object query language are almost the same and no other language binding has been added. Some subtle differences of the ODMG

3.0 object model with respect to the object model of the release 2.0 will be discussed in Section 2.2.

1.1.2 ODMG-Related Work

The publications related to the ODMG standard are really many. Once a new release of the standard was published, members of the group presented the standard in international database forums producing in this way many related publications. The aim of such publications was mainly to advertise the ideas contained in the standard itself. Thus, we do not mention such works here. By contrast, we focus on two different type of works which are related to ODMG: works devoted to analyze and formalize the standard and the ones aiming at extending it with new features.

One of the first papers reporting an analysis of the ODMG object model is by Alagić [Ala97]. In his paper, he shows several problems of the ODMG object model concerning the type system and the persistence model. He also points out discrepancies between the ODMG object model and the language bindings. The paper proposes solutions to those problems together with a formal type system for ODMG. Since the paper has been written with reference to the ODMG-93 model, the corresponding formal type system does not consider many features that have been introduced in subsequent releases. Moreover, the goal of the work by Alagić is to formalize typing aspects, rather than to provide a comprehensive formal definition of the object model. Thus, the type system does not distinguish between attributes and relationships, and does not consider some other features of the data model, such as extents and keys. For what concerns the persistence model the main problems pointed out in the paper have been fixed in ODMG 3.0 object model, though some problems remain with respect to the Java binding. For what concerns the typing aspects, by contrast, the problems pointed out by Alagić still persist in ODMG 3.0. Typing aspects in ODMG 3.0 are discussed in detail in Section 2.2. Finally, in contrast to what the model claims, in [Ala97] the notions of inheritance and subtyping are distinguished and the introduction of two separate hierarchies is advocated.

After his first work in 1997 Alagić continued working on ODMG and now he is part of the invited staff of the ODMG group. One of his most significant work on ODMG is [Ala99c] in which he proposes an approach to type check OQL queries. The work in [Ala99c] can be regarded as the mature expression of the one presented in [Ala97]. Based on the preliminary results of [Ala97], in [Ala99c] Alagić shows that it is not possible to type check OQL queries in the type system underlying the ODMG object model and object definition language. In addition, he shows that OQL queries cannot be type checked in the type system of the Java binding of the ODMG standard either. The proposed solution is similar to the one proposed in [Ala97]. The ODMG object model is extended to support parametric polymorphism. Polymorphism provides a general approach to describing data or algorithms where the structure is independent of the type of element manipulated. Type variables are used for this purpose. Parametric polymorphism, however, is not sufficient to properly model all the situations. Thus, F-bounded polymorphism is added to the model. Bounded polymorphism is a particular case of polymorphism, where a type variable may take on any class that is subclass of a given class. F-bounded polymorphism is in turn a particular case of bounded polymorphism where the bound is recursive. These advanced typing techniques are not included in the standard and it might be an argument of debate whether they will be included in future releases (cf. Section 2.2).

The last works by Alagić on ODMG are [Ala99a] and [Ala99b]. In [Ala99a] a family of ODMG object models is proposed as a solution to the problems of the ODMG standard which he pointed out in his previous works. Those problems are related to the type systems, the model of persistence, and the model semantics. Each model in the family is well-defined and avoids the inconsistencies of the ODMG object model. According to the author point of view, this family is intended to replace the existing single and inadequate ODMG object model. In [Ala99b] an evaluation of the level of compliance of a commercial OODBMS, O_2 [BDK92] with respect to ODMG is presented. Several reasons led the author to choose O_2 for this analysis. O_2 is claimed to be ODMG compliant. The ideas behind O_2 have obviously influenced the standard in many ways. An additional, but not a decisive reason, was the personal experience of the author in using O_2 in developing a complex application.

A paper by Riedel and Scholl was published at about the same time of [Ala97], aiming at formalizing ODMG [RS97]. Actually, the aim of such work is the formalization of ODMG queries, in order to investigate object-oriented query optimization. To formalize OQL queries, the starting point, however, is the formalization of the object model. Such a formalization refers to a subset of the object model of the release 1.2 of ODMG. Thus, many of the interesting aspects of the ODMG 3.0 object model are not addressed. In [RS97], however, the drawbacks and deficiencies of the ODMG object model, pointed out in [Ala97], are not addressed.

Sakkinen also strongly criticized the ODMG object model (release 1.2) that he described to be premature and incomplete but still to be regarded as a good starting point [Sak96]. In his paper, which has never been formally published even though it was very relevant, he pointed out, surprisingly, that not even all companies that were voting members of the ODMG looked active in promoting and implementing ODMG. His work mainly pointed out the inconsistencies and obscure points in the standard specification. Many of his remarks applies still to ODMG 3.0. The paper, however, is an informal description of the standard, and no formalization is proposed.

Kim [Kim94] considered ODMG 1.2 not to be a standard, rather a work-in-progress proposal for an object-oriented database language. Kim underlined the lack of a formal basis and he suggested what the ODMG group should have done both from a theoretical and practical point of view. The work done by Kim had a great impact on the ODMG group, so that a discussion between the two started. The ODMG group provided in a paper [Gro94] an answer to Kim's paper. In that work they presented their own point of view on several aspects that are matters of taste, or on which reasonable people may disagree, but where Kim had implied his taste and opinion to be the only right one. Moreover, in [Gro94] they presented the future course for ODMG.

Finally, let us mention [HK97], a theoretical work on the categorical formalization of the ODMG object model. In [HK97] the attention was focused on polymorphism and relationships satisfying required integrity constraints. Again, this work was based on the ODMG 1.2 object model, thus many interesting features which have been introduced in subsequent releases of the standard have not been considered.

In spite of all the various criticisms, ODMG is recognized to be the standard for object-oriented database systems. A proof of that are the several proposals extending the standard with new features. Among such proposals we would like to mention: a declarative language for defining integrity constraints [FMP97], a language for defining composite objects [BG98], a deductive query language [SP98], and a multidatabase layer for the standard model [RKJ98].

The query language of ODMG, OQL, has been the topic of interest of many interesting works. An excellent work presenting the story of OQL is [Clu98]. Among the works dealing with OQL let us mention the ones dealing with query optimization and visual query interfaces.

A quite recent work dealing with OQL query optimization is [RS97] that we have previously discussed. Another recent work dealing with optimization of OQL queries is [vB99]. In [vB99] a graphical technique for semantic query optimization for ODMG compliant databases. The author argues that his approach generalizes previous research on object-based query optimization by providing a more natural representation of the query allowing simpler, more intuitive transformations, and leading to more efficient evaluation plans.

A remarkable work dealing with ODMG OQL is the one developed at the University of Austin [FSRM00]. The λ-DB project described in [FSRM00] aims at developing frameworks and prototype systems that address the new query optimization challenges for object-oriented databases. The built system is ODMG compliant. Inside such a project a visual query formulation interface, called VOODOO, for OQL queries has been developed [Feg99].

Visual query languages for OQL queries have been addressed also in [CW97]. In [CW97] the design, implementation and user evaluation of QUIVER, a graph-based visual query language for ODMG compliant object-oriented databases. The design goals of QUIVER include compliance to standards, comprehensive representational power, and consistency of visual representation. The language is implemented as a visual front-end to the O_2 object database system. Results of a user evaluation suggest that users find it easier to formulate correct queries in QUIVER than in OQL. Finally, in [MGP98] Kaleidoquery, a visual query language for object databases with the same expressive power of OQL is presented. One of the peculiarities of Kaleidoquery is that the querying environment is three dimensional.

ODMG does not offer the possibility of dealing with time-varying data. The importance of managing time-varying data has been widely recognized by the database community and led to an extension of database technology, that is, temporal databases. As far as temporal databases are concerned, an extension of the ODMG object model supporting time management has been presented in [BFGM98]. Next section will survey the existing literature on temporal object-oriented databases.

Schema versioning in ODMG is addressed in [GM99, LK99]. Schema versioning is the process of changing the schema of an object database while maintaining its consistency. The schema can change, for instance, in response to the varying needs of the application. The importance of schema versioning in databases has been recognized by the scientific community. In [GM99] an extension of ODMG ODL and OQL to support a generalized schema versioning is presented. Such an extension aims at making the version facilities available at user-interface level and is the basis for the development of advanced multi-schema applications. In [LK99], the design and implementation of the SOP Object Version System which is based on SOP, an ODMG-compliant object database system is described. To support object versioning the SOP Object Version System provides a set of APIs (application programming interfaces) as a class library. One important design goal of SOP Object Version System was to achieve full compliance with ODMG standard.

Finally we would like to mention the work done by the database group of the Stanford University on extending the ODMG type system for managing semistructured data [AQM+96].

1.2 Temporal Object-Oriented Databases

Conventional database systems do not offer the possibility of dealing with time-varying data. The content of a database represents a *snapshot* of the reality in that only the current data are recorded, without the possibility of maintaining the complete history of data over time. If such a need arises, data histories must be managed at application program level, thus, making the management of data very difficult, if at all possible. Temporal database systems overcomes this lack. Indeed, temporal database systems efficiently manage not only the current value of data, but also the entire history of data over time. Issues related to temporal data models and query languages have been extensively investigated from the theoretical point of view [WJW98]. Most of the research and development efforts in the area of temporal databases have been carried out in the context of the relational model.

In 1995, TSQL2 [Sno95b], the temporal extension of the SQL-92 relational database standard has been proposed. In the new standard SQL:1999 [GP99, Mel99] temporal features have not yet been introduced. In [EM00] the reasons for such a delay are investigated. The research on the temporal extension to SQL stalled about three years ago for three reasons. First, there was not a lot of enthusiasm from SQL vendors for adding temporal support to database engines. Second, people working on the standard concentrated on completing SQL3 consolidating what was already part of it instead of introducing new facilities. Finally, there is still a fundamental disagreement on how temporal capabilities should be provided by the standard. However, recently the interest in adding temporal support to the standard has increased. Thus, according to what is stated in [EM00], temporal support will be probably published as an incremental part of the standard. A very recent book by Snodgrass [Sno99] presents effective techniques for designing and building relational database applications that must integrate past and current data.

The relevance of multiple granularities in the context of temporal applications is well recognized by the scientific community. A considerable amount of work on multiple granularities has been developed from a theoretical point of view, investigating the meaning and the properties of temporal granularities. Considering the database field, most of the work has been carried out in the context of temporal relational database systems. For instance, TSQL2 supports multiple granularities.

The aim of this section is to illustrate the most relevant literature on both the research topics involved in the thesis work: temporal object-oriented systems and temporal granularities.[1]

1.2.1 Temporal Object-Oriented Systems

One of the first issue which has to be examined in temporal databases is the *time domain*, that is, the set of primitive temporal entities used to define and interpret time-related concepts. In particular, its *structure* and *dimensionality* have to be established.

Temporal logic represent time as an arbitrary set of instants with an imposed partial order. *Linear* time can be specified by adding an axiom imposing a total order on this set. In the linear model, time advances from the past to the future in a step-by-step fashion. Another possible model is the *branching* model. In the branching model time is linear from the past to now, where it then divides into several time lines, each representing a potential sequence of events. Axioms may also be added to temporal logics to characterize the *density* of the time line. Combined with the linear model, *discrete* models of time are isomorphic to natural numbers, implying that each point in time has a single succesor. *Dense* models of time are isomorphic to either rational or the reals: between any two moments of time another moment exists. *Continuous* models of time are isomorphic to the reals, that is,

[1] These issues will be investigated in the thesis in Chapters 3 and 4.

they are both dense and, unlike the rationals, contain no "gaps".

In the discrete model, each natural number corresponds to a nondecomposable unit of time with an arbitrary duration. Such a nondecomposable unit of time is called *chronon*. Thus, a chronon is the smallest duration of time the can be represented in a model.

Table 1.1 compares some temporal object-oriented data models proposed in the literature. Most models presented in Table 1.1 support a *linear discrete* time structure, whereas only few of them model a user-defined hierarchy of time types. Several practical arguments justify this choice. First, measures of time are inherently imprecise. Second, most natural language references to time are compatible with the discrete time model. Third, the concepts of chronon and period allows to naturally model events that are not instantaneous but have a duration. Finally, any implementation of a data model with a temporal dimension will of necessity have to have some discrete encoding for time.

In the context of databases, two time dimensions are of general interest: *valid* time and *transaction* time. Valid time concerns the time a fact was true in reality. The valid time of an event is the time at which the event occurred in the real word, independent of the recording of that event in some database. Valid times can also be in the future, if it is expected the some fact will be true at a specified time after now. Transaction time concerns the time the fact was stored in the database. The transaction time of a fact identifies the transaction that inserted the fact into the database and the transaction that removed this fact from the database. These two dimensions are orthogonal. A data model supporting neither is termed *snapshot*, as it captures only a single snapshot in time of both the database and enterprise that the database models. A data model which supports only valid time is termed *valid-time*, one that supports only only transaction time is termed *transaction-time*, and one that supports both valid and transaction time is termed *bitemporal*. *Temporal* is a generic term implying some kind of time support. There is a third kind of time that may be included: *user-defined* time. This term refers to the fact that the semantics of these values are known only to the user, and are not interpreted by the DBMS as differentiated from valid and transaction time, whose semantics are supported by the DBMS. Most models in Table 1.1 consider only the valid time.

An important feature which distinguishes the approaches presented in Table 1.1 is the modelling entity to which time is associated. Usually, in temporal object-oriented databases objects or attributes can have an associated time. Some approaches associate a timestamp, that is, a time value, with the whole object state; others associate a timestamp with each object attribute often regarding the value of a temporal attribute as a function from a temporal domain to the set of legal values for the attribute. This is also the approach taken in the temporal model presented in the thesis.

Another important characteristic is whether temporal, immutable and non temporal attributes are supported. A *temporal* (or historical) attribute is an attribute whose value may change over time, and whose values at different times are recorded in the database. An *immutable* attribute is an attribute whose value cannot be modified during the object lifetime,[2] whereas a *static* (or non-temporal) attribute is an attribute whose value can change over time, but whose past values are not meaningful, and are thus not stored in the database. The support for temporal, immutable, and static attributes has been firstly proposed in the *T*_Chimera data model [BFG96, BFG97] and has then been adopted by the data model presented in [BFGM98].

In all these approaches, except than [BFGM98], one can notice the lack of a common basis and the complete neglecting of standardization issues. One of the main reasons is that the ODMG standard is recent, in particular compared to the SQL standard, and not yet well-established. The definition of the standard has represented for the object-oriented database community a common basis to define extensions of object-oriented databases, compliant with the standard, in many research directions, such as with temporal features.

[BFGM98] has been the first attempt to produce an extension of the ODMG object model to manage the time dimension. The model presented in [BFGM98] does not support multiple granularities. However, it can be considered as the basis on which the model presented in the thesis has been developed.

	[CG93]	[CCP97]	[OPS+95]	[KS92]	[EdOP93, EdOP94]
o-o data model	generic	ONTOS	Tigukat	MAD	F-ORM
time structure	linear discrete	linear discrete	user-def.	linear discrete	linear discrete
time dimension	valid	valid	valid + trans.	valid	valid + trans.
values & objects	objects	objects	objects	objects	both
relationships	NO	NO	NO	NO	NO
what is timestamped	attr.	objects	arbitrary	objects	arbitrary
temporal attribute values	funct.[2]	funct.[2]	sets of pairs	atomic valued[3]	sets of pairs
kinds of attributes	temp. + imm.	temp. + static	temp. + imm.	temp. + imm.	temp. + imm.
granularities	NO	YES	YES	NO	YES
QL	OOTempSQL	GCH-OSQL	TQL	TMQL	YES

Legenda:

[1] A single time dimension is considered, it can be transaction or valid time.
[2] With *funct.* we denote functions from a temporal domain.
[3] Time is associated with the entire object state.

Table 1.1: Comparison among temporal object-oriented data models

[2]Immutable attributes can be regarded as a particular case of temporal ones, since their value is a constant function from a temporal domain.

	[RS91]	[BFG97]	[WD93]	[SN97]	[BFGM98]
o-o data model	generic	Chimera	Oodaplex	OM	ODMG
time structure	linear discrete	linear discrete	user-def.	linear discrete	linear discrete
time dimension	valid + trans.	valid	arbitrary[1]	valid	valid
values & objects	objects	both	objects	both	both
relationships	NO	NO	NO	YES	YES
what is timestamped	attr.	attr.	arbitrary	objects relat.	attr. relat.
temporal attribute values	lists of tuples	funct.[2]	atomic valued[3]	funct.[2]	funct.[2]
kinds of attributes	temp. + imm.	temp. + imm. + static	temp. + imm.	temp. + imm.	temp. + imm. + static
granularities	YES	NO	YES	NO	NO
QL	TOSQL	YES	YES	YES	YES

Legenda:

[1] A single time dimension is considered, it can be transaction or valid time.
[2] With *funct.* we denote functions from a temporal domain.
[3] Time is associated with the entire object state.

Table 1.1: Comparison among temporal object-oriented data models (continued)

Once a data model has been defined, the need of querying objects expressed in such a data model arises. Indeed, usually, the definition of a new data model includes the definition of a new query language. Some query languages are compared in [Sno95a]. Recent works on temporal object-oriented query languages not mentioned in [Sno95a] are [FE98].

Though several temporal object-oriented query languages have been proposed in the literature, most of them do not consider multiple granularity data. Moreover, most of them are extensions of relational query languages rather than of existing nontemporal object-oriented query languages. Object-oriented database systems provide, in addition to traditional query languages, a *navigational* approach to data access. The navigational approach is based on object identifiers and aggregation relationships: given an oid, the system directly accesses the corresponding object and navigates through objects referred to by its components. This access modality can be combined with the query-based (e.g., SQL-like) access. Thus, conditions in a query can be imposed on the nested attributes of the queried objects; such nested attributes are in most cases denoted by *path expressions*, or similar syntactic notations. Among the few works addressing issues concerning object navigation in a temporal object-oriented data model we mention [BFG98].

1.2.2 Multiple Granularities

The pioneering research work on temporal granularities is [And82], which firstly pointed out the need to model time at multiple granularity level. After such work, many other works aimed at understanding and formalizing what temporal granularities are and how they can be managed properly have been published. Among them we mention [BWJ98, CR87, GLO$^+$98, MMCR92, WJL91, Ter97]. All of them are characterized by the search for a common basis on top of which one can reason on multiple granularities in temporal databases.

We believe that such a goal has been achieved by the consensus glossary of time granularity concepts [BDES98]. In such a work Bettini et al. clearly formalize all the main notions related to multiple granularities and established several interesting relationships among granularities.[3] Our work, starting from [BDES98], investigates the impact of temporal granularities in an object-oriented model compliant with the ODMG standard. Thus, the theoretical basis presented in [BDES98] and accepted by the temporal database community, is exploited as a foundation for our work.

Temporal granularity issues related to temporal databases have been investigated both in the relational and the object-oriented data models. For what concerns temporal relational databases, the SQL standard offered a common basis from the database side which has been given rise to TSQL2, the standard for temporal relational databases. In its first release [Sno95b] TSQL2 supported multiple granularities, but many important issues, such as scaling from one granularity to another, were not considered. A very recent paper [DELS00] solves many of the problems of TSQL2 related to the treatment of multiple granularities. Like us, in [DELS00], the authors elaborates on the theoretical framework of [BDES98].

Recently, a comprehensive book on temporal granularities in databases, data mining, and temporal reasoning has been published [BJW00]. This book is mainly a collection of papers which are the contribution to the field of the three authors: Bettini, Jajodia, and Wang. [BJW00] deals with several aspects of temporal information and provides a unifying model for granularities. The underlying data model considered in [BJW00] is the relational one. Among the works by those authors inserted in [BJW00] let us mention [BWJ98]. In [BWJ98] the concept of *semantic assumption* for temporal databases is formalized. A semantic assumption is a way of deriving implicit information from explicitly stored (relational) data. Thus, they allow one to query data stored with respect to a granularity with respect to another granularity. These way of reasoning has been extensively adopted in temporal databases. In the thesis we will investigate somehow the use of semantic assumption in temporal object-oriented databases. Indeed, coercion functions (cf. Chapter 3) presented in our approach can be viewed as particular case of semantic assumptions.

[3] A detailed description of the main contributions of [BDES98] can be found in Section 3.1.

By contrast, no comparable amount of work has been carried out in the context of the object-oriented data model to introduce temporal granularities. The introduction of multiple temporal granularities in an object-oriented data model poses additional issues with respect to the relational context, due to the semantic richness of such a model. Most of the temporal object-oriented data models proposed so far do not deal with temporal granularities (cfr. Table 1.1). The only ones dealing with temporal granularities [CCP97, OPS+95, RS91, WD93] support multiple temporal granularities as extensions to the set of types of the temporal model. However, in most of these approaches the specification and management of different granularities, e.g., how to convert from a granularity to another, is completely left to the user. None of them refers to the ODMG standard object model, which implies a strong dependence of each temporal model from the reference object model. Moreover, in contrast to the relational context, the introduction of temporal granularities in object-oriented data models is, in most cases, informal. For instance, none of the proposed approaches consider the impact of inheritance when multiple granularities are supported, nor they address issues concerning consistency, type refinement, substitutability, and object navigation. We believe that these are important topics to be investigated and that their treatment on a formal basis is a means to manage their complexity.

1.3 Active Object-Oriented Databases

An active database system is a system in which some operations are automatically executed when specified events happen and particular conditions are met. Active features are critical for many advanced data management applications. Examples of use of active capabilities are integrity constraint enforcement, authorization and monitoring. 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 to do when the rule is triggered and its condition is true.

In both the relational and the object frameworks active rules provide a comprehensive means to formally state the semantics of data, the high-level semantic operations on data and the integrity constraints. Even though current relational database systems provide stored procedures, and object-relational and object-oriented database systems provide methods, as a means to express behavior of data, specifying the semantics of data through rules has important advantages over coding it into methods (or stored procedures). For instance, the behavior represented by methods must be explicitly invoked by the user or by applications, whereas active rules are autonomously activated. Moreover, 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.

Looking at the history of active databases one can notice that the attitude towards active features of relational and object-oriented databases has been very different. Indeed, active features have been quickly included in commercial systems by relational DBMS producers and triggers are now part of the SQL:1999 standard. By contrast, in the last ten years, several research projects have been carried on aiming at extending object-oriented database systems with reactive capabilities, but, until now, no commercial product includes triggers. It is difficult to answer why this happened. However, we believe that probably object-oriented database are not yet mature to be extended with advanced features, such as triggers, and we have to wait some extra year to have triggers in OODBMSs. We believe that our work can be regarded as a step in this direction. Indeed, the inclusion of active capabilities in object-oriented products would certainly benefit from appropriate extensions to the ODMG standard.

In this section we first present active features in existing object-oriented databases. Then, we present triggers in the SQL:1999 standard [GP99, Mel99]. SQL triggers have strongly influenced the introduction of triggers in ODMG presented in the thesis (cf. Chapter 5). Appendix B includes some tables summarizing the main features and restrictions of different kinds of triggers, together with an Example of trigger in SQL:1999.

1.3.1 Active Object-Oriented Systems

The most important surveys on active databases are [CW96] by J. Widom and S. Ceri which was published in 1996 and [Pat99] by N. Paton which has been recently published. Both these surveys are collections of papers describing the main concepts and features of active databases, and presenting the proper characteristics of several systems.

One of the earliest object-oriented active database projects is HiPAC [DBC96, MD89]. HiPAC provides an expressive active database rule language with flexible execution semantics, supporting a full range of coupling modes through a nested transaction model. There are some projects on active object-oriented databases that are follow-on from HiPAC: among them, *Sentinel* [AMC93] and REACH [BBKZ93]. One of the best known active object-oriented database systems is Ode [GJ91, GJ96, GJS92], 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 EXACT [DJ97, DPG91], NAOS [CCS94, CC95], TriGS [KRSR94, KR98], SAMOS [GD92, GFV96], Chimera [CFPT96, BGM97], RAP [BFP$^+$95], and PFL [RPS95].

In Table 1.2 we compare the most significant systems cited above along a number of dimensions. Among the considered systems, NAOS, TriGS and SAMOS are extensions of commercial object-oriented database systems with active capabilities. All the proposals support *event-condition-action* (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 (for example object creations and deletions), temporal events, external (that is, raised by the application), and user-defined events. In addition, in some systems it is possible to specify whether a rule must be triggered before or after 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 or actions.

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 to 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 in terms of 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. 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 rules and set-oriented for deferred ones. Immediate and deferred rules are presented below.

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 earliest 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* [MD89]. 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 following, within the same transaction.
- *Deferred*: takes place at the commit point of the current transaction.
- *Decoupled*: takes place in a separate transaction.

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.

Another difference among existing systems is whether rule definitions are attached to classes (*targeted rules*). Attaching rule definitions to classes enhances modularization and allows an efficient detection of relevant events. However, there are sometimes useful rules triggered by events spanning sets of objects possibly from different classes (*untargeted rules*).

The *priority* notion is also extremely relevant for active rule semantics. 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 in order to achieve the correct behavior, while numeric priorities can be difficult to use since they may need to be adjusted as the set of rules evolves.

Another important aspect in introducing triggers in object-oriented databases concerns trigger inheritance and overriding. Since inheritance is one of the most significant features of the object-oriented paradigm this is a fundamental problem to cope with. In spite of that, the influence of inheritance on triggers has not been deeply investigated in existing object-oriented database systems. Under some proposals [BM91, CCS94, MP91], triggers are always inherited and can never be overridden nor refined. Such an approach, referred to as *full trigger inheritance*, simply means that event types are propagated across the class inheritance hierarchy. 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. Full trigger inheritance is, however, not appropriate for all the situations. Moreover, the meaning of the ISA hierarchy is just to define a class in terms of another class refining its attributes, methods and triggers. Thus, in our opinion, the possibility of redefining triggers in subclasses instead of simply inheriting them is needed in active object-oriented systems. For what concerns the analyzed systems, rule overriding is supported in TriGS, in Ode, under some restrictions, and in NAOS can be achieved using trigger priorities. The problems concerning trigger inheritance and overriding in Chimera have been investigated in [BGM00].

Most active database systems are specified in an informal and imprecise manner. However, formal description techniques have been used to give a semantics to specific systems, but there is no generally accepted approach. A survey on formal techniques analyzing the main characteristics of some active database systems such as the rule definition language and the execution model has been published in 1995 [PCFW95].

Among the approaches to the definition of a semantics mentioned in [PCFW95], [Wid92] is very similar to ours. [Wid92] 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 line of thinking, devised in [PCFW95], has adapted the semantics of deductive databases to active systems. The most significant works in this direction are [Kow92], [Zan94], and [FWP97].

The main works subsequent to the survey are [CPW97, FT95, BGM00]. In [CPW97] a generic framework for formally specifying the semantics of different systems is proposed, relying on the Object-Z formalism [Ros92]. The formalism used is object-oriented but totally abstract. The main drawback of this approach is that no aspect concerning the active rule language is analyzed. In [FT95] an approach, conceived for Chimera, providing a formal model onto which the various aspects of such a system, such as execution of a set of rules, coupling modes, consumption modes and so on, can be mapped, is proposed. Finally, we would like to cite [BGM00] where a semantics for an active object language that takes into account trigger inheritance and supports trigger overriding is formally defined.

The overview presented in [PCFW95] and the very few works that followed highlight the fact that most active database systems are informally specified and that there is no generally accepted approach. We believe that our work, which provides a trigger language for the ODMG standard and formally defines its semantics, can be seen a step forward in this direction.

1.3.2 Triggers in SQL:1999

Table 1.3 summarizes the most relevant features of triggers in SQL:1999, considering the same dimensions that have been taken into account in Section 1.3.1. Additional tables on SQL features and an example are included in Appendix B. The trigger language of SQL:1999 is very close to those of the best known object-relational DBMSs, such as Oracle [Ora97], DB2 [Cha96], and Sybase [SYB]. Triggers are expressed by means of event-condition-action (ECA) rules. In SQL:1999 each trigger reacts to a single event. Considered events are insert, delete, or update to a particular relation. For update events, the attribute (or set of attributes) target of the modification can also be specified. The condition (WHEN clause) is expressed as an arbitrary SQL predicate, potentially involving complex queries. The action can be any sequence of database operations, even operations not connected in any way to the triggering event. In particular, the action can include any SQL data manipulation statement as well as invocations of user-defined functions and procedures. The action could also undo the event (e.g., delete the inserted tuple). Not all kinds of actions are however allowed in all kinds of triggers, in particular, the action of *before* triggers shall not contain an SQL statement that changes data or invokes a routine that possibly modifies data.

The SQL:1999 trigger statement gives the user the possibility of specifying several different options. In particular:

- The trigger can be executed before or after the triggering operation.
- The possibility is given of specifying that the trigger be executed: (*i*) once for each modified tuple (*row-level trigger*), or (*ii*) once for all the tuples that are changed in a database operation (*statement-level trigger*).
- The trigger condition and action can refer to both old and new values of tuples that were inserted, deleted or updated by the operation that triggered the rule.

Each trigger can be executed FOR EACH ROW, that is, once for each tuple manipulated by the triggering operation, or FOR EACH STATEMENT, that is, once for each SQL statement that triggers the rule. In a statement-level trigger, the trigger condition is evaluated once on all the tuples affected by the triggering operation, and if it holds, the trigger action is executed in a set-oriented way. The trigger condition and action can refer to both old and new values of tuples that were inserted, deleted or updated by the operation that triggered the rule. These tuples can be referred to by associating names (*transition variables*) with them. In a statement-level trigger, old and new tuples cannot be referred directly. Rather, we can refer to the set of old tuples (deleted tuples or old versions of updated tuples) and the set of new tuples (inserted tuples or new versions of updated tuples) as two relations. Those *transition tables* can also be referred in row-level triggers, for instance to apply aggregations over the whole set of tuples manipulated by the triggering operation. However, no transition table declaration is allowed in *before* triggers, which, therefore, can only refer to individual transition variables.

In SQL:1999 there is no notion of priority, rather, the set of triggers is ordered based on the trigger creation time (old triggers are activated before new triggers). Triggers are all executed in the context of the same transaction to which the triggering operation belongs.

	HiPAC	Ode	EXACT	NAOS	TriGS	SAMOS	Chimera
Reference	[DBC96]	[GJ96]	[DJ97]	[CC95]	[KR98]	[GD92]	[CFPT96]
o-o data model	OODAPLEX	new	new	O_2	GemStone	ObjectStore	new
primitive events	messages db ops temporal external	messages db ops temporal	messages	messages db ops temporal user-def	messages temporal	messages db ops temporal user-def	messages db ops migrations
event composition	YES	YES	YES	YES	YES	YES	YES(1)
parametric events	YES	NO	YES	NO(3)	NO(2)	YES	NO
parameter passing	$E \rightarrow C, A$ $C \rightarrow A$	$E \rightarrow A$	$E \rightarrow C, A$	$E \rightarrow C, A$ $C \rightarrow A$	$E \rightarrow C, A$(2)	NO	$C \rightarrow A$
conditions on past states	NO	YES	NO	YES	NO	NO	YES
net effect	NO	NO	NO	YES	NO	NO	optional
evaluation mode	immediate deferred decoupled	immediate deferred decoupled(4)	immediate deferred	immediate deferred(5)	user-specified	immediate deferred decoupled	immediate deferred(5)
untargeted/ targeted rules	untargeted	targeted	untargeted(6)	untargeted	both	both	both
overriding	NO	YES(7)	NO	NO	YES	NO	YES

Legenda:
(1) In [CFPT96] only disjunction of events is considered; an extension of Chimera with other kinds of event composition is described in [MPC96].
(2) The only parameter is the object receiver of the event (on which the rule is being executed).
(3) Delta tables are used for referring to objects affected by events.
(4) 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.
(5) This coupling mode is between Event and Condition; coupling mode between Condition and Action is always immediate.
(6) Triggers are defined outside the scope of classes, however they are indexed on classes for efficient detection of relevant events.
(7) Rules can be overridden provided that they are not activated in the superclass.

Table 1.2: Comparison among active object-oriented data models

	SQL:1999
Reference	[GP99, Mel99]
data model	object relational
primitive events	db operations
event composition	NO
parametric events	NO
parameter passing	NO
conditions on past states	YES
net effect	NO
evaluation mode	immediate
untargeted/targeted rules	targeted
overriding	NO

Table 1.3: Triggers in SQL:1999

Chapter 2

Formalizing the ODMG Object Model

In this chapter we propose a formal definition of the object model of ODMG 3.0. The proposed framework is the formal basis on which the temporal and the active model will be built.

ODMG provides a relevant basis for investigating open research issues concerning object-oriented databases and for extending them with new features. The main drawback of ODMG, and in particular of its object model, is the lack of a formal basis. Such a lack makes the model difficult to understand and to extend. The ODMG object model is semantically a very rich data model, thus, its formalization entails several interesting issues that we address in the chapter, such as the differences between classes and interfaces, the characteristics of objects, the formalization of the properties of the inheritance relationships, and so on.

This chapter is organized as follows. Section 2.1 summarizes the main features of the ODMG standard object model. Section 2.2 deals with the controversial issue of collection management in ODMG. Sections 2.3 and 2.4 formally introduce types and values of the model, respectively. Section 2.5 deals with interfaces and classes, whereas Section 2.6 deals with objects. Section 2.7 is devoted to inheritance and subtyping. Section 2.8 formalizes the notions of ODMG database schema and database. In Section 2.9 an illustrative example is presented. Section 2.10 compares the ODMG object model with C++ and Java object models. Finally, Section 2.11 concludes the chapter.

2.1 An Overview of the ODMG Object Model

In what follows we summarize the main features of the ODMG 3.0 object model. A detailed description of the object definition language (ODL) and the object query language (OQL) can be found in Section 5.2 and 4.1, respectively.

2.1.1 Types

The basic modeling concepts in the ODMG object model are the notions of *object* and *literal*.[1] The main difference between an object and a literal is that an object has a unique identifier (*oid*), whereas a literal has no identifier. Literals are described as being constant or *immutable* since their values cannot change. By contrast, objects are described as being *mutable*. Changing the values of the attributes of an object, or the relationships in which the object is involved, does not change the identity of the object. Objects and literals can be categorized according to their *types*.

Object Types. Object types can be defined through *interface* or *class* declarations, whose characteristics will be presented in Section 2.1.2. In ODMG, class and interface identifiers are thus object types. Object types can be partitioned into three main groups: atomic object types, collection object types, and structured objects types. Atomic object types are user-defined types (e.g., `Person`, `Employee`) and they can be defined through interface and class declarations. Collection object types are pre-defined and represent set, bag, list, array and dictionary objects. Structured objects types are pre-defined types, namely `Date`, `Interval`, `Time`, `Timestamp`. Besides these three main groups of object types, ODMG provides several pre-defined system types. Object type features will be described in detail in Section 2.3.

Literal Types. Literal types can be partitioned into three groups: atomic literal types, collection literal types, and structured literal types. Atomic literal types are numbers, characters and so on. Collection literal types represent set, bag, list, array, and dictionary literals. Structured literal types have a fixed number of elements, each of which has a name and can contain either a literal value or an object identifier. They represent structures implementing records. Pre-defined structured literal types supported by the ODMG object model are `date`, `interval`, `time`, `timestamp`. Note that they are the corresponding literal version of structured objects types. In addition, the user can define its own structured literal type using the type constructor `struct`. The extent of a literal type is the classical set of values of the corresponding type.

[1]The concept of literal in ODMG 3.0 is similar to the on of *value* in common object-oriented languages. We refer to literal types instead of value types in order to be consistent with ODMG terminology.

Finally, the model provides constructs for assigning names to types. In the ODMG object model names can be assigned to types in three different ways: through the `typedef` declaration, declaring a user-defined structure, or declaring an enumeration type.

The formalization of object, literal, and named types, the difference between collection object types and collection literal types, as well as the difference between structured object types and structured literal types are discussed in Section 2.3.

Example 2.1 Let `Object` be an interface identifier and `Person` be a class identifier, then the following are ODMG object types:

```
Object;
Person;
Set<Person>;
Bag<short>;
```
[2]

whereas the following are ODMG literal types:

```
long;
boolean;
bag<Object>.
```
[3] ◊

2.1.2 Interfaces and Classes

There are two aspects in the definition of an object type: an external *specification* and one or more *implementations*. The external specification of an object type consists of an abstract, implementation-independent description of the properties, operations, and exceptions that are visible to users. The ODMG object model includes two different constructs to define the external specification of an object type: an *interface* definition, which is a specification only defining the abstract behavior of an object type; a *class* definition, which is a specification defining the abstract state and abstract behavior of an object type.

Object type external specifications are characterized by a *state* and a *behavior*. The state is defined by the names and the types of its *properties*. Properties can be either *attributes*

[2] Parametric collection object types, such as `Set<Person>`, and `Bag<short>` cause many problems in the ODMG object model. We will discuss about this in Section 2.2.

[3] Note that the ODMG notation [CBB+99] uses strings with upper case first letter (e.g. `Bag<`τ`>`) to denote collection object types (for bags) and strings with lower case first letter (e.g. `bag<`τ`>`) to denote the corresponding collection literal type.

or *relationships*. The behavior of an object type is defined by the set of *operations* that can be executed on or by the objects of that type.

The object type behavior is specified as a set of *operation signatures* (*method signatures*). Each signature defines the name of an operation, the name and type of each of its arguments, the types of the value returned.[4] Each operation is associated with a single type and its name must be unique within the corresponding type definition. However, operations with the same name can be defined for different types. In such case, we talk of *name overloading*. When an operation is invoked using an overloaded name, a specific operation must be selected for execution. This selection is called *operation dispatching*. The basic idea behind dispatching in ODMG is that at run time the most specific implementation (along the inheritance hierarchy) for the invoked method and the receiver object is selected. An operation may have side effects. Some operations may return no value. The ODMG object model does not include formal specification of the semantics of operations. The semantics is highly implementation dependent.

Extents and *keys* can be optionally associated with an object type defined through a class declaration. The extent of an object type is the set of all instances of the type within a particular database. In the ODMG object model, if an object type has an extent, then it is unique. If an object is an instance of an object type τ, then it necessarily belongs to the extent of τ. If type τ is a subtype of type τ', then the extent of τ is a subset of the extent of τ'. In some cases, instances of an object type, defined through a class declaration, can be uniquely identified by the values they have for some property or set of properties. These identifying properties are called *keys*. The scope of uniqueness for keys is the extent of the type, thus a type must have an extent in order to have a key.

Both class and interface types are object types. The main difference between a class and an interface type is that a class is a type which is directly instantiable, that is, instances of this type may be created, whereas an interface is a type that cannot be directly instantiated. Thus, according to the set-inclusion relationship holding between the set of instances of types related by inheritance, we can state that the set of objects instances of an interface type τ is the union of the set of instances belonging to the classes inheriting from τ.

We recall that interfaces represent the abstract behavior of an object type, whereas classes represent the abstract state and behavior of an object type. However, it is important to remark that, even though interfaces represent the abstract behavior of an object type, attributes and relationships can be defined within an interface declaration. Attribute and relationship declarations can be specified, exactly with the same notation, both in interfaces and classes. When declared within interfaces, however, properties specify abstract behavior, since they are merely shorthand for the *get* and *set* operations. The semantics of such property definitions is the same defined for the OMG [OPR96] accessor and mutator

[4] Moreover each signature defines the names of *exceptions* (error conditions) the operation can raise. In this context we do not consider exceptions.

methods. By contrast, when declared within classes, properties are abstract state, thus, they represent data structures rather than operations. The same construct for property declaration has, thus, different semantics when used within classes or when used within interfaces.

Many object-oriented programming languages, including C++, Java, and Smalltalk, have language constructs called classes. These correspond to implementation classes and are not to be confused with *abstract classes*[5] defined in the ODMG object model. Each language binding defines a mapping between ODMG abstract classes and its language classes. We do not introduce throughout the thesis any language binding, rather we refer the reader to [CBB+99].

Besides the external specification, an object type has one or more implementations. An implementation defines the *internal aspects* of the instances of the object type. The distinction between external specification and implementation is important, since the separation between them is the approach according to which ODMG supports encapsulation. Implementation details are not relevant from a modeling point of view. Thus, the ODMG object model focuses on the external specification, that is, interface and class definitions, disregarding the implementation specification of an object type.

Example 2.2 Let `Object` be an interface identifier. The following is an example of an object type external specification of a class inheriting from `Object`:

```
class Person:Object {
   attribute short age;
   attribute string name;
   attribute enum gender {male, female};
   attribute Address home_address;
   attribute set<Phone_no> phones;
   attribute Department dept;

   relationship Person is_married_to
      inverse Person::is_married_to;

   void get_married(in Person p);};
```

◊

[5]ODMG classes are referred to as abstract in that they specify no method implementation. They should not be confused, however, with abstract classes as supported, for instance, in Java, since ODMG classes can be instantiated.

2.1.3 Objects

An object in ODMG is characterized by a *state* and a *behavior*. The state of an object is defined by the values of its *properties*. Properties can be either *attributes* of the object itself or *relationships* among the object and one or more other objects. Typically the value of an object property can change over time. The behavior of an object is defined by the set of *operations* that can be executed on or by the object. All objects of a given type have the same set of properties and the same set of defined operations.

In addition to the object identifier, an object can be characterized by one or more *names* that are meaningful to the programmer or end user. Object names are used for convenience especially to refer to *root* objects, which provide entry points into the database.

An attribute value is either a literal or an object identifier, whereas relationships are defined between object types. The ODMG object model supports only binary relationships, i.e., relationships between two types, each of which must have instances that can be referenced by object identifiers. Therefore literal types cannot participate in relationships because they do not have object identifiers. Relationships in the ODMG object model are similar to relationships in the entity-relationship data model [Che76]. A binary relationship may be one-to-one, one-to-many, or many-to-many, depending on how many instances of each type participate in the relationship. One-to-many and many-to-many relationships can be represented using collection literal types, such as set, list, and bag.

A relationship is implicitly defined by declaring *traversal paths* that enable applications to use the logical connections between the objects participating in the relationship. For each relationship two traversal paths are declared, one for each direction of traversal of the binary relationship. The following example illustrates traversal path declarations.

Example 2.3 The relationship between a professor and the courses he/she teaches generates two traversal paths, since a professor *teaches* one or more courses, and a course *is taught by* a professor. The `teaches` traversal path is defined in the interface declaration of the `Professor` type. The `is_taught_by` traversal path is defined in the interface declaration of the `Course` type. The fact that the `teaches` and `is_taught_by` traversal paths refer to the same relationship is specified by an `inverse` clause in both traversal path declarations, as shown in what follows:

```
interface Professor { ...
   relationship set<Course> teaches
      inverse Course::is_taught_by;...};

interface Course {...
   relationship Professor is_taught_by
```

```
inverse Professor::teaches;...};
```

The relationship defined by the `teaches` and `is_taught_by` traversal paths is a one-to-many relationship between `Professor` and `Course` objects. This cardinality is shown in the traversal path declarations. A `Professor` instance is associated with a set of `Course` instances via the `teaches` traversal path. A `Course` instance is associated with a single `Professor` instance via the `is_taught_by` traversal path. ◊

The OODBMS is responsible for maintaining referential integrity of relationships. This means that if an object participating in a relationship is deleted, any traversal path leading to that object must also be deleted. Maintaining referential integrity prevents applications from accessing traversal paths that lead to non-existing objects. Object-valued attributes or, in other words, composite objects, offer an alternative to relationships. Object-valued attributes just enable one object to reference another, without expectation of an inverse traversal path or referential integrity.

2.1.4 Inheritance

Like several object models, the ODMG object model includes inheritance-based type-subtype relationships. More precisely, ODMG supports two inheritance relationships: the *ISA* relationship and the *EXTENDS* relationship.

Subtyping through the ISA relationship pertains to the inheritance of behavior only. Thus interfaces may inherit from other interfaces and classes may also inherit from interfaces. By contrast interfaces may not inherit from classes, nor may classes inherit from other classes through the ISA relationship. Because the ODMG object model supports multiple inheritance of object behavior, it could happen that a type inherits operations with the same name from two different interfaces. The model precludes such a possibility by disallowing name overloading along the ISA hierarchy.

In ODMG 3.0 each user-defined object type, declared both through a class and an interface specification, inherits through the ISA relationship from the system interface `Object` which is thus the root of the ISA relationship. Such relationship, between `Object` and the user-defined object types, can be either explicitly declared, as in Example 2.4, or can be implicitly deduced.

In addition to the ISA relationship, that defines the inheritance of behavior between object types, the ODMG object model provides the EXTENDS relationship for the inheritance of state and behavior. The EXTENDS relationship is a single inheritance relationship between two classes, whereby the subordinate class inherits all properties and operations of the class that it extends.

Table 2.1 summarizes the main features of the two inheritance relationships.

Relationship	**Inherited features**	**Multiple inheritance**	**supertype $\leftarrow$ subtype**
ISA	behavior	Yes	interface $\leftarrow$ class interface $\leftarrow$ interface
EXTENDS	state + behavior	No	class $\leftarrow$ class

Table 2.1: Inheritance relationships

The third column of Table 2.1 specifies the allowed relationships among classes and interfaces. In the first row of Table 2.1, the edge from the class (interface) construct to the interface construct denotes that a class (interface) can be related by a ISA relationship to another interface. Similarly, in the second row of Table 2.1, the edge from the class construct to the class construct denotes that a class can be related by an EXTENDS relationship to another class. These are the only allowed relationships. Thus, for instance, no EXTENDS relationship can be defined between a class and an interface.

The following example illustrates the differences between the two inheritance relationships.

Example 2.4 In the following, according to the ODL syntax, the colon denotes the ISA relationship, while `extends` denotes the EXTENDS relationship.

```
interface Object{...};
interface Employee:Object{...};
interface Professor:Employee{...};
class Person:Object{...};
class EmployeePerson extends Person:Employee{...};
```

The inheritance relationships induced by the previous declarations are illustrated in Figure 2.1. In Figure 2.1, interface object type names are placed in simple rectangles, whereas class object type names are placed in rectangles with round corners, the ISA and EXTENDS hierarchies are explicitly distinguished by labels on the edges.

According to the previous declarations, interfaces `Employee` and `Person` inherit the behavior from interface `Object` through the ISA relationship, whereas interface `Professor` inherits the behavior from interface `Employee`. Moreover, class `EmployeePerson` inherits the state and behavior from class `Person`, whereas it inherits the behavior from interface `Employee`.

◊

According to the notation used in the ODMG object model, given an interface i and a class c, if an edge $i \leftarrow c$ (or $i \leftarrow i'$ where i' is an interface) exists in the ISA relationship,

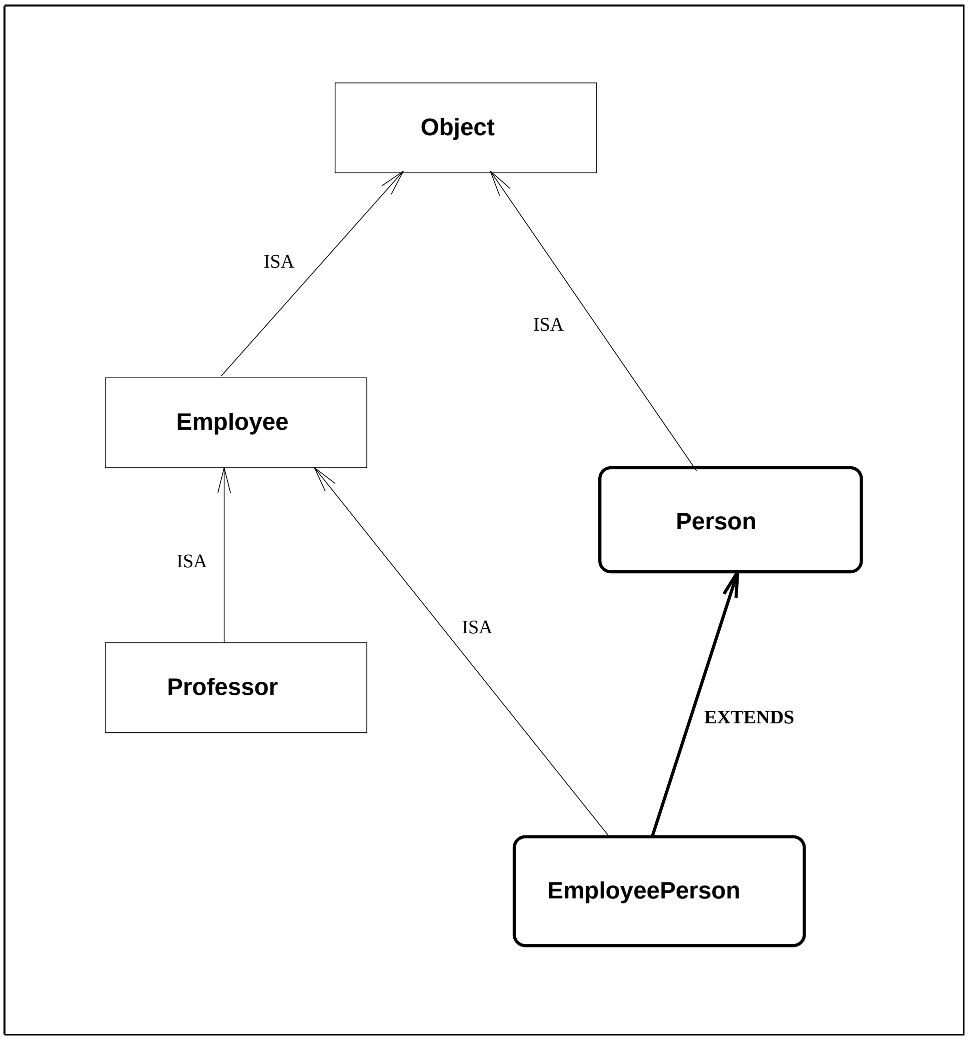

Figure 2.1: Example of inheritance relationships

then i is called *direct superinterface* of c (i'). In addition, given two classes c and c', if an edge $c \leftarrow c'$ exists in the EXTENDS relationship, then c is called *direct superclass* of c'.

If a class type has a direct superinterface, according to the ISA relationship the attributes defined in the interface type are not inherited, since only the behavior is inherited through the ISA relationship. Attributes must thus be redeclared in the class definition. This is in accordance with the fact that interface attributes are intended as accessor method signatures, whereas class attributes are intended as data storage structures. The same remarks apply to relationships.

2.2 A Controversial Issue: Typing Collections

One of the most controversial issues of the ODMG object model is the way in which ODMG manages collections. ODMG 3.0, similarly to previous releases, supports two different kinds of collections:

- collection *object* types, and
- collection *literal* types.

Those collection types are supported via "type generators" denoted by strings with upper case first letter in case of object types and by strings with lower case first letter in case of literal types. Thus, for instance, the type generator for sets is `Set<t>` for collection object types, whereas `set<t>` denotes the corresponding collection literal type, where `t` is the type parameter that can be instantiated with any type of the ODMG object model. In this way all elements of a set are of the same type `t`. However, the corresponding class declaration for sets, which is reported in Figure 2.2, does not define type `Set<t>`, rather type `Set`, without the type parameter. The fact that the type parameter disappears implies heterogeneity in collections which is not the intent of the model. The type parameter `t` appears again in the definition of the attribute `value` which is declared of the literal type `set<t>`. The appearance of this attribute, for each collection object type[6] is one of the new characteristics of the ODMG 3.0 object model. We believe that the introduction of this attribute is probably motivated by the need of solving the controversial issue of managing collections where all the elements of the collections are of the same type. Actually, this attribute does not solve any problem, by contrast it causes more. Indeed, there is still no way to bind the type parameter `t` appearing in `Set<t>` when, for instance, declaring a variable. Since the class identifier is `Set`, there is no way to bind the type parameter `t` appearing in the type of the attribute `value`. Concluding, we can state that though ODMG

[6]We refer to sets, but the same remarks apply to the others collection object types.

```
interface Iterator{...
   Object get_element();
...};

interface Collection{...
   void insert_element(in Object element);
   Iterator create_iterator(in boolean stable);
};

class Set:  Collection {
   attribute set<t> value;
   Set union_with(in Set other);
   Set intersection_with(in Set other);
   Set difference_with(in Set other);
   boolean is_subset_of(in Set other_set);
   boolean is_proper_subset_of(in Set other_set);
   boolean is_superset_of(in Set other_set);
   boolean is_proper_superset_of(in Set other_set);
};
```

Figure 2.2: ODMG collection interfaces

claims to support collection object types such as `Set<t>`, that is, whose elements are all of the same type, it, actually, supports collection object types with no type parameter appearing in their declarations, such as `Set`, that is, heterogeneous collections. Thus, with respect to ODMG collection object types presented in Example 2.1 the correct ones are `Set` for `Set<Person>`, and `Bag` for `Bag<short>`. A possible solution to these problems is parametric polymorphism.

Parametric polymorphism [CW85] provides a general approach to describe data and algorithms, so that their structure is made independent from the type of the manipulated elements. Parametric polymorphism has been extensively investigated by the object-oriented research community and it has been recognized as a very powerful technique for code reusing. For instance, with the introduction of parametric object types in object-oriented languages, the *binary method* problem can be solved [BCC+95]. Binary methods are operations where both arguments are of the same type. Parametric polymorphism is even more relevant in OODBMSs where collections of elements of arbitrary types are frequently manipulated.

The declaration of class `Set` with the support of parametric polymorphism is presented in Figure 2.3. Usually type `Object` is used in the ODMG object model where a type parameter would appear in a type system that supports parametric polymorphism. Note,

```
interface Iterator<T>{...
   T get_element();
...};

interface Collection<T>{...
   void insert_element(in T element);
   Iterator<T> create_iterator();
};

class Set<T>:  Collection<T> {
   Set<T> union_with(in Set<T> other);
   Set<T> intersection_with(in Set<T> other);
   Set<T> difference_with(in Set<T> other);
   boolean is_subset_of(in Set<T> other_set);
   boolean is_proper_subset_of(in Set<T> other_set);
   boolean is_superset_of(in Set<T> other_set);
};
```

Figure 2.3: Parametric collection interfaces

for instance, the definition of the `insert_element` signature in interface `Collection` in Figure 2.2. Since `Object` is the root of the inheritance hierarchy, each object of each object type is of type `Object`, thus, it can be passed as input parameter to the method `insert_element`. In the corresponding signature declaration of method `insert_element` in interface `Collection<T>` in Figure 2.3 the type `Object` is substituted by the type parameter `T`.

Note that, with the support of parametric polymorphism, when declaring a set variable, the type parameter has to be bound to the type of the model, e.g., `Set<Person>`, `Set<Employee>`, and so on.

The introduction of parametric polymorphism in ODMG is not a new issue. The most relevant proposals aiming at adding parametric polymorphism to the ODMG object model are by Alagić [Ala97, Ala99c], who is currently a member of the ODMG group. The first proposal by Alagić [Ala97] is dated 1997, thus it precedes of some years the publication of the release 3.0. However, the ODMG group has decided not to include parametric polymorphism in the last release. We believe that the most likely reason for that is the fact that ODMG standard defines bindings for C++, Smalltalk, and Java. Among them, only C++ supports parametric polymorphism. Recently there have been some proposals for extending Java to support parametric polymorphism [BOSW99, CJ99, SA99]. However, all these proposals are just research projects and nor Java, neither OODBMSs supporting a Java binding, have introduced parametric polymorphism. Thus, we can state that parametric polymorphism is still a research topic and it is not yet mature for being introduced in commercial products. Since the ODMG group has been strongly influenced by OODBMS producers and by the Java language,[7] the group has decided not to include it in the last release.

We believe that parametric object types can offer a good solution to overcome ODMG type system deficiencies, since they can be handled by static type checking. Moreover, it is important that the ODMG standard be extended to include these advanced typing techniques in order to properly manage collections of elements which are fundamental in database systems. Since, however, the ODMG 3.0 object model does not support parametric polymorphism and it seems that there is no intention to include its support in future releases, we have decided not to include it in the thesis. Thus, we consider collection object types without type parameters, such as `Set`, and collection literal types with type parameter, such as `set<t>`. We have dropped from the model object type generators, such as `Set<t>`, since they are actually not supported in ODMG.

Such a modified ODMG object model is as close as possible to the ODMG standard, but it does not have the inconsistencies of the original one. We are aware of the fact that heterogeneous collections can be created, but this is common to all object-oriented languages supporting collections, and not supporting parametric polymorphism, such as Java. Run-time casting can be used to prevent it.

Example 2.5 Let `Person` be the class of Example 2.2. Suppose we want to create a collection `person_coll` of persons. The corresponding statement, using the ODMG method `new`, is:

```
Collection person_coll = new Collection();
```

However, since the type of elements that can be inserted in `person_coll` is `Object` (see the declaration of method `insert_element` in Figure 2.2), we have to control the insertion of elements in `person_coll` through run-time casts. That is, let `o` be an object denoting variable, then the statement:

```
person_coll.insert_element((Person)o);
```

must be used to insert persons in `person_coll`. In this way only objects that can be cast down to class `Person` can be inserted in `person_coll`. ◊

We strongly believe that parametric polymorphism should be integrated in future releases of the ODMG object model, because it allows a proper management of homogeneous object

[7]Note that, currently, the editor in chief of ODMG, R. Cattel, works for Sun Microsystems.

collections guaranteeing type safety, which is a required feature for a model to be adopted as a standard. However, parametric polymorphism should be first introduced in largely used object-oriented languages, such as Java, before it could be considered as a feature of the standard.

2.3 ODMG Types

As we mentioned in Section 2.1, objects and literals can be categorized according to their types. First we give the formal definition of object types, then the definitions of ODMG literal types and named types are given. In the remainder, $\mathcal{T}$ denotes the set of ODMG types. It is formally defined later in this subsection.

Collection object types are a particular kind of system-defined object types, since they are explicitly supported by ODMG, thus they are formalized apart in Definition 2.1.

Definition 2.1 (*Collection Object Types*). The set of ODMG collection object types $\mathcal{COT}$ is defined as the following set $\mathcal{COT} = \{\texttt{Collection}, \texttt{Set}, \texttt{Bag}, \texttt{List}, \texttt{Array}, \texttt{Dictionary}\}$.[8]

□

According to the ODMG object model, collection object types $\mathcal{COT}$ are system-defined. As specified in the definition of the model [CBB+99], `Bag` denotes unordered collections of elements that may contain duplicates. In the scientific literature, such data structures are sometimes referred to as *multisets*. A `Dictionary` object is an ordered sequence of pairs $< key, value >$ with no duplicate keys. `Set`, `List`, and `Array` types have the common meaning. Type `Collection` is defined through an interface declaration, whereas the remaining ones are defined through a class declaration. Collection object types `Set`, `Bag`, `List`, `Array`, `Dictionary` inherit from the type `Collection` through the ISA hierarchy. We do not report here all the definitions for collection object type classes. We refer the interested reader to [CBB+99] for such definitions. Moreover, in [CBB+99] more system-defined object types are listed. We do not consider them in the thesis, since they make the formalization more complex without adding anything interesting to it. Let $\mathcal{CI}$ be a set of class identifiers and $\mathcal{II}$ be a set of interface identifiers, then the following definition formalizes object types.

Definition 2.2 (*Object Types*). The set of ODMG object types $\mathcal{OT}$ is defined as follows.

$$\mathcal{OT} = \mathcal{UOT} \cup \mathcal{COT} \cup \mathcal{SOT} \cup \{\texttt{Object}\}$$

[8]We recall that strings with upper case first letter (e.g. `Set`) denote collection object types (for sets) and strings with lower case first letter (e.g. `set<t>`) denote the corresponding collection literal type.

where:

- $\mathcal{UOT}$ is the set of ODMG *user-defined object types*; it is defined as the union of the set of interface identifiers and the set of class identifiers $\mathcal{II} \cup \mathcal{CI}$;

- $\mathcal{COT}$ is the set of ODMG *system-defined collection object types*, as defined in Definition 2.1;

- $\mathcal{SOT}$ is the set of ODMG *system-defined structured object types*, defined as the following set {`Date`, `Interval`, `Time`, `Timestamp`}.

$\square$

As stated by Definition 2.2, ODMG object types can be partitioned into three sets. The set $\mathcal{UOT}$ denotes user-defined object types. The set $\mathcal{COT}$ denotes the types of objects of type set, bag, etc. ODMG also provides collection literal types (Definition 2.3). The set $\mathcal{SOT}$ denotes the set of structured object types. Finally, type `Object` is an object type. With respect to the ODMG object types, collection object type generators have been eliminated by the object model for the reasons discussed in Section 2.2.

Since an object type can be defined through an interface or through a class definition, the set of ODMG object types can be partitioned into two disjoint sets as follows: $\mathcal{OT}_{int}$, which denotes the set of object types defined through an interface definition, and $\mathcal{OT}_{class}$, which denotes the set of object types defined through a class definition. The fact that a system-defined object type belongs to $\mathcal{OT}_{int}$ or to $\mathcal{OT}_{class}$ is established by the standard itself. For instance, the type `Set`, presented in Figure 2.2, is defined through a class definition, thus `Set` $\in \mathcal{OT}_{class}$. By contrast, user-defined object types, can be defined either through interface or class declarations. Let $\mathcal{STRUCT}_{field}$ be a set of struct field names, then the following definition formalizes ODMG literal types.

Definition 2.3 (*Literal Types*). The set of ODMG literal types $\mathcal{LT}$ is defined as follows.

$$\mathcal{LT} = \mathcal{ALT} \cup \mathcal{CLT} \cup \mathcal{SLT}$$

where:

- $\mathcal{ALT}$ is the set of ODMG *atomic literal types*; it is defined as the following set $\mathcal{ALT}$ = {`long`, `short`, `unsigned long`, `unsigned short`, `float`, `double`, `boolean`, `octet`, `char`, `string`, `enum`};

- $\mathcal{CLT}$ is the set of ODMG *collection literal types*; it is defined as the following set $\mathcal{CLT} = \{\texttt{set<}\tau\texttt{>} \mid \tau \in \mathcal{T}\} \cup \{\texttt{bag<}\tau\texttt{>} \mid \tau \in \mathcal{T}\} \cup \{\texttt{list<}\tau\texttt{>} \mid \tau \in \mathcal{T}\} \cup \{\texttt{array<}\tau\texttt{>} \mid \tau \in \mathcal{T}\} \cup \{\texttt{dictionary<}\tau_1, \tau_2\texttt{>} \mid \tau_1, \tau_2 \in \mathcal{T}\}$;

- $\mathcal{SLT}$ is the set of ODMG *structured literal types*; it is defined as the following set $\mathcal{SLT} = \{$`date`, `interval`, `time`, `timestamp`$\} \cup \{$ `struct`$\{\tau_1\ p_1; \ldots; \tau_n\ p_n\} \mid \tau_i \in \mathcal{T}$ and $p_i \in \mathcal{STRUCT}_{field}$, $i \in [1, n]\}$.

$\square$

An important difference between collection object types and collection literal types is that instances of collection object types have identifiers, whereas instances of collection literal types do not have identifiers. Similarly, ODMG supports both structured object types and pre-defined structured literal types to deal with time. The difference between the two types is the same as in the case of collection types.

In ODMG a name can be associated with a type in three different ways: through a `typedef` declaration, a `struct` declaration, and an `enum` declaration. The following example presents the three kinds of declarations.

Example 2.6 Given the class `Person`, the following are `typedef` declarations.

```
typedef unsigned short ushort;
typedef set<Person> PSet;
typedef struct family{Person mother;
                      Person father;
                      PSet children;};
```

The following are `struct` and `enum` declarations, respectively.

```
struct Address{string college, string room_number};
enum Color{Black,White,Yellow,Blue,Red,Pink,Orange};
```

`ushort`, `PSet`, `family`, `Address`, and `Color` are ODMG named types. $\diamond$

ODMG named types are formalized by the following definition.

Definition 2.4 (*Named Types*). Let $Tname$ be a type name, $\tau, \tau_1, ..., \tau_n \in \mathcal{T}$ be ODMG types, $p_1, ...p_n \in \mathcal{STRUCT}_{field}$ be field names, and $v_1, ..., v_n$ be values, then the set of ODMG type names $\mathcal{NT}$ is defined as follows:

- if $Tname$ is declared as a name for τ through a `typedef` declaration `typedef` τ $Tname$, then $Tname$ is an ODMG *typedef named type* ($Tname \in \mathcal{NT}$);

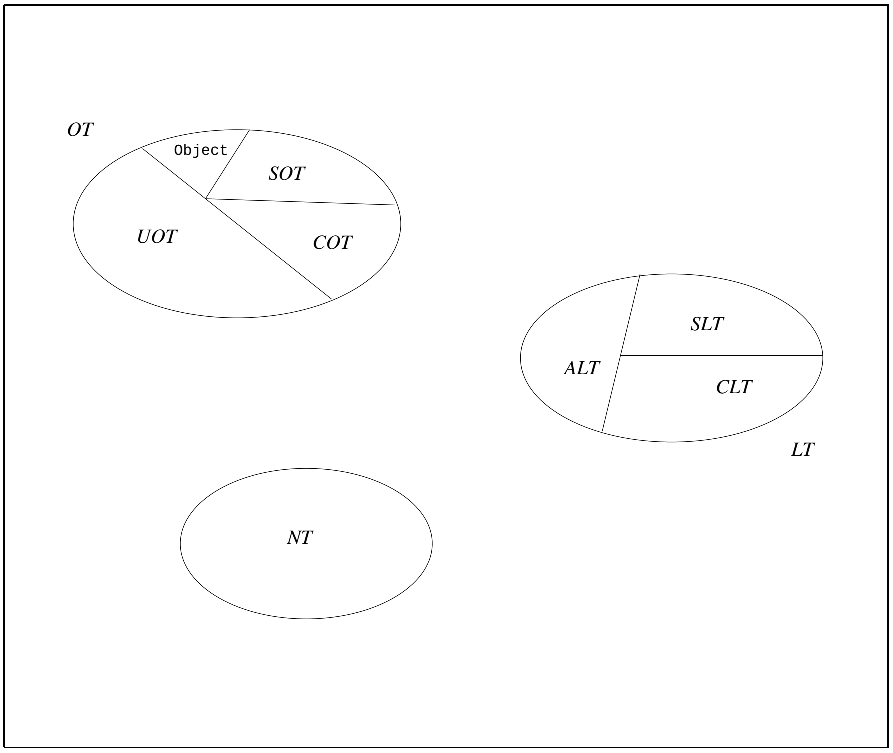

Figure 2.4: ODMG Types (see Table A.1 for acronym meanings)

- if $Tname$ is declared as a name through a `struct` declaration `struct` $Tname\{\tau_1\ p_1; \ldots; \tau_n\ p_n\}$, then $Tname$ is an ODMG *struct named type* ($Tname \in \mathcal{NT}$);

- if $Tname$ is declared as a name through an `enum` declaration `enum` $Tname\{v_1, ...v_n\}$, then $Tname$ is an ODMG *enumeration named type* ($Tname \in \mathcal{NT}$). □

In ODMG named types are introduced as a mean to bind a type name to a type defined according to the model, that is, belonging to $\mathcal{T}$. Note, however, that a name is assigned to a type only to simplify references to that type and that we have a complete equivalence and substitutability of the type with respect to the name. A function $type : \mathcal{NT} \rightarrow \mathcal{T}$ is defined that for each type τ, returns the type obtained from τ by substituting each type name with the type identified by the name. The following example presents some *type* function results.

Example 2.7 Consider the `typedef` declarations of Example 2.6. The application of function *type* to each of the previous types gives the following results:

```
type(ushort) =unsigned short
type(PSet) = set<Person>
type(family) = struct {Person mother;
                        Person father;
                        set<Person> children;}
type(Address) =struct{string college, string room_number}
type(Color) =enum{Black,White,Yellow,Blue,Red,Pink,Orange}
```

$\diamond$

Figure 2.4 presents the set of ODMG types $\mathcal{T}$. $\mathcal{T}$ is defined as the union of the set of ODMG object types $\mathcal{OT}$, the set of ODMG literal types $\mathcal{LT}$ and the set of ODMG named types $\mathcal{NT}$. Therefore:

$$\mathcal{T} = \mathcal{OT} \cup \mathcal{LT} \cup \mathcal{NT}.$$

We also recall that in the ODMG object model `void` stands for "no values". It is used for operations which return no values. In the following we denote with $\mathcal{T}+$ the set $\mathcal{T} \cup \{$`void`$\}$.

2.4 ODMG Values

For each predefined atomic literal type $\tau \in \mathcal{ALT}$, we postulate the existence of a non-empty set of values, denoted as $dom(\tau)$. For instance, the domain of the atomic literal type `boolean` consists of the two classical boolean values `true` and `false`, and the domain of the atomic literal type `short` consists of integer values, such as `31`, `22` and so on.

We consider as legal values for an object type $\tau \in \mathcal{OT}$ all the oids of objects belonging to the extent of τ both as instances or as proper instances. According to the usual terminology, an object is a *proper instance* of an object type τ, if τ is the most specific object type, in the inheritance hierarchy, to which the object belongs. If an object is a proper instance of an object type it is also an *instance* of all the supertypes[9] of τ. We recall that classes are types that are directly instantiable, whereas interfaces are types that cannot be directly instantiated. According to the ISA hierarchy, the set of objects instances of an interface type is the union of the set of objects created for the classes that inherit from the considered interface. Thus, we can say that interfaces have instances, but they do not have proper instances. To define the extension, that is, the set of legal values, for ODMG object types, we introduce function π assigning an extent to each object type. Let $\mathcal{OI}$ be a set of object identifiers.

[9] Subtyping in the ODMG object model will be discussed in Subsection 2.7.2.

Definition 2.5 (*Function* π). Function π: $\mathcal{OT} \rightarrow 2^{\mathcal{OI}}$, for each $\tau \in \mathcal{OT}$ returns the set of the identifiers of objects that belong to the extent of τ as instances. $\square$

Note that, if $\tau \in \mathcal{OT}_{class}$[10] $\subset \mathcal{OT}$ then $\pi(\tau)$ is the union of the set of the proper instances of type τ with the set of instances of τ, whereas if $\tau \in \mathcal{OT}_{int} \subset \mathcal{OT}$, then $\pi(\tau)$ only contains the set of instances of the classes that inherit from τ. In the thesis we will make use frequently of function π^+ which assigns to each object type τ the set of *proper* instances of τ.

Definition 2.6 (*Function* π^+). Function π^+: $\mathcal{OT} \rightarrow 2^{\mathcal{OI}}$, for each $\tau \in \mathcal{OT}$ returns the set of the identifiers of objects that belong to the extent of τ as proper instances. $\square$

Since interface object types cannot be directly instantiated, $\pi^+(\tau) = \emptyset$ for each $\tau \in \mathcal{OT}_{int}$. In addition, the following rule relates the two functions.

Rule 2.1 *Let* $\tau \in \mathcal{OT}$ *be an object type. Then* $\pi^+(\tau) \subseteq \pi(\tau)$.

We denote the set of legal values for type τ as $[\![\tau]\!]$. The following definition states the set of legal values for each type in $\mathcal{T}$.

Definition 2.7 (*Type Legal Values*). $[\![\tau]\!]$ denotes the extension of type $\tau \in \mathcal{T}$.

- $[\![\tau]\!] = dom(\tau) \cup \{nil\}, \forall \tau \in \mathcal{ALT}$;
- $[\![\tau]\!] = \pi(\tau) \cup \{nil\}, \forall \tau \in \mathcal{OT}$;
- $[\![\texttt{set<}\tau\texttt{>}]\!] = \{\texttt{set}\ (v_1, \ldots, v_n) \mid v_i \in [\![\tau]\!],\ i \in [1,n], \text{ and } v_i \neq v_j\ \forall i,j \in [1,n]\ i \neq j\} \cup \{nil\}$;
- $[\![\texttt{bag<}\tau\texttt{>}]\!] = \{\texttt{bag}\ (v_1, \ldots, v_n) \mid v_i \in [\![\tau]\!],\ i \in [1,n]\} \cup \{nil\}$;
- $[\![\texttt{list<}\tau\texttt{>}]\!] = \{\texttt{list}\ (v_1, \ldots, v_n) \mid v_i \in [\![\tau]\!],\ i \in [1,n]\} \cup \{nil\}$;
- $[\![\texttt{array<}\tau\texttt{>}]\!] = \{\texttt{array}\ (v_1, \ldots, v_n) \mid v_i \in [\![\tau]\!],\ i \in [1,n]\} \cup \{nil\}$;
- $[\![\texttt{dictionary<}\tau_1,\tau_2\texttt{>}]\!] = \{\texttt{dictionary}((k_1, v_1), \ldots, (k_n, v_n)) \mid k_i \in [\![\tau_1]\!], v_i \in [\![\tau_2]\!],\ i \in [1,n], \text{ and } k_i \neq k_j\ \forall i,j \in [1,n]\ i \neq j\} \cup \{nil\}$;

[10] We recall that $\mathcal{OT}_{class}$ denotes the set of object types created through a class definition. Similarly $\mathcal{OT}_{int}$ denotes the set of object types created through an interface definition.

- $[\![\, \texttt{struct}\{\tau_1\ p_1;\ldots;\tau_n\ p_n\}\,]\!] = \{\texttt{struct}(p_1 : v_1, \ldots, p_n : v_n) \mid p_i \in \mathcal{STRUCT}_{field}$, $p_i \neq p_j\ \forall i,j \in [1,n]\ i \neq j$, and $v_i \in [\![\, \tau_i \,]\!]\,,\ i \in [1,n]\} \cup \{nil\}$;
- $[\![\, Tname \,]\!] = [\![\, type(Tname) \,]\!]\,,\ \forall Tname \in \mathcal{NT}$.

□

The extensions of pre-defined basic value types are the elements of their corresponding domains, the extensions of object types are their explicit extents, whereas the set of legal values of the collection literal types are defined recursively in terms of the legal values of their component types. The notation used for building collection literal values is the one adopted by ODMG OQL. In order to ensure the correctness of the database, oids must be correctly typed. Therefore, an oid must denote an object whose state meets the type requirements of the definition of the object type to which the object belongs. Finally, note that according to Definition 2.7 $nil \in [\![\, \tau \,]\!]\,,\ \forall \tau \in \mathcal{T}$.

Example 2.8 Let $\texttt{i}_1$ and $\texttt{i}_2 \in \mathcal{OI}$ such that $\texttt{i}_1,\ \texttt{i}_2 \in \pi(\texttt{Person})$.

- $\texttt{10,100} \in [\![\, \texttt{short} \,]\!]$;
- $\texttt{set}(\texttt{i}_1,\texttt{i}_2) \in [\![\, \texttt{set<Person>} \,]\!]$;
- $\texttt{struct}(\texttt{name:Bob,score:40}) \in$ $[\![\, \texttt{struct}\{\texttt{string name;short score}\} \,]\!]$. ◊

In what follows $\mathcal{V}$ denotes the set of legal values for ODMG types. In Figure 2.5 an alternative notation to the one presented in Definition 2.7 is given. We remark that the type deduced according to the typing rules defined in Figure 2.5 is not unique. This is in accordance with the ODMG approach. However, since in ODMG an object belongs to one most specific class, a single most specific type (w.r.t. inheritance) can be identified among the types deduced for a value through the typing rules. This type is, for instance, the one considered at run time for dispatching.

Note that no typing rule is specified for named types in Figure 2.5. Consider for instance the named type `Address` of Example 2.6. No typing rule among the ones specified in Figure 2.5 allows one to deduce v :`Address`, $v \in \mathcal{V}$. By contrast, $[\![\, \texttt{Address} \,]\!] \neq \emptyset$. Adding typing rules for named types causes the problem of deducing more than one single most specific type. Indeed, for each legal value of a named type, the named type and the corresponding structural type are deduced. Thus, we might deduce several types for a value and a most specific type among the deduced ones does not exist. Thus, no typing rule for named types has been considered.

$$\frac{}{nil : \tau} \;\; \forall \tau \in \mathcal{T}$$

$$\frac{v \in dom(\tau)}{v : \tau} \;\; \forall \tau \in \mathcal{ALT}$$

$$\frac{id \in \pi(\tau)}{id : \tau} \;\; \forall \tau \in \mathcal{OT}$$

$$\frac{v_i : \tau \quad i \in [1, n]}{\texttt{set}(v_1, \ldots, v_n) : \texttt{set} <\!\tau\!>} \;\; \forall\ i, j \in [1, n],\ i \neq j\text{:}\ v_i \neq v_j$$

$$\frac{v_i : \tau \quad i \in [1, n]}{\texttt{bag}(v_1, \ldots, v_n) : \texttt{bag} <\!\tau\!>}$$

$$\frac{v_i : \tau \quad i \in [1, n]}{\texttt{list}(v_1, \ldots, v_n) : \texttt{list} <\!\tau\!>}$$

$$\frac{v_i : \tau \quad i \in [1, n]}{\texttt{array}(v_1, \ldots, v_n) : \texttt{array} <\!\tau\!>}$$

$$\frac{k_i : \tau_1 \quad v_i : \tau_2 \quad i \in [1, n]}{\texttt{dictionary}((k_1, v_1), \ldots, (k_n, v_n)) : \texttt{dictionary} <\!\tau_1, \tau_2\!>}$$

Figure 2.5: Typing rules for values

2.5 ODMG Type External Specifications: Interfaces and Classes

An object type can be defined through an interface or a class declaration. Such definition represents the object type *external specification*. The object type external specification is composed by two subparts:

- the object type external specification *body*;
- the object type external specification *characteristics*.

In this section, first the body and then the characteristics of an object type external specification are formally defined.

The body consists of a set of property (attribute and relationship) specifications and a set of operation specifications. Each attribute is characterized by its type and its name. Each relationship is characterized by its type,[11] its name, and its inverse. Let $\mathcal{AN}$, $\mathcal{RN}$, and $\mathcal{MN}$ be a set of attribute names, a set of relationship names, and a set of method names, respectively.[12] Operations are specified through method definitions, each of which has the following form:

```
op_type id(parameter_list)
```

where:

`op_type`$\in \mathcal{T}+$ is the type of the result; it is `void` if no result is returned;

`id` $\in \mathcal{MN}$ is the method name;

`parameter_list` is a list of triples (`param_attribute`, `param_type`, `param_name`) where:

> `param_attribute` $\in$ {`in`, `out`, `inout`} denotes whether the parameter is an input, output, or both input and output parameter;
>
> `param_type` $\in \mathcal{T}$ is the parameter type;
>
> `param_name` is the parameter name.

[11]Suppose that a one-to-one relationship is defined between two object types τ_1 and τ_2, then in τ_1 the type of the relationship is τ_2, whereas, in τ_2 the type of the inverse relationship is τ_1.

[12]To simplify definitions and results, we assume all these sets be mutually disjoint. This assumption is used only at theoretical level, while in ODMG it does not hold. One of the most remarkable characteristics of object-oriented data models is indeed the independence of each object type definition from the others.

The parameter list determines the functionality of the operation defined through the method. The following definition formalizes the previous concepts.

Definition 2.8 (*Object Type External Specification Body*). An external specification body of an object type is a 4-tuple $(i, attr, rel, meth)$, where:

$i \in \mathcal{OT}$ is the type identifier;

$attr$ is a set containing an item for each attribute of the object type. Each item is a pair (a_type, a_name), where:

- $a\_type \in \mathcal{T}$ is the attribute domain type;
- $a\_name \in \mathcal{AN}$ is the attribute name;

rel is a set containing an item for each relationship of the object type. Each item is a triple (r_type, r_name, r_inv), where:

- r_type can be of two different categories: $r\_type \in \mathcal{OT}$ or $r\_type \in \{\texttt{set <}\tau\texttt{>} \mid \tau \in \mathcal{OT}\} \cup \{\texttt{bag <}\tau\texttt{>} \mid \tau \in \mathcal{OT}\} \cup \{\texttt{list <}\tau\texttt{>} \mid \tau \in \mathcal{OT}\}$;
- $r\_name \in \mathcal{RN}$ is the relationship name;
- r_inv is a pair denoting the inverse traversal path of the relationship, that is, $r\_inv = (r\_inv\_type, r\_inv\_name)$ where:
  - $r\_inv\_type \in \mathcal{OT}$ is the inverse relationship domain type;
  - $r\_inv\_name \in \mathcal{RN}$ is the inverse relationship name;

$meth$ is a set containing an item for each method of the object type. Each item is a pair (m_name, m_sign), where:

- $m\_name \in \mathcal{MN}$ is the method name;
- m_sign is the signature of the method, expressed as:

$$p\_name_1 : \tau_1 \times \ldots \times p\_name_j : \tau_j \rightarrow \tau_{j+1} \times p\_name_{j+2} : \tau_{j+2} \times \ldots \times p\_name_n : \tau_n$$

where p_name_i and $\tau_i \in \mathcal{T}$, $i \in [1, j]$, denote names and types of input parameters, $\tau_{j+1} \in \mathcal{T}+$ denotes the type of the method result, and p_name_i and $\tau_i \in \mathcal{T}$, $i \in [j+2, n]$, denote names and types of output parameters.[13]

□

[13]Note that τ_{j+1} is without name since it is the result of the method; if the method returns no value, then τ_{j+1}=`void`. This the only position where `void` can appear. If the method has no input parameter the list at the left side of the arrow can be empty.

Note that the *attr* and *rel* components of the external specification body compose the property specification. As we have seen in Section 2.1, a relationship holds between two types, each of which must have instances that can be referenced by object identifiers. Moreover, to implement one-to-many and many-to-many relationships the relationship domain can be defined as a set, a bag, or a list of objects. This is the reason why in the specification of the component *rel* of an object type, *r_type* can be an object type or a collection literal type, whose inner type is an object type.

ODMG method signature declaration is the same as in CORBA IDL. In our formalization of method signature we have three cases:

- if the parameter attribute is `in`, then the parameter appears on the left side of the signature;
- if the parameter attribute is `out`, then the parameter appears on the right side of the signature;
- if the parameter attribute is `inout`, then the parameter appears both on the left side and on the right side of the signature.

The following example clarifies the formalization of method signatures.

Example 2.9 Consider the following method signature declaration:

```
boolean reg_for_course(in unsigned short course,
                       in unsigned short section);
```

Then the method signature formalization is the following:

$(\texttt{reg\_for\_course}, \texttt{course}: \texttt{unsigned short} \times \texttt{section}: \texttt{unsigned short} \rightarrow \texttt{boolean})$

◊

Inheritance information, extent naming, and key specification are characteristics of object types. The external specification body of an object type is the same for both interface and class types. By contrast, the characteristic specification for interfaces differs from the one for classes. Thus, we provide two distinct definitions, one for interfaces and one for classes.

Definition 2.9 (*Interface External Specification Characteristics*). The external specification characteristics of an interface i, $super_{ISA} \subseteq \mathcal{OT}_{int}$, is the set of interface names of the direct superinterfaces of i. □

Definition 2.10 (*Class External Specification Characteristics*). The external specification characteristics of a class c are defined as a 4-tuple $(super_{ISA}, super_{EXTENDS}, extent, keys)$, where:

$super_{ISA} \subseteq \mathcal{OT}_{int}$ is the set of interface names of the direct superinterfaces of c;

$super_{EXTENDS} \in \mathcal{OT}_{class}$ is the class name of the direct superclass of c;

$extent$ is the extent name;

$keys$ is a set of lists of property names, where each list represents a key of the class c, that is, let n be the number of properties of c, then $keys = \bigcup_{j=1}^{n}\{(p_1, \ldots, p_j) \mid p_i \in \mathcal{AN} \cup \mathcal{RN}$, $i \in [1, j]$, such that $(p_1, \ldots, p_j)$ is a key of $c\}$. □

Note that a key of a class can be composed by an arbitrary number of the properties of the class itself. Thus, $keys$ in the previous definition is defined as a union of sets of property lists of variable length. Each set contains the keys, if any, of that length.

Not all the characteristics have to be specified. In what follows, we denote with the symbol "$\emptyset$" the lack of a particular characteristic. For instance, the class characteristics of a class in which no EXTENDS relationship is specified are denoted by the 4-tuple $(super_{ISA}, \emptyset, extent, keys)$. The same applies for the object type external specification body. For instance, the object type external specification body of a class in which no relationship is specified is denoted as $(i, attr, \emptyset, meth)$. Based on the previous definition, we give the definition of interface and class external specifications.

Definition 2.11 (*Interface External Specification*). An interface specification is a pair $(ext\_spec, charact)$, where:

ext_spec is a type external specification body defined according to Definition 2.8, such that $ext\_spec.i$[14] $\in \mathcal{OT}_{int}$;

$charact$ represents the interface characteristics defined according to Definition 2.9. □

Definition 2.12 (*Class External Specification*). A class specification is a pair $(ext\_spec, charact)$, where:

ext_spec is a type external specification body defined according to Definition 2.8, such that $ext\_spec.i \in \mathcal{OT}_{class}$;

[14]Given an external specification body ext_spec, we use the dot notation to denote each component of ext_spec, for instance, $ext\_spec.i$ denotes the identifier, $ext\_spec.attr$ denotes the set of attributes, and so on.

charact represents the class characteristics defined according to Definition 2.10. □

In the following example the definition in ODL of an ODMG object type and its corresponding external specification, defined according to Definitions 2.11 and 2.12, are illustrated.

Example 2.10 Let `Salary` and `Employee` be two classes defined as follows.

```
class Salary:Object {
   attribute float base;
   attribute float overtime;
   attribute float bonus;};


class Employee:Object
   (extent employees, key id){
   attribute string name;
   attribute short id;
   attribute Salary annual_salary;
   void hire();
   void fire();};
```

Let $(\texttt{Object}, \emptyset, \emptyset, meth')$[15] be the external specification body of type `Object`, then the `Salary` external specification is the pair $(ext\_spec, charact)$ where:

$ext\_spec = (\texttt{Salary}, attr, rel, meth)$ where:

$attr = \{(\texttt{float},\texttt{base}),(\texttt{float},\texttt{overtime}), (\texttt{float},\texttt{bonus})\}$;
$rel = \emptyset$;
$meth = meth'$;

and $charact = (super_{ISA}, super_{EXTENDS}, extent, keys)$ where:

$super_{ISA} = \{\texttt{Object}\}$;
$super_{EXTENDS} = \emptyset$;
$extent = \emptyset$;

[15] We do not report in this context the complete specification of type `Object`, since it is not relevant. We refer the interested reader to [CBB+99].

$keys = \emptyset$.

The `Employee` external specification is the pair $(ext\_spec, charact)$ where:

$ext\_spec =$ (`Employee`, $attr, rel, meth$) where:

$attr = \{$(`string`,`name`),(`short`,`id`), (`Salary`,`annual_salary`)$\}$;
$rel = \emptyset$;
$meth = \{$(`hire`,$\rightarrow$ `void`), (`fire`,$\rightarrow$ `void`)$\} \cup meth'$;

and $charact = (super_{ISA}, super_{EXTENDS}, extent, keys)$ where:

$super_{ISA} = \{$`Object`$\}$;
$super_{EXTENDS} = \emptyset$;
$extent =$ `employees`;
$keys = \{$(`id`)$\}$.

$\diamond$

In what follows we frequently use the object type identifier, $\tau \in \mathcal{OT}$, as an overloaded symbol to denote both the object type external specification and the type identifier. Each component of the object type external specification is accessed through dot notation. Thus $\tau.attr$ denotes the attribute component, $\tau.rel$ denotes the relationship component, $\tau.extent$ denotes the extent component, and so on.[16] For instance, consider the class `Employee` of the previous example, `Employee`$.super_{ISA}$ denotes the set of direct superinterfaces of `Employee`, that is, $\{$`Object`$\}$.

2.6 ODMG Objects

In the ODMG object model each object is characterized by an object identifier. All identifiers of objects in a database are unique. Moreover an object retains the same object identifier for its entire lifetime. Thus, the identifier of an object never changes, even if its attribute and relationship values do. Object identifiers are the means by which an object refers to another object.

In addition to being uniquely identified by an identifier, an object can be uniquely identified by one or more names. Note that object identifiers are automatically generated by the

[16]Note that, since τ is an overloaded symbol, $\tau.i = \tau$.

system and cannot be directly manipulated by users, whereas names are assigned and have to be managed by users.

The lifetime of an object determines how the memory and storage allocated to the object are managed. The lifetime of an object is specified at the time the object is created. Two lifetimes are supported in the ODMG object model: *transient* and *persistent*. An object whose lifetime is transient is a temporary object which is deallocated when the process creating it terminates. By contrast, persistent objects continue to exist after the process that creates them terminates. They are sometimes referred to as *database objects*. According to the previous observations, the model of persistence of ODMG is *orthogonal*. A model of persistence is orthogonal if objects of any type may be persistent or transient [AM95]. In a type system with orthogonal persistence, there may exist both persistent and transient objects of any type. However, orthogonal persistence disappears in language bindings where persistence is a feature of a type and not of an object. Since the aim of our work is to formalize the ODMG object model, we support an orthogonal model of persistence. Thus, in our model persistence information is attached to objects and not to types.

The state of an object is characterized by the set of properties (attributes and relationships) of the most specific object type to which the object belongs and the corresponding values. Finally an object is characterized by a behavior. The behavior of the object is the set of operations associated with the most specific type to which the object belongs. Thus, we associate with the object such type information and from that we can deduce the object behavior. Note that an object must be a proper instance of a class object type, thus the most specific type of an object is always a class object type and cannot be an interface object type. Let $\mathcal{NAME}$ be a set of object names, then the previous concepts are formalized by the following definition.

Definition 2.13 (*Object*). An object o is a 6-tuple (id, N, l, v, r, c), where:

$id \in \mathcal{OI}$ is the oid of o;

$N \subseteq \mathcal{NAME}$ is the set of names of o;

$l \in \{\texttt{transient}, \texttt{persistent}\}$ is the lifetime of o;

v is a tuple, containing the values of each attribute of o. It is a tuple $(a_1 : v_1^a, \ldots, a_n : v_n^a)$, where $a_1, \ldots, a_n \in \mathcal{AN}$ are the names of the attributes of o, and $v_1^a, \ldots, v_n^a \in \mathcal{V}$ are their corresponding values;

r is a tuple, containing the values of each relationship of o. It is a tuple $(r_1 : v_1^r, \ldots, r_m : v_m^r)$, where $r_1, \ldots, r_m \in \mathcal{RN}$ are the names of the relationships of o, and $v_1^r, \ldots, v_m^r \in \mathcal{V}$ are their corresponding values;

$c \in \mathcal{OT}_{class}$ is the most specific object type to which the object belongs. □

Example 2.11 Let `Employee` and `Salary` be the two classes defined as in Example 2.10. Provided that $\mathtt{i}_e$ is an object identifier of type `Employee` and $\mathtt{i}_s$ is an object identifier of type `Salary`, an example of ODMG object belonging to class `Employee` is the 6-tuple $(\mathtt{i}_e, \{\mathtt{john\_smith}\}, \mathtt{persistent}, v, \emptyset, \mathtt{Employee})$, where $v = (\mathtt{name:'John\ Smith', id:123, annual\_salary:i}_s)$. ◊

In the ODMG object model, objects cannot migrate along the inheritance hierarchy and they have a unique most specific class object type to which they belong. Thus, given an object o its last component, c, is the same for the whole object life.

Each object must be a consistent instance of its most specific object type, say class τ. This constraint ensures that the object can be accessed properly as an instance of all the object types superclasses or superinterfaces of τ. We introduce the notion of *consistency* of an object o with respect to an object type τ. Intuitively, an object is a consistent instance of an object type τ if its state matches the object type specification body of τ. That is, each attribute value is a legal value for the corresponding attribute type and each relationship value is a legal value for the corresponding relationship type. In addition, no extra property has to be specified for the object.

Definition 2.14 (*Consistent Instance*). An object $o = (id, N, l, (a_1 : v_1^a, \ldots, a_n : v_n^a), (r_1 : v_1^r, \ldots, r_m : v_m^r), c)$ is a consistent instance of an object type $\bar{\tau}$ whose type external specification body (cf. Definition 2.8) is the 4-uple $(\bar{\tau}, attr, rel, meth)$, if the following conditions hold:

- for each $(\tau, a) \in attr$, then i exists, $i \in [1, n]$, such that $a = a_i$ and $v_i^a \in [\![\, \tau \,]\!]$;
- for each $(\tau, r, r\_inv) \in rel$, then i exists, $i \in [1, m]$, such that $r = r_i$ and $v_i^r \in [\![\, \tau \,]\!]$;
- for each a_i, $i \in [1, n]$, then $(\tau, a) \in attr$ exists such that $a_i = a$;
- for each r_i, $i \in [1, m]$, then $(\tau, r, r\_inv) \in rel$ exists, such that $r_i = r$. □

The following example clarifies the concepts formalized previously.

Example 2.12 Consider the class `Employee` of Example 2.10 and the object

$$\begin{aligned} o = &(\mathtt{i}_e, \{\mathtt{john\_smith}\}, \mathtt{persistent}, \\ &(\mathtt{name:'John\ Smith', id:123, annual\_salary : i_s}), \emptyset, \mathtt{Employee}) \end{aligned}$$

of Example 2.11. Then o is a consistent instance of class `Employee`. That is:

- for each $(\tau, a) \in \{(\mathtt{string}, \mathtt{name}), (\mathtt{short}, \mathtt{id}), (\mathtt{Salary}, \mathtt{annual\_salary})\}$ the corresponding attribute value is defined and its type is correct in the object state (`name:'John Smith'`,`id:123`, `annual_salary:`$\mathtt{i}_s$);
- the set rel of `Employee` is empty, thus the second condition is trivially satisfied;
- for each $a \in \{\mathtt{name}, \mathtt{id}, \mathtt{annual\_salary}\}$ the corresponding attribute definition (τ, a) exists in class `Employee`;
- no relationship has been defined for object o, thus, the last condition is trivially satisfied. ◊

2.7 Inheritance in the ODMG Object Model

ODMG provides two different inheritance hierarchies for object types: the ISA hierarchy and the EXTENDS hierarchy (see Table 2.1). In [CBB+99] the two inheritance relationships and their differences are only informally described. As it is well known in literature, inheritance is a very powerful mechanism, but it requires addressing several issues in order to define a sound data model. In this section we investigate such issues.

First, we formalize important peculiarities of the two ODMG inheritance relationships. Then, we address the controversial issue concerning whether inheritance and subtyping coincide in the ODMG object model.

In what follows we introduce the ISA and the $EXTENDS$ functions and their transitive closures that will be used in this section.

The ISA hierarchy pertains to the inheritance of behavior only. It represents which object types, defined through either a class or an interface declaration, are subclasses or subinterfaces of other interface object types.

Definition 2.15 (*Function ISA*). Let $\mathcal{OT}_{int}$ be the set of object types defined through an interface definition, then the ISA hierarchy is represented by function:

$$ISA : \mathcal{OT} \rightarrow 2^{\mathcal{OT}_{int}}$$

such that for each $\tau \in \mathcal{OT}$, $ISA(\tau)$ is the set of direct superinterfaces of τ, that is, $ISA(\tau) = \tau.super_{ISA}$. □

Function $ISA^* : \mathcal{OT} \rightarrow 2^{\mathcal{OT}_{int}}$ is the transitive closure of the ISA relationship.

The EXTENDS hierarchy pertains to the inheritance of both state and behavior. It represents which object types, defined through a class declaration, are subclasses of other class object types.

Definition 2.16 (*Function EXTENDS*). Let $\mathcal{OT}_{class}$ be the set of object types defined through a class definition, then the EXTENDS hierarchy is represented by function:

$$EXTENDS : \mathcal{OT}_{class} \rightarrow \mathcal{OT}_{class}$$

such that for each $\tau \in \mathcal{OT}_{class}$, $EXTENDS(\tau)$ is the direct superclass of τ,[17]that is, $EXTENDS(\tau) = \tau.super_{EXTENDS}$ □

Function $EXTENDS^* : \mathcal{OT}_{class} \rightarrow 2^{\mathcal{OT}_{class}}$ is the transitive closure of the EXTENDS relationship. $EXTENDS$ is a partial function. Let c be a class, if no superclass is defined for c, then $EXTENDS(c)$ is undefined, denoted by $EXTENDS(c) = \perp$.

Example 2.13 Let `Employee`, `Professor`, `Person` and `EmployeePerson` be the object types defined in Example 2.4, then:

$$ISA(\texttt{Object}) = \emptyset$$

$$ISA(\texttt{Person}) = ISA(\texttt{Employee}) = \{\texttt{Object}\}$$

$$ISA(\texttt{EmployeePerson}) = ISA(\texttt{Professor}) = \{\texttt{Employee}\}$$

$$EXTENDS(\texttt{Person}) = \perp$$

$$EXTENDS(\texttt{EmployeePerson}) = \texttt{Person}$$

Since types `Object`, `Employee`, and `Person` defined in Example 2.4 have at most one ancestor, the transitive closures, ISA^* and $EXTENDS^*$, are equal to the functions themselves. For instance, $ISA^*(\texttt{Person}) = ISA(\texttt{Person}) = \{\texttt{Object}\}$. For type `EmployeePerson` $EXTENDS^*$ is equal to $EXTENDS$, whereas $ISA^*(\texttt{Professor}) = ISA^*(\texttt{EmployeePerson}) = \{\texttt{Object}, \texttt{Employee}\}$. ◊

Inheritance relationships are system-defined or established by the user when defining classes and interfaces. For instance, the type `Set` of Figure 2.3 is system-defined through inheritance from the type `Collection`. Generally speaking each collection object type is defined

[17]Note that since the EXTENDS relationship is a single inheritance relationship the value returned by function $EXTENDS$ is a single type and not a set of types as in the case of function ISA.

through inheritance, according to the ISA hierarchy, from type `Collection`. That is, `Collection` $\in ISA(\tau)$ for each $\tau \in \mathcal{COT}$, $\tau \neq$ `Collection`. By contrast, the inheritance relationships of Example 2.13 are user-defined inheritance relationships.

According to the ISA and EXTENDS inheritance relationships, an object type inherits all properties and methods of the object types from which it has been derived. Thus the set of attributes of an object type, that is, the *attr* component of Definition 2.8, can be partitioned into two disjoint sets:

- the set of *proper* attributes, that have been defined specifically for the object type;
- the set of *inherited* attributes, that the object type inherits from other object types.

Given an object type τ, whose external specification is the 4-tuple $(i, attr, rel, meth)$, functions $\mathcal{A}$, $\mathcal{A}^+$, $\mathcal{A}_{names}$, and $\mathcal{A}^+_{names}$ return the *attr* component of τ, the subset of *attr* representing the proper attributes, the names of the attributes of τ, and the names of the proper attributes of τ, respectively.[18] Such functions are used in what follows to define rules on the ISA and the EXTENDS hierarchies. Analogous notations are defined for relationships and methods.

2.7.1 Inheritance Relationship Properties

According to the ODMG object model specification, a set of conditions must be satisfied by the ISA and the EXTENDS relationships. These conditions are formalized in this subsection through definitions of rules on the inheritance hierarchies. In the ODMG object model the ISA hierarchy is a multiple inheritance relationship. In order to avoid name conflicts the model precludes the possibility of inheriting the same property or method through two different interfaces. In [CBB+99] the fact that name conflict can result from the EXTENDS relationship is not addressed. For instance, in the example reported in Figure 2.6 method `m` is inherited in `sub_c` through the ISA relationship from `i` and through the EXTENDS relationship from `c`.

In precluding name conflicts along the inheritance hierarchies we have to take into account the well-known *diamond* problem [Ber88]. Consider the inheritance hierarchy presented in Figure 2.7. In Figure 2.7 interface i_1 and interface i_2 have a common ancestor, $\bar{i}$. Suppose moreover that a method `m` is defined in $\bar{i}$ and is inherited by i_1 and i_2 through the ISA relationship, then i inherits method `m` from interface $\bar{i}$. This is not an example of name conflict, since the *source* of method `m` in i, that is, the most generic object type along the inheritance hierarchy in which `m` is defined, is unique and it is equal to $\bar{i}$. Thus, in this

[18] Note that usually in object-oriented data models only proper attributes are declared in a class, whereas the inherited ones are not redeclared.

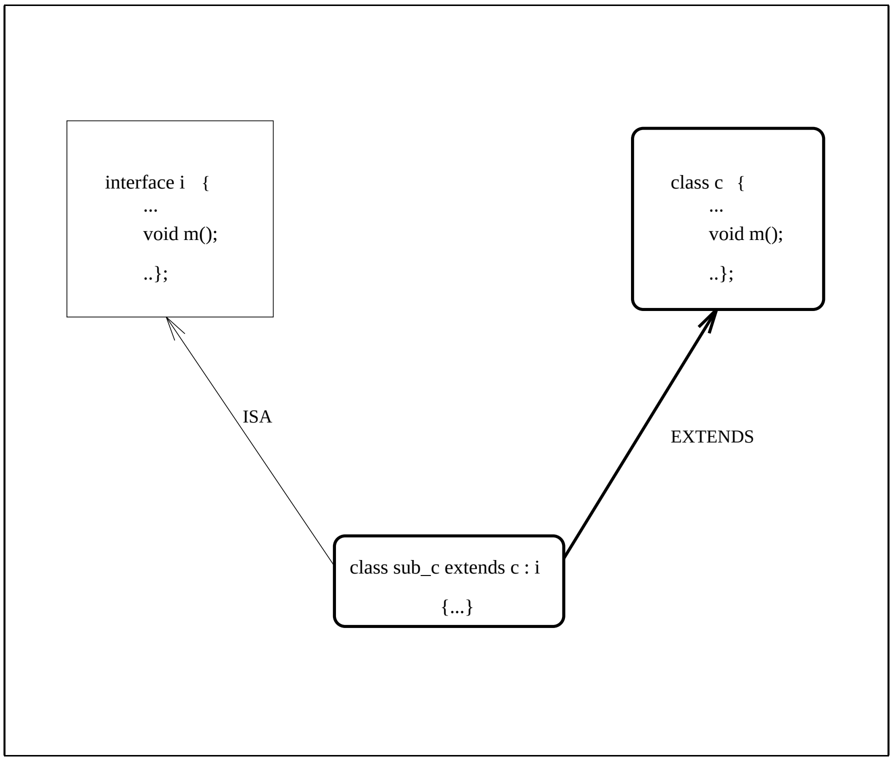

Figure 2.6: Example of overloaded method name

case no name conflict problem exists. In the following definition we formalize the concept of source of a property or of a method with respect to an object type.

Definition 2.17 (*Source*). Let $\tau \in \mathcal{OT}$ be an object type and $n \in \mathcal{AN} \cup \mathcal{RN} \cup \mathcal{MN}$ a property or method name. We denote with $source(n, \tau)$ the set of most generic object types, ancestors of τ, where n is defined, that is:

$$\begin{aligned} source(n,\tau) = \{\bar{\tau} \mid & \bar{\tau} \in ISA^*(\tau) \cup EXTENDS^*(\tau) \cup \{\tau\} \text{ and} \\ & n \in \mathcal{A}_{names}(\bar{\tau}) \cup \mathcal{R}_{names}(\bar{\tau}) \cup \mathcal{M}_{names}(\bar{\tau}) \text{ and} \\ & \nexists\, \tau' \in ISA^*(\bar{\tau}) \cup EXTENDS^*(\bar{\tau}) \text{ such that} \\ & n \in \mathcal{A}_{names}(\tau') \cup \mathcal{R}_{names}(\tau') \cup \mathcal{M}_{names}(\tau')\} \end{aligned}$$

□

Note that in ODMG, property and method definitions can appear only once along each path of the inheritance hierarchies.[19] The following rule precludes name overloading taking into account both inheritance relationships.

Rule 2.2 *Let $\tau \in \mathcal{OT}$ be an object type. Then for each $n \in \mathcal{A}_{names}(\tau) \cup \mathcal{R}_{names}(\tau) \cup \mathcal{M}_{names}(\tau)$, $source(n, \tau)$ is a singleton.*[20]

The inheritance hierarchy presented in Figure 2.6 does not satisfy Rule 2.2, since $source(\texttt{m}, \texttt{sub\_class}) = \{\texttt{i}, \texttt{c}\}$. By contrast, the inheritance hierarchy in Figure 2.7 does, since $source(\texttt{m}, i) = source(\texttt{m}, i_1) = source(\texttt{m}, i_2) = source(\texttt{m}, \bar{i}) = \bar{i}$.

The previous rule precludes name conflicts but it is not enough to ensure the absence of conflicts. As far as attribute and relationship definitions are concerned no problem arises. The situation is different for methods. Even though name overloading is not allowed, conflicts can arise in method dispatching, because of conflicts among method implementations. Consider the ISA inheritance hierarchy among interfaces presented in Figure 2.7. Suppose, for instance, that a method `m` is defined in interface $\bar{i}$ and an implementation is provided for it in $\bar{i}$. In addition, suppose that i_1 and i_2 inherit method `m` but they override its implementation providing two different implementations. Note that this is allowed in the ODMG object model, since method signature overriding is forbidden, whereas method implementation overriding is allowed. An object type i that inherits method `m` satisfies Rule 2.2, since $source(\texttt{m}, i) = \{\bar{i}\}$ which is a singleton, but, at run time, two possible implementations can be chosen: the one in i_1 and the one in i_2.

[19] This property is guaranteed by Rules 2.4 and 2.5, stated in what follows.

[20] We recall that a singleton is a set with only one element.

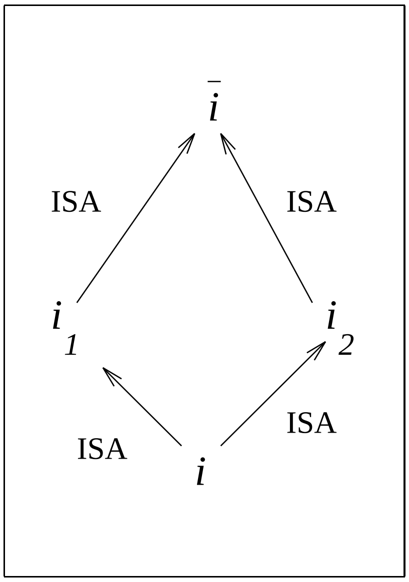

Figure 2.7: Example of the *diamond* problem

With respect to such a problem, the approach taken by ODMG, in accordance to the Java one, is to detect and forbid at compile time the conflicts among method implementations that could generate ambiguities in run time method dispatching. The basic idea behind dispatching is indeed that at run time the most specific implementation (along the inheritance hierarchy) for the invoked method and the receiver object is selected. In the example of Figure 2.7 , however, an invocation `o.m` where `o` is an instance of class i, cannot be unambiguously dispatched. Two implementations for `m` are specified in i_1 and i_2, and no one is more specific than the other. Thus, these situations are avoided by generating a compile error when the schema is compiled.

According to this approach, a message is sent to a receiver object and the dynamic type of the method receiver determines the method implementation 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. Single dispatching is adequate for several kinds of messages. However, it is not adequate for all kinds of methods; an example is given by binary methods [BCC$^+$95]. To address such problem, multiple dispatching techniques have been proposed [Cha92]. *Multi-methods* [Cas97] are a generalization of selfish methods. A multi-method is a collection of method bodies associated with a single message name. The selection of the method body to execute depends on the dynamic types of one or more method parameters (rather than just on the type of the receiver as in the selfish methods). We believe that the possibility of extending ODMG with multiple dispatch could be an interesting topic, which we leave for future work.

In ODMG, the fact that the ISA and the EXTENDS relationships have different semantics has several implications on inheritance. For example, as it is specified in Table 2.1, interface object types cannot be related by the EXTENDS relationship, since interface object types represent abstract behavior and not abstract state, whereas class object types represent abstract state and behavior. Another interesting property of the inheritance relationships is that if a class object type inherits a set of properties, which represents the state of the type, through the ISA hierarchy from an interface, these properties must be redefined, exactly with the same names and types, in the class. Such a redefinition is required because property definitions represent accessor methods in interface external specifications, whereas they represent data structures implementing the object state in class external specifications. This property is formalized by the following rule.

Rule 2.3 *Let $\tau \in \mathcal{OT}_{class}$ be an object type, defined through a class definition. Then for each $\tau' \in ISA(\tau)$ the following conditions must hold:*

- $\mathcal{A}(\tau') \subseteq \mathcal{A}^{+}(\tau)$;
- $\mathcal{R}(\tau') \subseteq \mathcal{R}^{+}(\tau)$.

Intuitively this rule states that each property, attribute or relationship, that the class object type τ inherits from an interface object type τ' through the ISA hierarchy, that is, each element in $\mathcal{A}(\tau') \cup \mathcal{R}(\tau')$, has to be redeclared in τ. That is, such property has to be included in the set of proper properties of τ, that is, $\mathcal{A}^{+}(\tau) \cup \mathcal{R}^{+}(\tau)$. The set $\mathcal{A}(\tau')$ stands for abstract behavior in τ', whereas it stands for abstract state in τ. Note that, according to the previous rule, attributes and relationships must be redefined, whereas methods are simply inherited since, both in τ and in τ', they represent abstract behavior.

2.7.2 Inheritance Relationships and Subtyping

Besides the inheritance relationships, the *subtyping* relationship is often present in classical object-oriented models. From an intuitive point of view, the subtyping relationship is a set inclusion relationship between sets of values. Usually the subtype relationship satisfies two properties [AC96]. The first is *extent inclusion*, that is, the property ensuring that the extent of a subtype is included in the extent of its supertypes. The second is *substitutability* (or *subsumption*), that is, the property that each instance of a type can be used whenever an instance of one of its supertypes is expected.

One of the main features of type systems of classical object-oriented languages is the strict correlation between inheritance and subtyping. The identification between the two relationships confers much flexibility in the use of subsumption: an object of a subclass can always be used in place of an object of a superclass by virtue of subtyping.

There are situations, however, in which inheritance and subtyping conflict. Opportunities of code reuse, both by inheritance and by parameterization, can be limited by the coincidence of these relationships. Therefore considerable attention has recently been devoted to separating them [Cas97]. The approach that separates the inheritance and subtyping relationships abandons completely the notion that subclassing implies subtyping and it is well known among the object-oriented community under the name *inheritance-is-not-subtyping* [CHC90]. This approach is motivated by the desire to use self types as arguments of methods. Self types allow one to establish that, for instance, the type of an input parameter of a method must be the same of the object executing the message. In order to preserve substitutability, the redefinition of method signatures must verify the *covariance rule* for result parameters and the *contravariance rule* for the input ones. With the use of self types, even if refinement of method signature is not explicitly allowed, it is implicitly achieved. A method signature defined in a class where self types are used is implicitly refined in all the classes that inherit the method. Although the covariance/contravariance rule guarantees type safety with static type checking, it is well known that naturally, in realistic examples, it is counter intuitive. To compensate the disadvantage of renouncing to such rule, several proposals have been presented in literature [Cas97].

In the ODMG object model, which do not support self types, the specialization of property domains, and method signature redefinition,[21] inheritance, both through the ISA and the EXTENDS relationships, and subtyping collapse in a single relationship.

We now present two rules, one for interface object types and the other for class object types, which formalize the fact that neither the specialization of property domains, nor method signature redefinitions are allowed along the inheritance hierarchy.

Rule 2.4 *Let $\tau \in \mathcal{OT}_{int}$ be an interface object type. Then the following conditions must hold:*

- $\forall\ a \in \mathcal{A}^+_{names}(\tau)\ \nexists\ \tau' \in ISA^*(\tau)$ *such that* $a \in \mathcal{A}^+_{names}(\tau')$*;*
- $\forall\ r \in \mathcal{R}^+_{names}(\tau)\ \nexists\ \tau' \in ISA^*(\tau)$ *such that* $r \in \mathcal{R}^+_{names}(\tau')$*;*
- $\forall\ m \in \mathcal{M}^+_{names}(\tau)\ \nexists\ \tau' \in ISA^*(\tau)$ *such that* $m \in \mathcal{M}^+_{names}(\tau')$*.*

Rule 2.5 *Let $\tau \in \mathcal{OT}_{class}$ be a class object type. Then the following conditions must hold:*

[21] The only exception to such restriction is the case in which a class is defined through the ISA relationship from an interface as it is stated by Rule 2.3. In this case, however, the property is not redefined. For each accessor method a corresponding property is defined, whose type is exactly the return type of the accessor method.

- $\forall\ a \in \mathcal{A}^{+}_{names}(\tau)\ \nexists\ \tau' \in EXTENDS^{*}(\tau)$ *such that* $a \in \mathcal{A}^{+}_{names}(\tau')$;
- $\forall\ r \in \mathcal{R}^{+}_{names}(\tau)\ \nexists\ \tau' \in EXTENDS^{*}(\tau)$ *such that* $r \in \mathcal{R}^{+}_{names}(\tau')$;
- $\forall\ m \in \mathcal{M}^{+}_{names}(\tau)\ \nexists\ \tau' \in ISA^{*}(\tau) \cup EXTENDS^{*}(\tau)$ *such that* $m \in \mathcal{M}^{+}_{names}(\tau')$.

Starting from the ISA and the EXTENDS relationship the subtype relationship $\leq_T$ can be imposed on the types of the model. In what follows, let `l_constr` $\in$ {`collection`, `set`, `bag`, `list`, `array`} be a literal collection type constructor. Finally, let $Tname_1, Tname_2$ be type names.

Definition 2.18 (*Subtypes*). Let $\tau_1,\ \tau_2 \in \mathcal{T}$ be two types. Then, τ_2 is a subtype of τ_1 (denoted as $\tau_2 \leq_T \tau_1$) iff one of the following conditions holds:

1. $\tau_1 = \tau_2$;[22]
2. $\tau_1 \in ISA^{*}(\tau_2)$;
3. $\tau_1 \in EXTENDS^{*}(\tau_2)$;
4. $\tau_1 =$ `l_constr<`τ_1'`>`, $\tau_2 =$ `l_constr<`τ_2'`>` and $\tau_2' \leq_T \tau_1'$;
5. $\tau_1 =$ `dictionary<`τ_1', τ_1''`>`, $\tau_2 =$ `dictionary<`τ_2', τ_2''`>`, $\tau_2' \leq_T \tau_1'$, and $\tau_2'' \leq_T \tau_1''$;
6. $\tau_1 =$`struct`$\{\tau_1'\ p_1,...,\tau_n'\ p_n\}$, $\tau_2 =$`struct`$\{\tau_1''\ p_1,...,\tau_n''\ p_m\}$ such that $m \geq n$ and $\forall i \in [1, n]$: $\tau_i'' \leq_T \tau_i'$;
7. $\tau_1 = Tname_1$ and $\tau_2 \leq_T \mathit{type}(\tau_1)$;
8. $\tau_2 = Tname_2$ and $\mathit{type}(\tau_2) \leq_T \tau_1$;
9. $\tau_1 = Tname_1$ and $\tau_2 = Tname_2$ $\mathit{type}(\tau_2) \leq_T \mathit{type}(\tau_1)$. □

The following rule has to be satisfied by function π (cf. Definition 2.5) and function π^{+} (cf. Definition 2.6) with respect to object types related by subtyping relationship.

Rule 2.6 *Let τ_1 and $\tau_2 \in \mathcal{OT}$ be two object types such that $\tau_2 \leq_T \tau_1$, then $\pi(\tau_2) \subseteq \pi(\tau_1)$ and $\pi^{+}(\tau_2) \subseteq \pi^{+}(\tau_1)$.*

[22] Type equality we refer to is the one in which, two types are equal if they are exactly the same type, e.g. `Person` = `Person`, `short` = `short`, and so on.

Since function π models the set of instances of an object type, this rule formalizes the *extent inclusion* property, which holds in the ODMG object model. Indeed, in ODMG, when an object is created as a proper instance of a class, it is an instance of all the superclasses or superinterfaces of that class.

The following result holds, relating the subtype relationship $\leq_T$ between two types and the set of their legal values.

Proposition 2.1 *Let τ_1 and $\tau_2 \in \mathcal{T}$ be two types such that $\tau_2 \leq_T \tau_1$, then $[\![\, \tau_2 \,]\!] \subseteq [\![\, \tau_1 \,]\!]$.* ○

Proof 2.1 The proof is done for each item of Definition 2.18.

If $\tau_2 \leq_T \tau_1$ according to the first item, then the proposition is trivially proved, since τ_1 and τ_2 are the same type and in particular $[\![\, \tau_2 \,]\!] = [\![\, \tau_1 \,]\!]$.

If $\tau_2 \leq_T \tau_1$ according to the second and the third item, then τ_1 and τ_2 are object types. Since $[\![\, \tau_1 \,]\!] = \pi(\tau_1) \cup \{nil\}$, $[\![\, \tau_2 \,]\!] = \pi(\tau_2) \cup \{nil\}$, and π satisfies Rule 2.6 then the proposition is trivially proved.

If $\tau_2 \leq_T \tau_1$ according to the fourth and the fifth item, then τ_1 and τ_2 are literal types. The proof is done by induction on the literal value constructors. Consider the fourth item of Definition 2.18. Let $\tau_1 = \texttt{set<}\tau_1'\texttt{>}$, and $\tau_2 = \texttt{set<}\tau_2'\texttt{>}$ such that $\tau_2' \leq_T \tau_1'$. We have to prove that $[\![\, \tau_2 \,]\!] \subseteq [\![\, \tau_1 \,]\!]$, that is, $[\![\, \texttt{set<}\tau_2'\texttt{>} \,]\!] \subseteq [\![\, \texttt{set<}\tau_1'\texttt{>} \,]\!]$. According to Definition 2.7:

$[\![\, \texttt{set<}\tau_1'\texttt{>} \,]\!] = \{\texttt{set}\ (v_1, \ldots, v_n) \mid v_i \in [\![\, \tau_1' \,]\!],\ i \in [1,n], \text{ and } v_i \neq v_j\ \forall i,j \in [1,n]\ i \neq j\}$[23]

and

$[\![\, \texttt{set<}\tau_2'\texttt{>} \,]\!] = \{\texttt{set}\ (v_1, \ldots, v_n) \mid v_i \in [\![\, \tau_2' \,]\!],\ i \in [1,n], \text{ and } v_i \neq v_j\ \forall i,j \in [1,n]\ i \neq j\}$.

Since $\tau_2' \leq_T \tau_1'$ we assume for inductive hypothesis that $[\![\, \tau_2' \,]\!] \subseteq [\![\, \tau_1' \,]\!]$. Then, if $v_i \in [\![\, \tau_2' \,]\!]$, then $v_i \in [\![\, \tau_1' \,]\!]$. This implies that:

$[\![\, \texttt{set<}\tau_2'\texttt{>} \,]\!] =$
$\{\texttt{set}\ (v_1, \ldots, v_n) \mid v_i \in [\![\, \tau_2' \,]\!],\ i \in [1,n], \text{ and } v_i \neq v_j\ \forall i,j \in [1,n]\ i \neq j\} \subseteq$
$\{\texttt{set}\ (v_1, \ldots, v_n) \mid v_i \in [\![\, \tau_1' \,]\!],\ i \in [1,n], \text{ and } v_i \neq v_j\ \forall i,j \in [1,n]\ i \neq j\} =$
$[\![\, \texttt{set<}\tau_1'\texttt{>} \,]\!]$.

We omit the proof for the other collection literal type constructor, that is, $\texttt{l\_constr} \in \{\texttt{collection}, \texttt{bag}, \texttt{list}, \texttt{array}\}$, and for dictionary types since it is very similar to the one for sets.

If $\tau_2 \leq_T \tau_1$ according to the remaining items, then τ_1 and/or τ_2 are named types we can refer to one of the previous cases using the *type* function. ○

[23]We omit *nil* which appears in each set of values.

The converse of Proposition 2.1 does not hold, as it is shown by the following example.

Example 2.14 Consider the following definitions of enumeration named types.

```
enum Color{Black,White,Yellow,Blue,Red,Pink,Orange};
enum BrightColor{White,Yellow,Pink,Orange};
```

$[\![\,\texttt{BrightColor}\,]\!] \subseteq [\![\,\texttt{Color}\,]\!]$ but `BrightColor` is not a subtype of `Color`, that is, `BrightColor` $\leq_T$ `Color` does not hold. $\diamond$

If we restrict the set of types to the set of object types, then we can state that the inclusion of type extents implies subtyping.

Proposition 2.2 *Let τ_1 and $\tau_2 \in \mathcal{OT}$ be two object types such that $[\![\,\tau_2\,]\!] \subseteq [\![\,\tau_1\,]\!]$, then $\tau_2 \leq_T \tau_1$.* $\circ$

Proof 2.2 Since we do not allow objects with multiple most specific classes, if the set of objects corresponding to type τ_2 is a subset of the set of objects of type τ_1, then τ_2 and τ_1 are related by the ISA or EXTENDS relationships. Since in our model inheritance implies subtyping, then they are one subtype of the other. $\circ$

Note that in case of object types, $[\![\,\tau_2\,]\!] \subseteq [\![\,\tau_1\,]\!]$ iff $\tau_2 \leq_T \tau_1$, thus both the implications hold.

2.8 ODMG Schema and Database

A database has the purpose of storing objects, enabling multiple users and applications to share them [CBB$^+$99]. A database is based on a *schema* that is defined in ODL and contains instances of the types defined by its schema. Informally a schema is a definition of a set of user-defined object types including system-defined object types. For instance, the system-defined type `Object` belongs to every database schema.

Note that the set of types appearing in a database schema as property domain types, method parameter types, and so on is always finite. This is the consequence of the fact that a database schema is a finite set of interface and class external specifications related by ISA and EXTENDS relationships. We are now ready to define formally a database schema.

Definition 2.19 (*Database Schema*). Let $S_I = \{I_1, \ldots, I_m\}$ be a set of interface external specifications and let $Id(S_I)$ be the corresponding set of interface identifiers. Moreover, let $S_C = \{C_1, \ldots, C_n\}$[24] be a set of class external specifications and let $Id(S_C)$ be the corresponding set of class identifiers. Finally, let $All\_Types = Id(S_I) \cup Id(S_C) \cup \mathcal{LT} \cup \mathcal{NT}$. Then an ODMG database schema S is defined as the 4-tuple $(S_I, S_C, ISA, EXTENDS)$ such that:

1. $ISA : Id(S_I) \cup Id(S_C) \rightarrow 2^{Id(S_I)}$ is a total function and represents a DAG;[25]
2. $EXTENDS : Id(S_C) \rightarrow Id(S_C)$ is a partial function and represents a tree;[26]
3. S satisfies rules 2.2, 2.3, 2.4, and 2.5;
4. for each T_i, $T_i \in S_I \cup S_C$, $i \in [1, m+n]$, then the following conditions hold:
   - for each $(\bar{\tau}, a) \in T_i.attr$, $\bar{\tau} \in All\_Types$;
   - for each $(\bar{\tau}, r, (\tau', r')) \in T_i.rel$, $\bar{\tau} \in Id(S_I) \cup Id(S_C)$ and $\bar{\tau} = \tau'$, or $\bar{\tau} \in \{\texttt{set<}\tau\texttt{>} \mid \tau \in Id(S_I) \cup Id(S_C)\} \cup \{\texttt{bag<}\tau\texttt{>} \mid \tau \in Id(S_I) \cup Id(S_C)\} \cup \{\texttt{list<}\tau\texttt{>} \mid \tau \in Id(S_I) \cup Id(S_C)\}$, and $\tau = \tau'$;
   - for each $(m, p\_name_1 : \tau_1 \times \ldots \times p\_name_j : \tau_j \rightarrow \tau_{j+1} \times p\_name_{j+2} : \tau_{j+2} \times \ldots \times p\_name_n : \tau_n) \in T_i.rel$, $\tau_i \in \{\texttt{void}\} \cup All\_Types$, $i \in [1, n]$;
   - for each $\bar{\tau} \in T_i.super_{ISA}$, then $\bar{\tau} \in Id(S_I) \cup Id(S_C)$;
   - if $T_i \in S_C$, then $T_i.class \in Id(S_C)$;
5. for each T_i, $T_i \in S_I \cup S_C$, $i \in [1, m+n]$, then for each $(\bar{\tau}, r, (\tau', r')) \in T_i.rel$, exists $T_i' \in S_I \cup S_C$, $i \in [1, m+n]$, such that $T_i'.i = \tau'$ and $(\bar{\tau}', r', (T_i.i, r)) \in T_i'.rel$. □

In the previous definition the first and the second conditions properly define the ISA and the EXTENDS relationships. The third condition ensures that all the rules established throughout the chapter are satisfied. The fourth condition ensures that each type appearing in the schema has been properly defined. Finally, the fifth condition ensures the correctness of relationship traversal path definitions.

An ODMG schema, defined according to Definition 2.19, induces a safe type system, that is, it allows static type checking without the occurrence of run time type errors. Indeed, the ODMG type system is equivalent to an object-oriented type system, where property and

[24]Pre-defined system types belong to S_I or to S_C depending on their definitions.

[25]We recall that DAG is the acronym for *direct acyclic graph*. This graph is such that an edge $n_1 \rightarrow n_2$ belongs to the graph if and only if $n_2 \in ISA(n_1)$.

[26]This tree is such that an arc $n_1 \rightarrow n_2$ belongs to the tree if and only if $n_2 = EXTENDS(n_1)$.

method signature redefinitions are forbidden. Those kinds of object-oriented type system have been formally proved to be type safe [AC96, Cas97].

Intuitively, given a schema, a database over it is a set of objects satisfying some constraints. Example of these constraints are *oid uniqueness* and *referential integrity*. Oid uniqueness establishes that objects in the set are identified by their oids. Referential integrity establishes that if an object o has as value of one of its attributes an object identifier $o'.id$, then o' belongs to the set. For what concerns relationships, which have been introduced in the ODMG object model mainly to leave to the DBMS the task to manage the references among objects, a constraint stronger then referential integrity has to satisfied by the DBMS. Indeed, if an object o is related to an object o' through a relationship r, then o' has to belong to the set, as in the case of referential integrity, and, in addition, has to be related to o through the inverse of r. We call this constraint *coherence of relationships*. In the following definition we introduce the notion of set of objects satisfying the coherence of relationships which will be used in the formal definition of database.

Definition 2.20 (*Set of Objects Satisfying the Coherence of Relationships*).
Let $S = (S_I, S_C, ISA, EXTENDS)$ be an ODMG database schema expressed according to Definition 2.19. A set of objects OBJ satisfies the coherence of relationships in S if, for each object $o \in OBJ$, $o.r = (r_1 : v_1, \ldots, r_n : v_n)$, then, for each $j \in [1, n]$:

- $(\tau_j, r_j, (\tau_j^{inv}, r_j^{inv})) \in o.c.rel$ and $v_j = nil$ or an object $o' \in OBJ$ exists, such that $v_j = o'.id$; in addition, let $o'.r = (r'_1 : v'_1, \ldots, r'_m : v'_m)$, then $k \in [1, m]$ exists such that $r'_k = r_j^{inv}$ and $o.id = v'_k$ or $o.id \in v'_k$; or
- $(coll < \tau_j >, r_j, (\tau_j^{inv}, r_j^{inv})) \in o.c.rel$, $coll \in \{\texttt{set}, \texttt{list}, \texttt{bag}\}$, and $v_j = nil$ or an object $o' \in OBJ$ exists, such that $o'.id \in v_j$; in addition, let $o'.r = (r'_1 : v'_1, \ldots, r'_m : v'_m)$, then $k \in [1, m]$ exists such that $r'_k = r_j^{inv}$ and $o.id = v'_k$ or $o.id \in v'_k$.

□

We are now ready to introduce the notion of ODMG database.

Definition 2.21 (*Database*). Let $S = (S_I, S_C, ISA, EXTENDS)$ be an ODMG database schema expressed according to Definition 2.19. An ODMG database over S is a pair (OBJ, π), where:

1. for each pair of objects o_1 and $o_2 \in OBJ$: if $o_1.id = o_2.id$, then $o_1.N = o_2.N$, $o_1.l = o_2.l$, $o_1.v = o_2.v$, $o_1.r = o_2.r$, and $o_1.c = o_2.c$;
2. OBJ satisfies the coherence of relationships expressed in S (cf. Definition 2.20);

3. each object $o \in OBJ$ is a consistent instance of $o.c$ according to Definition 2.14 and $o.c \in Id(S_C)$;

4. let $(p_1, \ldots, p_n)$ be defined as a key for class c, then for each pair of objects o_1 and $o_2 \in OBJ$ such that $o_1, o_2 \in [\![c]\!]$, let $(v_1, \ldots, v_n)$ and $(v'_1, \ldots, v'_n)$ be the tuples representing the values of $p_1, \ldots, p_n$ in o_1 and o_2 respectively, then $(v_1, \ldots, v_n) \neq (v'_1, \ldots, v'_n)$;

5. $\pi : Id(S_I) \cup Id(S_C) \to 2^{OBJ}$ is a function handling object types extensions, such that:

    - for each object $o \in OBJ$, $o.id \in \pi(o.c)$;
    - Rule 2.6 is satisfied by π.

□

The first condition of Definition 2.21 establishes that a database has to satisfy the oid uniqueness constraint. The second condition establishes the coherence of relationships. The third condition constrains each object to be a consistent instance of its most specific class. Such condition implies that the referential integrity constraint is satisfied by the database. Indeed, to be a consistent instance of its most specific class the value of each attribute of the object has to be a legal value for the type of the attribute domain. Thus, in case of an object o whose value of an attribute is an object identifier $o'.id$, then to be a legal value for the type of such attribute o' has to exist, that is, to belong to the set. The same applies for relationships. Actually, the ODMG object model does not establish any integrity constraint on references through attributes, since it is up to the programmer to maintain them. We have decided to accept as legal databases, set of object satisfying the integrity constraint both for attributes and for relationships, since we consider integrity constraint one of the most important semantic constraints in databases. Thus, with this respect, we are more strict then the ODMG object model. The fourth condition establishes key constraints. Finally, the last condition formalizes the properties of function π which represents the set of instances of each object type defined in the schema.

The last notion we introduce in this chapter is the one of database state that will in used in Chapter 5 to model the semantics of triggers. We consider a simple and rather standard definition of database state [AHV95].

Definition 2.22 Let (OBJ, π) be a database. A database state is a pair $S = (\pi^+, \nu)$ where:

- π^+ is the function defined in Definition 2.6;

- for each $o \in OBJ$, $o = (id, N, l, (a_1 : v_1^a, \ldots, a_n : v_n^a), (r_1 : v_1^r, \ldots, r_m : v_m^r), c)$, $\nu(id)$ returns the state of the object, that is, the value of its properties; then $\nu(id) = [a_1 : v_1^a, \ldots, a_n : v_n^a, r_1 : v_1^r, \ldots, r_m : v_m^r]$.[27] □

2.9 An Illustrative Example

In this section we present an example of a database schema where several aspects are involved.

Example 2.15 Let `Salary` and `Employee` be the two classes defined in Example 2.10. Moreover, let `Course` and `Section` be two classes modeling courses and their sections. Then the definitions of the interface `Student` and of the class `Teaching_Assistant`, which inherits from the interface `Student` and from the class `Employee`, are the following.

```
interface Student{
   struct Address{string college, string room_number};
   attribute string stud_name;
   attribute string student_id;
   attribute short year;
   attribute Address dorm_address;
   relationship set<Section> takes
      inverse Section::is_taken_by;
   boolean reg_for_course(in Course c, in Section s);
   void drop_course(in Course c);
   void assign_major(in Department d);
   short transfer(in Section old_s, in Section new_s);};

class Teaching_Assistant extends Employee:Student (extent Teaching_Assistants){
   relationship Section assists
      inverse Section::has_TA;
   attribute string stud_name;
   attribute string student_id;
   attribute short year;
   attribute Address dorm_address;
   relationship set<Section> takes
      inverse Section::is_taken_by;};
```

[27] To denote the values of single attributes we use the following notation: $\nu(o).a_j = v_j^a$, $j \in [1, n]$. The same applies for relationships.

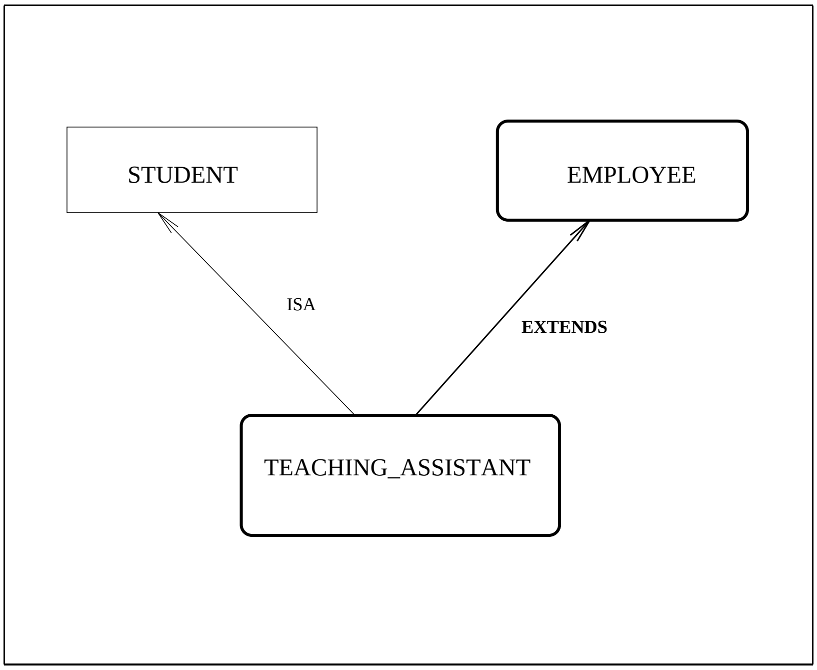

Figure 2.8: Inheritance hierarchy

The inheritance hierarchy corresponding to the previous definition is illustrated in Figure 2.8. Note that in Example 2.10 class `Employee` has been defined as subclass of `Object` through the ISA hierarchy. We do not consider interface `Object` in this example, since it makes the example more complex without involving any new relevant issue. To verify that the presented hierarchy satisfies Rule 2.2, first we have to compute the source of each attribute, relationship and method in class `Teaching_Assistant`. For each $n \in$ {`stud_name`, `student_id`, `year`, `dorm_address`, `takes`, `reg_for_course`, `drop_course`, `assign_major`, `transfer`}, $source(n,$ `Teaching_Assistant`$)$ = {`Student`}, that is, a singleton. For each $n \in$ {`name`, `id`, `annual_salary`, `hire`, `fire`}, $source(n,$ `Teaching_Assistant`$)$ = {`Employee`}, that is, a singleton. For all the remaining properties or methods, the source is the class `Teaching_Assistant` itself, thus the hierarchy satisfies Rule 2.2.

For what concerns Rule 2.3, $ISA(\texttt{Teaching\_Assistant}) = \{\texttt{Student}\}$. We have that:

$$\mathcal{A}^+(\texttt{Teaching\_Assistant}) = \{(\texttt{string}, \texttt{stud\_name}), (\texttt{string}, \texttt{student\_id}), (\texttt{short}, \texttt{year}), (\texttt{Address}, \texttt{dorm\_address})\}$$

$\mathcal{A}(\texttt{Student}) = \{(\texttt{string}, \texttt{stud\_name}), (\texttt{string}, \texttt{student\_id}), (\texttt{short}, \texttt{year}), (\texttt{Address},$

$\texttt{dorm\_address})\}$

Thus $\mathcal{A}(\texttt{Student}) \subseteq \mathcal{A}^{+}(\texttt{Teaching\_Assistant})$. The same can be proved for relationships, thus the hierarchy satisfies Rule 2.3.

Since $\texttt{Teaching\_Assistant} \in \mathcal{OT}_{class}$, Rule 2.4 does not apply, thus we have to verify that the hierarchy satisfies Rule 2.5. More precisely, we have to verify that:

$\forall a \in \mathcal{A}^{+}_{names}(\texttt{Teaching\_Assistant}) \nexists \tau' \in EXTENDS^{*}(\texttt{Teaching\_Assistant}) = \{\texttt{Employee}\}$ such that $a \in \mathcal{A}^{+}_{names}(\tau')$.

$\mathcal{A}^{+}_{names}(\texttt{Teaching\_Assistant}) = \{\texttt{stud\_name}, \texttt{student\_id}, \texttt{year}, \texttt{dorm\_address}\}$ and $\mathcal{A}^{+}_{names}(\texttt{Employee}) = \{\texttt{name}, \texttt{id}, \texttt{annual\_salary})\}$, thus the previous condition holds. Similarly, the remaining conditions of Rule 2.5 can be verified, thus Rule 2.5 is satisfied. $\diamond$

2.10 Comparison between the ODMG Object Model and the Object Models of C++ and Java

In this section we briefly compare the ODMG object model with two languages for which ODMG provides bindings: C++ and Java. Actually ODMG provides an additional language binding, that is, the Smalltalk language binding. We believe, however, that the C++ binding and, in particular, the Java binding are much more significant with respect to the Smalltalk one, since C++ and Java are the most common languages in industrial software production. In addition, the object models of C++ and Java have more heavily influenced the ODMG object model definition.

In comparing the ODMG, C++, and Java data models, we focus on the different meaning they provide for the basic constructs of interface and class. In Table 2.2 the interface, class and abstract class constructs are compared. In such a table we refer to *abstract* state or behavior, whenever the implementation is not provided. In the case of state the implementation is represented by a data structure, by contrast, in the case of methods it is represented by method bodies.

The first column of Table 2.2 reports the summary of all the features of ODMG interfaces, classes and abstract classes as they have been described in Section 2.1. In the second column C++ is described. Note that only classes are supported by C++. Finally, in the third column, the features of the Java interface, class and abstract class constructs are illustrated. Note that only Java supports both interfaces and abstract classes.

Construct	**ODMG**	**C++**	**Java**
interface	abstract behavior attributes + relationships + method signatures no proper instances no implementation	not supported	abstract behavior method signatures no proper instances no implementation
class	abstract state & behavior attributes + relationships + method signatures proper instances no impl.	state & behavior attributes + method signatures & impl. proper instances includes the impl.	state & behavior attributes + method signatures & impl. proper instances includes the impl.
abstract class	not supported	not supported	abstract state & behavior attributes + method signatures no proper instances impl. only of non-abstract methods
inheritance	ISA & EXTENDS no mult. inh. for EXTENDS mult. inh. for ISA (cf. Table 2.1)	a single hierarchy multiple inheritance	extends & implements no mult. inh. for extends mult. inh. for implements extends: class ← class interface ← interface implements: interface ← class
param. types	not supported proposals exits [Ala97, Ala99c]	supported	not supported proposals exits [BOSW99, CJ99, SA99]

Table 2.2: Comparison among ODMG, C++, and Java object models

From Table 2.2 it is very easy to notice that the Java object model is very close to the ODMG one. ODMG interfaces and classes are very similar to the corresponding Java constructs. The feeling arising inside the object-oriented database community is that Java is becoming very relevant also in the database area. In [GMF+99] it is pointed out that the C++ group of ODMG (that is, the group working on the C++ binding) has almost stopped its activity. The name of the group is to be changed from Object Database Management Group to Object Data Management Group, and this can be seen as a witness of the Java "hegemony".

The other relevant fact is that none of the language bindings support relationships. Relationships play a central role in databases, where persistent objects and how they are related have to be stored on a persistent storage. A possible implementation of relationships in

C++ has been proposed [CBB+99]. However, no proposal exists for what concerns the Java language binding. We believe that this open problem should be addressed by following releases of the ODMG object model.

As far as inheritance relationships are concerned, both ODMG and Java support two inheritance relationships, whereas only one inheritance relationship is supported in C++. The ISA relationship of the ODMG object models is similar to the implements relationship of the Java data model. Analogously, the EXTENDS relationship of the ODMG object models is similar to the extends relationship of the Java data model. However, they are not exactly the same. For instance the Java extends relationship can be defined between interfaces, whereas the ODMG EXTENDS relationship can be defined only between classes.

Finally, note that in ODMG the state and the behavior associated with an interface or a class are always *abstract*, that is, no implementation is provided. This is in accordance with the main goal of ODMG, that is, to allow one to define portable database schema and to leave the implementation to different languages.

Moreover, object-oriented programming languages have not being conceived as database programming languages. Therefore, they lack many relevant features which are fundamental for managing huge amounts of data, such as a set-oriented data manipulation language. We believe that, in order to be a standard, ODMG should support such an "ad hoc" data manipulation language. We will deal with this topic in Chapter 5.

2.11 Concluding Remarks

The object-oriented technology has established itself as a very powerful paradigm in several computer science areas. One of the most remarkable proofs of that is the hegemony of Java with respect to the other programming languages. Also in the database area the object-oriented paradigm is recognized to be successful and the new object-relational model of SQL:1999 is a concrete proof of that. The most important commercial products, such as DB2 and Oracle, have added object features to their recent releases and declare themselves SQL:1999 compliant.

However, the pure object-oriented database systems are always a step behind the relational ones in the market. ODMG offers to OODBMS vendors the chance to increase their competition with respect to RDBMS vendors because of the definition of a standard model which can be regarded as the SQL of OODBMSs. ODMG has many interesting ideas, but it lacks a formalization in order to be fully understood and managed.

In this chapter we have provided a comprehensive formalization of the ODMG object model. In our formalization, fundamental notions such as classes, interfaces and objects are precisely defined. All the peculiarities of the model such as the two inheritance relationships, ISA and EXTENDS, among objects, method overloading, are clearly formalized and many interesting properties have been devised. Constraints on the inheritance hierarchies are established in order to capture the restrictions on ODMG schema definition. Finally, we recall that aim of the ODMG group was to define portable schemas. In this chapter the notions of schema and database are also formalized.

Chapter 3

T_ODMG: An ODMG Compliant Temporal Object Model Supporting Multiple Granularity Management

In this chapter we present T_ODMG, a temporal extension of the ODMG object model, which is innovative in many respects. In particular, T_ODMG allows one to manage temporal data expressed with respect to different temporal granularities. Although issues concerning temporal granularities have been investigated in the context of temporal relational database systems [BJW00], no comparable amount of work has been done in the context of object-oriented models. Moreover, the main drawback of the existing proposals is the lack of a formal basis – which we believe is essential to manage the inherent complexity of the object-oriented data model. In this chapter, we formally define the main notions of the temporal data model such as temporal types, their legal values, the impact of temporal features on classes and objects. We address issues related to inheritance, type refinement, and substitutability. Finally, we describe the implementation of the presented model on top of an ODMG compliant DBMS.

The model proposed in this chapter is based on two previous works [BFGM98, MBFG99]. In [BFGM98] we proposed an extension to the ODMG object model to manage time-varying. Such extension did not support multiple granularities. In [MBFG99] we investigated the support of multiple granularities in an object-oriented data model. The thesis work can be considered the mature result of those previous works.

In order to focus on temporal aspects, we consider now a simplified version of the ODMG object model. Indeed, interfaces and their inheritance relationship ISA are not considered in T_ODMG. Thus, object types can be only declared through class definitions in this context. In addition, in T_ODMG, only the structural part of classes is considered, that is, attributes and relationships. This restriction is motivated by the fact that in T_ODMG we would focus on the structural part of objects, which can vary over time, rather than on their behavior. Other restrictions, such as the fact that named types are not considered in T_ODMG, are marginal and will be clarified throughout the chapter. By contrast, in T_ODMG, an extension to the EXTENDS inheritance hierarchy of ODMG is considered. In ODMG property refinement along the EXTENDS hierarchy is not allowed. In order to investigate property refinement in an object-oriented temporal model supporting multiple granularity management, we drop this restriction in T_ODMG and allow the redefinition of a property domain. This issue will be discussed in Section 3.6.2.

The remainder of this chapter is organized as follows. Section 3.1 presents the notion of temporal granularity we refer to in the thesis. Sections from 3.2 to 3.7 are devoted to the formal definition of the extended data model: Section 3.2 introduces our temporal type system, Section 3.3 defines the notion of temporal value, Sections 3.4 and 3.5 present the needed temporal extensions to classes and objects. Issues related to inheritance are discussed in Section 3.6, whereas the notions of schema and database are formalized in Section 3.7. Section 3.8 deals with implementation issues. Finally, Section 3.9 concludes the chapter.

3.1 Preliminaries: Multiple Granularities

Issues concerning temporal granularities have received many attentions from the temporal database area in the last years. In the glossary of time granularity concepts [BDES98] all the main notions concerning temporal granularities are formally defined independently from a specific data model. In the thesis, in order to be as general as possible, we refer to the definitions in [BDES98].

As suggested in the glossary, one of the first notions to be fixed in a temporal context is the *time domain*, that is, the set of primitive temporal entities used to define and interpret time-related concepts. In our model, the time domain is the pair $(\mathbb{N},\leq)$, where $\leq$ is the order on $\mathbb{N}$. Thus, we assume the existence of a relative beginning denoted by symbol "0", but no last element. Moreover, we consider the special variable *now* [CDI$^+$97] to denote the distinguished time instant denoting the current time, and a special symbol ∞ denoting the distinguished time instant $forever$ [JD98]. We recall that our time domain is discrete since every element has an immediate successor, and every element except the first one has an immediate predecessor. Temporal granularities are formally defined as follows.

Definition 3.1 (*Granularity*)[BDES98]. Let $\mathcal{IS}$ be an index set, that is, the set of natural numbers $\mathbb{N}$, and $2^{\mathbb{N}}$ be the power set of the time domain. A *granularity* G is a mapping from $\mathcal{IS}$ to $2^{\mathbb{N}}$ such that both the following conditions hold:

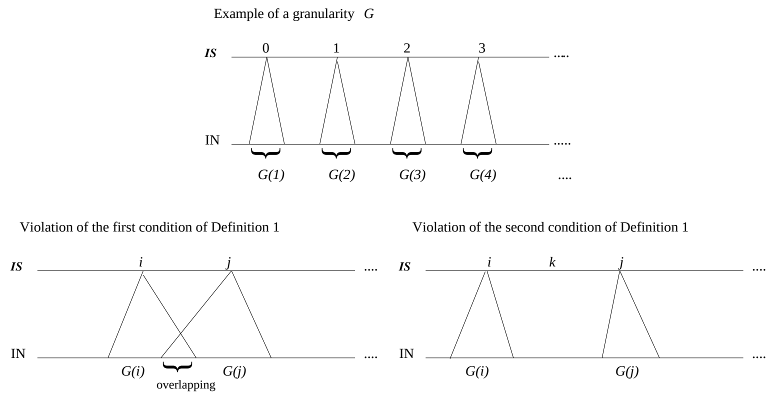

Figure 3.1: Graphical representation of granularity

(1) if $i < j$ and $G(i)$ and $G(j)$ are non-empty, then each element of $G(i)$ is less than all elements of $G(j)$;

(2) if $i < k < j$ and $G(i)$ and $G(j)$ are non-empty, then $G(k)$ is non-empty. □

Actually, in [BDES98] the index set $\mathcal{IS}$ corresponds to the set of integers Z. In order to simplify the notation, we have decided to use the set of natural numbers as index set, that is, we do not consider negative numbers as indexes. Since the set of natural numbers $\mathbb{N}$ is isomorphic to the set of integers Z, such a decision does not have any meaningful impact on Definition 3.1. Thus, the set of natural numbers $\mathbb{N}$ corresponds both to the set of indexes and to the time domain. However, these concepts are completely different. In order to avoid confusion, in what follows, the symbol $\mathbb{N}$ will be used to denote the time domain, whereas the symbol $\mathcal{IS}$ will be used to denote the index set.

Intuitively a granularity defines a countable set of *granules*, such that each granule $G(i)$ corresponds to a natural number. The first condition in Definition 3.1 states that granules in a granularity do not overlap and that their index order is the same as their time domain order. The second condition states that the subset of the index set that maps to non-empty subsets of the time domain is contiguous. Figure 3.1 graphically illustrates those concepts. The set of granularities we refer to throughout the thesis is denoted by $\mathcal{G}$.

The usual collections *days*, *months*, *weeks* and *years* are granularities. For each non-empty granule, we use a "textual representation", termed as *label*. For example, throughout the thesis, *days* are in the form *mm/dd/yyyy*, *months* are in the form *mm/yyyy* and so on.

By contrast, when we refer to a generic granularity G and a granule index $i \in \mathcal{IS}$, we denote l_i^G the label of the granule corresponding to the ith granule of granularity G.

For each granularity $G \in \mathcal{G}$ the set of labels of granules of G is denoted by $\mathcal{LABEL}_G$; in addition, $\mathcal{LABEL}$ denotes the set of all labels, that is, $\mathcal{LABEL} = \bigcup_{G \in \mathcal{G}} \mathcal{LABEL}_G$. Each granule label identifies the granularity, thus, it can belong at most at one of the sets $\mathcal{LABEL}_G$. That is, the set of labels are disjoint, formally, $\bigcap_{G \in \mathcal{G}} \mathcal{LABEL}_G = \emptyset$.

Note that the label is more descriptive than the granule index. Indeed, given a label of a granule we know which is the granularity and the index of the denoted granule. Given a generic label l_i^G, the granule index is i and the granularity is G. When we refer to usual labels, such as the ones for *days* and *months*, we denote with i_l the index corresponding to the granule denoted by label l. Thus, for instance, the label denoting the granule February 1999 with respect to granularity *months* is 02/1999 and the corresponding index is $i_{02/1999}$.

A *finer than* relationship can be defined among temporal granularities, with the following meaning.

Definition 3.2 (*Finer than Relationship*)[BDES98]. A granularity $G \in \mathcal{G}$ is said to be *finer than* a granularity H, denoted $G \preceq H$, if, for each index i, there exists an index j such that $G(i) \subseteq H(j)$. □

For example, *days* is a granularity finer than *months* (*days* $\preceq$ *months*). Note that $G \preceq G$ is always true according to the finer than relationship definition. In what follows the symbol "$\prec$" denotes the anti-reflexive finer than relationship, that is, $G \prec H$ if $G \preceq H$ and $G \neq H$. Throughout the thesis the finer than relationship will be used to state how property domains can be refined along the inheritance hierarchy, to compare values with respect to different granularities, and to navigate through objects.

A set of granularities having the same time domain forms a *granularity lattice* with respect to the $\preceq$ relationship, if for each pair of granularities in the set a least upper bound (*lub*) and a greatest lower bound (*glb*) with respect to $\preceq$ exist. This constraint is easily violated considering very common sets of granularities. For instance, the granularity set $\{weeks, months\}$ does not have neither a lub nor a glb with respect to $\preceq$. Consider now the granularity $G_\emptyset$ such that $G_\emptyset(i) = \emptyset$ for each $i \in \mathcal{IS}$, which we refer to as *empty granularity*. The empty granularity is a lower bound with respect to the $\preceq$ relationship for each set of granularities containing it. From a practical point of view, however, the empty granularity is not meaningful, since it does not supply a unit of measure of the time domain $\mathbb{N}$. For this reason, we do not include the empty granularity into the set of granularities $\mathcal{G}$ we refer to. Note that we do not require the set of granularities we consider to be a lattice.

In our model a *chronon*, that is, the non-decomposable time unit, corresponds to a time instant t belonging to the time domain $\mathbb{N}$. The granularity which establishes a biunivocal correspondence with the time domain is called *chronon granularity*, denoted as G_I. G_I is such that $G_I(i) = \{i\}$ for each $i \in \mathcal{IS}$.

G_I is a bijective function. Indeed, it is injective since for each $i, j \in \mathcal{IS}$, $i \neq j$, $G_I(i) \neq G_I(j)$ and it is surjective since for each $t \in \mathbb{N}$ $i \in \mathcal{IS}$ exists, such that $G_I(i) = \{t\}$, that is, $i = t$. We assume that the chronon granularity G_I belongs to $\mathcal{G}$. Such an assumption is motivated throughout the chapter.

We define now the notion of *temporal interval*.

Definition 3.3 (*Temporal Interval*). Let $G \in \mathcal{G}$ be a granularity and $i, j \in \mathcal{IS}$ be two indexes such that $i \leq j$. Then $[i, j]^G = \{G(k) \mid i \leq k \leq j, k \in \mathcal{IS}\}$ is called *temporal interval*, with respect to granularity G. □

The temporal interval $[i, \infty]^G$ denotes an infinite countable set of granules, that is, $[i, \infty]^G = \{G(k) \mid k \geq i, k \in \mathcal{IS}\}$. The usual operations on sets can be defined for temporal intervals. For instance, given two temporal intervals $[i, j]^G$ and $[i', j']^G$, the intersection between the two is defined as follows: $[i, j]^G \cap [i', j']^G = \{G(k) \mid i \leq k \leq j, i' \leq k \leq j', k \in \mathcal{IS}\}$. Of course a compact notation can be used to denote this set, as it is shown by the following example.

Example 3.1 Let $[i_{01/1999}, i_{07/1999}]^{years}$ and $[i_{04/1999}, i_{12/1999}]^{years}$ be two temporal intervals with respect to granularity *months*, then $[i_{01/1999}, i_{07/1999}]^{years} \cap [i_{04/1999}, i_{12/1999}]^{years} = \{months(k) \mid i_{01/1999} \leq k \leq i_{07/1999}, i_{04/1999} \leq k \leq i_{12/1999}, k \in \mathcal{IS}\} = [i_{04/1999}, i_{07/1999}]^{years}$. ◊

Finally, we formally define the notion of *contiguous* granules and temporal intervals.

Definition 3.4 (*Contiguous Granules*). Let $G \in \mathcal{G}$ be a granularity and $i, j \in \mathcal{IS}$ be two indexes, then the granule corresponding to i in G is *contiguous* to the granule corresponding to j in G if and only if $\nexists\, k$, $i < k < j$, such that $G(k)$ is non-empty. □

Note that a temporal interval is a set of contiguous granules. The notion of contiguous granules can be extended to temporal intervals as follows.

Definition 3.5 (*Contiguous Temporal Intervals*). Let $G \in \mathcal{G}$ be a granularity and $[i, j]^G$ and $[i', j']^G$ be two temporal intervals, then $[i, j]^G$ is *contiguous* to $[i', j']^G$ if and only if the granule corresponding to j is contiguous to the granule corresponding to i' or the granule corresponding to j' is contiguous to the granule corresponding to i. □

We refer to Table A.1 (cf. Appendix A) for the meaning of the symbols most frequently used in the remainder of the chapter.

3.2 T_ODMG Types

In this section, the set of ODMG types, defined in the previous chapter, is extended with the set of temporal types related to different temporal granularities.

We recall that, according to Definitions 2.2, 2.3, and 2.4, the set of ODMG types $\mathcal{T}$ is defined as the union of the set of ODMG object types $\mathcal{OT}$, the set of ODMG literal types $\mathcal{LT}$ and the set of ODMG named types $\mathcal{NT}$. Therefore:

$$\mathcal{T} = \mathcal{OT} \cup \mathcal{LT} \cup \mathcal{NT}.^1$$

In order to focus on temporal aspects, we do not consider interface object types and named types in this context, thus the set of ODMG types we consider for extension is $\mathcal{T}_R = \mathcal{OT}_{class} \cup \mathcal{LT}$. We recall that $\mathcal{OT}_{class}$ denotes the set of object types defined through a class declaration. We extend the set of ODMG types $\mathcal{T}_R$ with a collection of *temporal types*, so that temporal and non-temporal domains can be handled in a uniform way. For each type $\tau \in \mathcal{T}_R$ and granularity $G \in \mathcal{G}$, a corresponding temporal type, $temporal_G(\tau)$, is defined. Intuitively, instances of type $temporal_G(\tau)$ are partial functions from indexes of granules in G to instances of type τ.[2]

Definition 3.6 (*T_ODMG Temporal Types*). Let $\tau \in \mathcal{T}_R$ be an ODMG type and $G \in \mathcal{G}$ be a temporal granularity, then $temporal_G(\tau)$ is the temporal type corresponding to type τ and granularity G. □

Note that according to the previous definition temporal types cannot be nested. Indeed, the inner type τ has to belong to the set of ODMG types $\mathcal{T}_R$.

Example 3.2 Let `Employee` and `Course` be two class object types and *months* and $years \in \mathcal{G}$ be two temporal granularities, then the following are T_ODMG temporal types:

$\texttt{temporal}_{months}\texttt{(Employee);}$
$\texttt{temporal}_{years}\texttt{(Course);}$
$\texttt{temporal}_{years}\texttt{(short);}$
$\texttt{temporal}_{months}\texttt{(list<Course>);}$

By contrast, the nested temporal type $\texttt{temporal}_{months}\texttt{(temporal}_{months}\texttt{(Employee))}$ is not a T_ODMG temporal type, since the inner type $\texttt{temporal}_{months}\texttt{(Employee)}$ is not an ODMG type in $\mathcal{T}_R$. ◊

[1]See Figure 2.4.
[2]We elaborate on that throughout the chapter.

In the following the set of temporal types will be denoted as $\mathcal{TT}$. With the introduction of temporal types the whole set of ODMG types has to be revised. We denote with $\mathcal{T}_{temp}$ the whole set of T_ODMG types. Similarly, we denote with $\mathcal{OT}_{temp}$ the set of T_ODMG object types, and with $\mathcal{LT}_{temp}$ the set of T_ODMG literal types.

The definition of the set of object types of the model is similar to the ODMG one (cf. Definition 2.2). Indeed, the set of type identifiers used to build the set of object types $\mathcal{OT}_{class}$, that is, $\mathcal{CI}$, is still the same and pre-defined types, such as the collection ones, remain unaltered. Thus, $\mathcal{OT}_{temp} = \mathcal{OT}_{class}$. However, class definitions of the new model differ from the previous ones in the type of the properties used to define their object type specifications, since temporal types can be used. This difference will be clarified in Section 3.4 dealing with T_ODMG type external specifications.

The ODMG object model provides structured object types, $\mathcal{SOT}$, and structured literal types, $\mathcal{SLT}$, to deal with time, but they have not to be confused with temporal types. Structured types measure time, whereas, the aim of a temporal type is to store the history of values over time. Structured types are used in the implementation of temporal types on top of an ODMG compliant DBMS (see Section 3.8).

In our model, temporal types can be used in the definition of collection literal types. For instance, a list of temporal values can be defined. The type of such list will be `list<`$temporal_G(\tau)$`>`. In what follows the set of legal values of such kind of types will be clarified. Let `l_constr`$\in$ {`set`,`bag`,`list`,`array`} and $\mathcal{STRUCT}_{field}$ be a set of struct field names, then the set of literal types of T_ODMG is defined as follows.

Definition 3.7 (*T_ODMG Literal Types*). The set of T_ODMG literal types $\mathcal{LT}_{temp}$ is defined as follows.

- Each element of $\mathcal{LT}$ (cf. Definition 2.3) belongs to $\mathcal{LT}_{temp}$ ($\mathcal{LT} \subseteq \mathcal{LT}_{temp}$).
- For each type $\tau \in \mathcal{T}_{temp}$, `l_constr<`$\tau$`>` is a T_ODMG collection literal type (`l_constr<`τ`>` $\in \mathcal{LT}_{temp}$).
- For each type $\tau_1 \in \mathcal{T}_{temp} \setminus \mathcal{TT}$, and $\tau_2 \in \mathcal{T}_{temp}$, `dictionary<`$\tau_1, \tau_2$`>` is a T_ODMG collection literal type (`dictionary<`τ_1, τ_2`>` $\in \mathcal{LT}_{temp}$).
- let p_1,...,$p_n \in \mathcal{STRUCT}_{field}$ be distinct labels and τ_1,...,τ_n be types in $\mathcal{T}_{temp}$, then `struct<`τ_1 p_1`,...,`τ_n p_n`>` is a T_ODMG struct literal type (`struct<`τ_1 p_1`,...,`τ_n p_n`>` $\in \mathcal{LT}_{temp}$). □

We recall that dictionary literal types represent ordered sequence of pairs $< key, value >$ with no duplicate keys. Note that the key component cannot be of temporal type. This is because the key component of a dictionary type represent the search value and temporal keys are not meaningful in this context.

Example 3.3 Let `Employee` and `Course` be two class object types and *months* and $years \in \mathcal{G}$ be two temporal granularities, then the following are T_ODMG literal types:

$\texttt{list<temporal}_{months}\texttt{(Employee)>;}$
$\texttt{set<Course>;}$
$\texttt{temporal}_{years}\texttt{(bag<short>);}$
$\texttt{dictionary<Course,temporal}_{months}\texttt{(short)>;}$

By contrast, the type

$\texttt{list<temporal}_{months}\texttt{(temporal}_{months}\texttt{(Employee))>}$

is not a T_ODMG temporal type, since the inner type

$\texttt{temporal}_{months}\texttt{(temporal}_{months}\texttt{(Employee))}$

is not a legal T_ODMG type in $\mathcal{T}_{temp}$ as it has been discussed in Example 3.2.

$\Diamond$

The set of types of our temporal model is defined as the union of object, literal and temporal types, that is,

$$\mathcal{T}_{temp} = \mathcal{OT}_{temp} \cup \mathcal{LT}_{temp} \cup \mathcal{TT}.$$

Where $\mathcal{OT}_{temp} = \mathcal{OT}_{class}$ (cf. Definition 2.2), $\mathcal{LT}_{temp}$ has been defined in Definition 3.7, and, finally, $\mathcal{TT}$ has been defined in Definition 3.6. In the following, we refer to the elements in the set $\mathcal{ST} = \mathcal{T}_{temp} \setminus \mathcal{TT}$ as *static types*.[3] Note that, according to Definition 3.7 a static type may contain as component a temporal type. For instance, the type $\texttt{list<temporal}_{months}\texttt{(Employee)>}$ presented in Example 3.3 is a static type even though its inner type $\texttt{temporal}_{months}$ `(Employee)` is temporal. This is because the type $\texttt{list<temporal}_{months}\texttt{(Employee)>}$ represents a list of "histories" of values related to employees, but it is not a history itself.

3.3 T_ODMG Values

In this section we define the set of legal values for each type in $\mathcal{T}_{temp}$. Similarly to ODMG literal types, given a T_ODMG literal type τ belonging to $\mathcal{LT}_{temp}$, we denote with $dom(\tau)$ the set of values of type τ. By contrast, if τ belongs to $\mathcal{OT}_{temp}$, then the set of values of type τ is the set of object identifiers belonging to τ. Since objects can be dynamically created and deleted, this set of object identifiers depends on time. Thus, we define a temporal extension π_{temp} of function π defined in the previous chapter (cf. Definition 2.5). We recall that $\mathcal{OI}$ is a set of object identifiers.

[3]That is, $\mathcal{ST} = \mathcal{OT}_{temp} \cup \mathcal{LT}_{temp}$.

Definition 3.8 (*Function π_{temp}*). Function π_{temp}: $\mathcal{OT}_{temp} \times \mathbb{N} \rightarrow 2^{\mathcal{OI}}$, for each type $\tau \in \mathcal{OT}_{temp}$ and time instant $t \in \mathbb{N}$ returns the set of the identifiers of objects that belong to the extent of τ as instances at time t. □

Since in Definition 2.5 $\pi(\tau)$ has been defined as the set of instances *currently* belonging to an object type, we have that $\pi_{temp}(\tau, now) = \pi(\tau)$.

In what follows function π^+_{temp} denotes function π_{temp} restricted to the proper instances.

Definition 3.9 (*Function π^+_{temp}*). Function π^+_{temp}: $\mathcal{OT}_{temp} \times \mathbb{N} \rightarrow 2^{\mathcal{OI}}$, for each type $\tau \in \mathcal{OT}_{temp}$ and time instant $t \in \mathbb{N}$ returns the set of the identifiers of objects that belong to the extent of τ as *proper* instances at time t. □

Obviously, for each $\tau \in \mathcal{OT}_{temp}$ and for each $t \in \mathbb{N}$ $\pi^+_{temp}(\tau, t) \subseteq \pi_{temp}(\tau, t)$.

We define now the set of legal values of each static type τ with respect to the time domain and with respect to a granule. Then, we define the set of values of temporal types.

Definition 3.10 (*Legal Values of a Static Type with Respect to the Time Domain*). Let $\tau \in \mathcal{ST}$ be a static type and $t \in \mathbb{N}$ be a time instant. $[\![\, \tau \,]\!]_t$ denotes the extension of type τ at time t:

$$[\![\, \tau \,]\!]_t = \begin{cases} [\![\, \tau \,]\!] & \text{if } \tau \in \mathcal{LT}_{temp} \text{ (cf. Definition 2.7)} \\ \pi_{temp}(\tau, t) & \text{if } \tau \in \mathcal{OT}_{temp} \end{cases}$$ □

Note that, according to the previous definition, only object types have extents which vary over time. Intuitively, the set of values of a literal type does not change over time since no literal value can be explicitly created or deleted, whereas objects belonging to object types are dynamically created and deleted, thus an object type extent depends on the considered instant.

When dealing with temporal granularities, the set of values of a static type with respect to the time domain is generalized to a granularity G through the following definition.

Definition 3.11 (*Legal Values of a Static Type with Respect to a Granule*). Let $\tau \in \mathcal{ST}$ be a static type, $G \in \mathcal{G}$ be a granularity, and $i \in \mathcal{IS}$ an index. $[\![\, \tau \,]\!]^G_i$ denotes the extension of type τ with respect to the ith granule of G:

$$[\![\, \tau \,]\!]^G_i = \bigcap_{t \in G(i)} [\![\, \tau \,]\!]_t.$$ □

Each ith granule of a granularity $G \in \mathcal{G}$ is identified by a label $l_i^G \in \mathcal{LABEL}_G$. In order to simplify the notation, we frequently use $[\![\, \tau \,]\!]_{l_i^G}$ to denote the legal values of a type with respect to a granule, thus:

$$[\![\, \tau \,]\!]_i^G = [\![\, \tau \,]\!]_{l_i^G}.$$

Such a notation is used mainly in examples.

Example 3.4 Let `Department` be an object type such that $[\![\, \texttt{Department} \,]\!]_{1997} = \{\texttt{d}_1,\texttt{d}_2, \texttt{d}_3\}$ where the label 1997 denotes the granule corresponding to year 1997, that is, the considered granularity is *years*. Then, for each instant belonging to 1997, $\texttt{d}_1$,$\texttt{d}_2$, $\texttt{d}_3$ must exist. Formally, $[\![\, \texttt{Department} \,]\!]_{1997} = \bigcap_{t \in years(i_{1997})} [\![\, \texttt{Department} \,]\!]_t$ where i_{1997} is the index corresponding to granule 1997 in granularity *years*. The idea is that if a department exists only during a portion of a year it does not belong to the extent of the object type `Department` of that year.

$\diamond$

We are now ready to define the set of legal values for temporal types.

Definition 3.12 (*Temporal Type Legal Values*). Let $temporal_G(\tau) \in \mathcal{TT}$ be a temporal type. $[\![\, temporal_G(\tau) \,]\!]$ denotes the set of legal values of type $temporal_G(\tau)$:

$$\begin{aligned} &[\![\, temporal_G(\tau) \,]\!] = \\ &\quad \{f \mid f : \mathcal{IS} \rightarrow \textstyle\bigcup_{i \in \mathcal{IS}} [\![\, \tau \,]\!]_i^G \text{ is a partial function such that} \\ &\quad \text{for each } i \in \mathcal{IS} \text{ if } f(i) \text{ is defined then } f(i) \in [\![\, \tau \,]\!]_i^G\} \end{aligned}$$

$\square$

The following example clarifies the above definition.

Example 3.5 Let `Department` be a class such that: $D = \bigcup_{i \in \mathcal{IS}} [\![\, \texttt{Department} \,]\!]_i^{years} = \{\texttt{d}_1,\texttt{d}_2, \texttt{d}_3, \ldots, \texttt{d}_n\}$, then, according to Definition 3.12:

$$\begin{aligned} &[\![\, temporal_{years}(\texttt{Department}) \,]\!] = \\ &\quad \{f \mid f : \mathcal{IS} \rightarrow D \text{ is a partial function such that for each} \\ &\quad i \in \mathcal{IS} \text{ if } f(i) \text{ is defined then } f(i) \in [\![\, \texttt{Department} \,]\!]_i^{years}\} \end{aligned}$$

Examples of functions, denoted as set of pairs, in $[\![\, temporal_{years}(\texttt{Department}) \,]\!]$ are:

$$v_1 = \{\langle i_{1992}, \texttt{d}_1 \rangle, \langle i_{1993}, \texttt{d}_4 \rangle\};$$

$$v_2 = \{\langle i_{1995},\ \mathtt{d}_2\rangle, \langle i_{1998},\ \mathtt{d}_2\rangle\};$$

$$v_3 = \{\langle i_{1992},\ \mathtt{d}_1\rangle, \langle i_{1993},\ \mathtt{d}_1\rangle, \langle i_{1994},\ \mathtt{d}_1\rangle\}$$

where i_{1992} is the index corresponding to the year granule labeled 1992, i_{1993} is the index corresponding to the year granule labeled 1993, and so on. We recall that temporal values are partial functions, thus, for instance, $v_1(i) = \perp$ for each $i \notin \{i_{1992}, i_{1993}\}$.

◊

In Example 3.5 we have denoted temporal values as set of pairs $\langle index, value\rangle$. However, when the function representing a temporal value is constant for a set of contiguous granules, that is, a time interval, we denote temporal values as set of pairs $\langle time\ interval, value\rangle$. For instance, $\{\langle [i_{1992}, i_{1994}]^{years},\ \mathtt{d}_1\rangle\}$ is a compact notation for the value $v_3 = \{\langle i_{1992},\ \mathtt{d}_1\rangle, \langle i_{1993},\ \mathtt{d}_1\rangle, \langle i_{1994},\ \mathtt{d}_1\rangle\}$ of type $temporal_{years}(\mathtt{Department})$ discussed in Example 3.5.

Note that the set of legal values of a given temporal type does not depend on the particular time instant. If $[\![\, temporal_G(\tau)\,]\!]_t$ denotes the set of values of type $temporal_G(\tau)$ with respect to time instant $t \in \mathbb{N}$, we can state that:

$$[\![\, temporal_G(\tau)\,]\!] \;=\; [\![\, temporal_G(\tau)\,]\!]_t \text{ for each } t \in \mathbb{N}.$$

This is in accordance with the intuition that only object type extents vary over time. Since the set of values of a given temporal type does not depend on time, we can extend the definition of the set of legal values of a static type with respect to a granule (Definition 3.11) to temporal types as follows:

$$[\![\, temporal_H(\tau)\,]\!]_i^G = \bigcap_{t\in G(i)} [\![\, temporal_H(\tau)\,]\!]_t = [\![\, temporal_H(\tau)\,]\!]\,.$$

We denote with $\mathcal{V}_{temp}$ the set of legal values for types in $\mathcal{T}_{temp}$. Let v be a value of type $temporal_G(\tau)$, we denote with $v(i)$ the value of v in ith granule of granularity G. We assume that, for each granularity H such that $H \preceq G$, and for each $i, j \in \mathcal{IS}$ such that $H(j) \subseteq G(i)$, the value of v in granule j of H is the one in the ith granule of G. This assumption is known in the temporal reasoning community [Sho87] as *downward hereditary* property. We simply denote such value as $v^H(j)$. For instance, let $v_1 \in [\![\, temporal_{years}(\mathtt{Department})\,]\!]$ be the temporal value presented in Example 3.5. Then $v_1^{months}(i_{01/1992}) = \mathtt{d}_1$, whereas $v_1^{months}(i_{01/1993}) = \mathtt{d}_4$.

Definition 3.13 (*Value Conversion*). Let v be a value of type $temporal_G(\tau)$ and H be a granularity such that $H \preceq G$, then v^H denotes the conversion of value v according to the downward hereditary property, that is:

$$v^H(j) = \begin{cases} v(i) & \text{if } H(j) \subseteq G(i) \\ \perp & \text{otherwise} \end{cases}$$

□

One can argue that downward hereditary property is not always appropriate. In our model temporal values are stored in objects as values of properties whose type is temporal. For instance, suppose that a value of an attribute is stored with respect to months, then one can argue that this fact does not imply that such value is the same for every day in the month. Suppose that such value represents the temperature of geographical object, then one can argue that if the temperature of a geographical object has been stored with respect to months, it is not realistic to deduce that the temperature in such object was the same for every day of the month. We believe that in most cases to assume downward hereditary property is realistic. In fact, storing the value of an attribute with respect to a granularity is somehow deciding the temporal precision associated with the information. If the attribute granularity is *months*, then for each granularity finer than *months*, such as, for instance, *days* this information is imprecise. However, since a value has been associated with each month, this information is the only one we have "closer" to the value of each day. Consider, for example, the value of the temperature of a geographical object stored monthly, probably such value is an average value, thus, it is realistic to deduce that every day the value of the temperature has been very close to the stored value. This implies that converting the value to the granularity *days* is a reasonable choice. We will see in Chapter 4 that this approach allows one to compare this value with the ones expressed with respect a finer granularity.

3.4 T_ODMG Type External Specifications

Object type external specifications in the ODMG object model have been formally defined in Section 2.5. We recall that an object type can be defined through a class or interface declaration. According to the ODMG terminology, such definition is the object type external specification. Actually, in T_ODMG an object type can be defined only through a class definition. We do not deal with interfaces in this context since they make more complex the notation without adding any contribution with respect to temporal features. The object type external specification is composed by two subparts: the object type external specification body and the object type external specification characteristics. Since in T_ODMG an object type external specification can be defined only through a class definition, in what follows we use the term class to denote the object type external specification.

Analogously to the ODMG object model, in T_ODMG the body of a class consists of an identifier, $c \in \mathcal{OT}_{temp}$, which represents the type of the objects belonging to the class, a set of property (attribute and relationship) specifications. We do not deal with operation specifications, which are part of the class body, in T_ODMG, since we focus on the structural aspects of a class which involve multiple temporal granularities.

T_ODMG supports two kinds of properties: *temporal* and *static* properties. A temporal property is a property whose value may change over time with respect to different granularities depending on the needs of the application domain, and whose values at different time granules are recorded in the database. The choice of the granularity for a property coincides in our model with the choice of the type for the property and allows the system to store the minimal amount of data. Since many different granularities exist and no granularity is inherently "better" than another, we believe that a temporal database system should support a wide range of temporal granularities and should allow the user to define his/her own application-specific granularities.

If a property p of a class c has type $temporal_G(\tau)$, then such property for objects in class c cannot vary more than once for each granule of G. Therefore, given the ith granule of granularity G, the value of the attribute is the same for each instant $t \in G(i)$. The kind of time we consider is *valid time*. We recall that valid time denotes the time a fact happens in reality. Thus, if the value of a property p is $v \in [\![temporal_G(\tau)]\!]$ such that $v(i) = \bar{v}$, $i \in \mathcal{IS}$, then the fact that the property p was equal to $\bar{v}$ in the ith granule of G refers to the *real time* in which the property p evaluates in $\bar{v}$ and not to the time in which this fact has been stored in the database.[4]

Among temporal properties we distinguish two other kind of properties: *mutable* and *immutable* temporal properties. Immutable properties are a particular kind of temporal properties. Indeed, an immutable property is a temporal property whose value is constant for each granule in which it is defined. Thus, immutable properties model temporal properties on which a semantic constraint is established. We do not impose immutable properties to be defined for each instant in which the object exists. However, once an immutable property gets a value, such value has to be constant for a time interval, which can be strictly included in the time interval during which the object exists. An example of temporal property is the one which relates a person to his/her father. The father is always the same for the person, but once he dies, such a relationship is undefined. Mutable and immutable temporal properties are modeled adding an extra component in each property, denoted as *a_imm* for attributes and *r_imm* for relationship. Such component stores a boolean value, which is equal to *true* if the property is immutable, $false$ otherwise.

A static property is a property whose value can change over time, but whose past values are *not* recorded in the database. We want to stress the fact that immutable properties are semantically different from static properties. Indeed, if we look at these kind of properties with respect to the query language asking which is the value of a static property at a time

[4]As discussed in Chapter 1, this time is known as *transaction time*.

instant arises a type error, since the type of the property is static. Whereas, asking which is the value of a temporal property at a time instant is of course legal and the answer depends on the value of the property. The peculiarity is that such value is the same for each time instant in which the property is defined.

The attributes of a class are characterized by a name, a type, which represents the set of legal values for that attribute, that is, the attribute *domain*, and a third information which stores whether the attribute is mutable or not, that is, the flag *a_imm*. The relationships of a class are characterized by a name, a type, an inverse relationship,[5] and the additional component *r_imm*, storing whether the relationship is mutable or not. In case of temporal properties, both mutable and immutable, the type of the considered property is a temporal type.

Classes, or equivalently, object types, are organized into an inheritance hierarchy, named EXTENDS hierarchy. Properties and features of the EXTENDS hierarchy in the ODMG object model has been described in Section 2.7. The impact of the introduction of time on the EXTENDS hierarchy will be investigated in Section 3.6. With respect to the ODMG object type external specification body (cf. Definition 2.8), we do not consider method definitions in T_ODMG, thus the component *meth* of the object type external specification body $(c, attr, rel, meth)$ is not considered. We recall that $\mathcal{AN}$ and $\mathcal{RN}$ denote a set of attribute and relationship names, respectively. In what follows we formally define T_ODMG object type external specification bodies.

Definition 3.14 (*T_ODMG Object Type External Specification Body*). In T_ODMG an external specification of an object type is a triple $(i, attr, rel)$, where:

$i \in \mathcal{OT}_{temp}$ is the type identifier;

attr is a set containing an item for each attribute of the object type. Each item is a triple (a_type, a_name, a_imm), where:

> $a\_type \in \mathcal{T}_{temp}$ is the attribute domain type;
>
> $a\_name \in \mathcal{AN}$ is the attribute name;
>
> $a\_imm \in [\![\texttt{boolean}]\!]$ is equal to *true* if the attribute is immutable, *false* otherwise;

rel is a set containing an item for each relationship of the object type. Each item is a 4-tuple $(r\_type, r\_name, r\_inv, r\_imm)$, where:

[5]Suppose that a one-to-one relationship is defined between two object types τ_1 and τ_2, then in τ_1 the type of the relationship is τ_2, whereas, in τ_2, the type of the inverse relationship is τ_1.

r_type can be of three different categories: $r\_type \in \mathcal{OT}_{temp}$, $r\_type \in \{\texttt{set <}\tau\texttt{>} \mid \tau \in \mathcal{OT}_{temp}\} \cup \{\texttt{bag <}\tau\texttt{>} \mid \tau \in \mathcal{OT}_{temp}\} \cup \{\texttt{list <}\tau\texttt{>} \mid \tau \in \mathcal{OT}_{temp}\}$, or $r\_type \in \{temporal_G(\tau) \mid \tau \in \mathcal{OT}_{temp}\} \cup \{temporal_G(\texttt{o\_constr<}\tau\texttt{>}) \mid \texttt{o\_constr} \in \{\texttt{set,bag,list}\}$ and $\tau \in \mathcal{OT}_{temp}\}$;

$r\_name \in \mathcal{RN}$ is the relationship name;

r_inv is a pair denoting the inverse traversal path of the relationship, that is, $r\_inv = (r\_inv\_type, r\_inv\_name)$ where:

$r\_inv\_type \in \mathcal{OT}_{temp}$ is the inverse relationship domain type;

$r\_inv\_name \in \mathcal{RN}$ is the inverse relationship name;

$r\_imm \in [\![\texttt{boolean}]\!]$ is equal to $true$ if the relationship is immutable, $false$ otherwise.

□

Given an object type external specification $(i, attr, rel)$, properties can be temporal or static. If a property p is temporal, that is its associated type belongs to $\mathcal{TT}$, then one is interested in storing the history of such property over time, according to a particular granularity. In case of immutable temporal properties, that is, a_imm, for attributes, or r_imm, for relationships, is equal to $true$, such a value is the same for each granule in which the property is defined. Of course, if a property is defined to be immutable, that is, its a_imm/r_imm component is set to true, then the type of such property has to be temporal. This is formalized by the following rule.

Rule 3.1 *Let $(i, attr, rel)$ be an object type external specification body. Then the following conditions have to be verified:*

- *for each $(a\_type, a\_name, a\_imm) \in attr$ such that $a\_imm = true$, then $a\_type \in \mathcal{TT}$;*
- *for each $(r\_type, r\_name, r\_inv, r\_imm) \in rel$ such that $r\_imm = true$, then $r\_type \in \mathcal{TT}$.*

If a property p is static, that is its associated type belongs to $\mathcal{ST}$, then one is interested in storing the current value of p. This means that the value of static properties can change over time, but one is not interested in storing changes.

Temporal relationships, both mutable and immutable, are modeled through temporal types where the inner type is an object type or, in the case of one-to-many and many-to-many temporal relationships, is a set, a bag, or a list of objects. This is the reason why in the specification of the rel component of a class (cf. Definition 3.14), r_type can be an object type, a collection literal type whose inner type is an object type, or a temporal type whose inner type is an object type or a collection literal type whose inner type is in turn an object type. The inverse of a temporal relationship can be non-temporal, that is, static. Indeed, for instance, one can be interested in storing the history of the relationship r according to which objects belonging to class c are related to objects belonging to class c', which corresponds to declaring r as a temporal relationship in c. Whereas, one can be interested in storing *only* the current value of the inverse relationship r_{inv} of r which relates objects in c' to objects in c. This corresponds to declaring r_{inv} as a static relationship in c'.

The following example clarifies the previous concepts.

Example 3.6 Consider two classes `Employee` and `Department` and a relationship between them storing for each employee the department in which he/she works and for each department the set of employees working in it. The definition of this relationship in ODMG is the following:

```
class Employee {...
  relationship Department works_in
     inverse Department::workers;...};

class Department { ...
  relationship set<Employee> workers
     inverse Employee::works_in;...};
```

With the introduction of temporal types, we can model somehow the history of these relationships. Suppose, for instance, we would like to store for each month the department in which the employee works. Then the T_ODMG definition of the relationship would be the following:

```
class Employee {...
  relationship temporal_months(Department) works_in
     inverse Department::workers;...};

class Department { ...
  relationship set<Employee> workers
     inverse Employee::works_in;...};
```

Note that the type of the relationship `workers` in class `Department` is still static, that is, its type is static. Of course, in this case, referential integrity has to be guaranteed at time instant *now*. That is, if an employee e works for a department d now, then e has to belong to the set of employees currently working for the department d.[6] ◊

[6]We elaborate on this in Section 3.7.

Similarly to the ODMG object model, in the $T_$ODMG object model, inheritance information, extent naming, and key specification are characteristics of classes. Class characteristics have been formally defined for ODMG in Definition 2.10. Since we do not consider schema evolution, the fact that a class is subclass of another and the name of its extent are established by its definition and they cannot be modified.[7] Thus, inheritance information and extent naming do not change over time. A key for a class is a list of property names which uniquely identifies each object of the class. With the introduction of the temporal dimension the concept of key has to be revised. We consider three kinds of keys: *static*, *absolute*, and *relative* [BFGM98]. Static keys are composed by static properties and they model the classic notion of key which has been discussed in the previous chapter. That is, a key is static if it is composed by static properties and if no two objects which belong to the *current* extent have the same value for these properties. We recall that static properties denote the value of the property at the time instant *now*, thus objects in this case have to belong to the current extent. Absolute and relative keys are composed by temporal properties. Intuitively, a key is absolute if it *always* identifies the objects of a class, that is, independently from the considered time instant. By contrast, a key is relative if, fixed a time instant, it identifies the objects of the class. That is:

- if the property is temporal and the key is *relative*, no two instances may have the same value for the property for an overlapping period of time.
- if the property is temporal and the key is *absolute*, no two instances may have the same value for the property in different time periods.

Example 3.7 The American social security number corresponds in Italy to what is known as *fiscal code*. Fiscal codes are automatically generated once a person is registered to the designated office and two persons can never have the same fiscal code even though one of the two persons is no more alive. First the following issue arises: is the fiscal code an immutable or a static property? From a conceptual point of view the fiscal code is an immutable property, since it never changes. However, depending on the application domain, one can regard the fiscal code as a static property. In this case, one can know the fiscal codes of objects currently belonging to the database, and the fiscal codes of objects which have belonged to the database in the past is unknown because they are not kept. For what concerns the constraint of the uniqueness of the fiscal code, if the property is immutable, then one can define the fiscal code to be an absolute key for the class `Person`. In this case fiscal codes cannot be recycled. If the property is declared as a static key, then in the current extent of each class no pair of objects can have the same fiscal code, but the fact that fiscal codes cannot be recycled cannot be guaranteed by static keys.

[7] Extent naming has not to be confused with the extent of the class, that is, the set of objects belonging to it, which varies over time, since objects can be created and deleted. Such information is recorded in the history of the class.

Consider now an attribute `room` of a class `Employee` storing the room number of the office of the employee. Suppose that the history of the offices changed by an employee has to be kept and that once the room is associated to an employee it is the same for the whole year. The type of the attribute `room` of class `Employee` would be $\texttt{temporal}_{years}\texttt{(short)}$. In addition, suppose that a room is allocated only to one employee every year. Then `room` is an example of a relative key for class `Employee`. Indeed, no pair of employees may have the same value for the room within the same extent for an overlapping period of time. $\diamond$

We elaborate more on key constraints in Section 3.7 where they are formalized. To the standard information represented by class characteristics in ODMG, in T_ODMG two additional ones have to be added: *history* and *lifespan*. Since the objects belonging to a class vary over time, each class also maintains the history of all the objects instances of the class over time. Such information is a temporal value storing the evolution of the set of objects instances of the class. Finally, each class has a lifespan, representing the time interval during which objects of that class can exist. For instance, objects representing cars can exist only after the car has been created. We make the assumption, common to most of the other models, that the lifespan of a class is contiguous, that is, it consists of a set of consecutive time instants. Note that the lifespan of a class establishes somehow a temporal constraint on the lifespan of the objects belonging to it. The lifespan of a class has not to be confused with the time in which the class exists in the database which is an information related to transaction time. For instance, we can store archeological information in a class c whose objects are archeological discoveries. In this case the lifespan of c represents the historical period to which the archeological discoveries belong, which is surely different from the time in which the class is part of the database!

Both these temporal information are stored with respect to the time domain, that is, with respect to the chronon granularity G_I. Since temporal properties can be expressed with respect to different temporal granularities some constraints have to be expressed in order to guarantee the consistency between the temporal information of the lifespan of the class and the temporal information stored in each object. Those consistency constraints are formally defined in Section 3.5.

In what follows, we report the definition of the characteristics of T_ODMG classes, which differs from the ones of ODMG classes in many respects. For instance, only the EXTENDS inheritance hierarchy is considered in this context. This follows from the fact that only class object types are considered in T_ODMG and the ISA hierarchy involves interfaces.

Definition 3.15 (*T_ODMG Class Characteristics*). The external specification characteristics of a T_ODMG class c are defined as a tuple $(super, extent, keys, history, lifespan)$, where:

$super \in \mathcal{OT}_{temp}$ is the class name of the direct superclass of c;

extent is the extent name;

keys is a set of pairs of the form ⟨*list of property names, key type*⟩ where each list represents a key of the class c and each key type establishes whether the key is static, absolute or relative, that is, $keys = \bigcup_{j=1}^{n}\{\langle(p_1, \ldots, p_j), kt\rangle \mid p_i \in \mathcal{AN} \cup \mathcal{RN},\ i \in [1, j]$ such that $(p_1, \ldots, p_j)$ is a key of the class and $kt \in \{static, absolute, relative\}\}$;

history is a value of type $temporal_{G_I}$(`set<c>`) representing the set of objects instances of c over time;

lifespan is a temporal interval with respect to granularity G_I representing the interval during which objects of the class can exist. □

In what follows, given a class c whose body is the triple $(i, attr, rel)$ we denote through the dot notation each component of class c. That is, $c.i$ denotes the class identifier,[8] $c.attr$ denotes the attribute component, and so on. The same applies for the class characteristics. We recall that we denote with the symbol "$\emptyset$" the lack of a particular characteristic. A strong relationship exists between the class history and function π_{temp} defined in Section 3.3. This relationship is formalized by the following rule.

Rule 3.2 *Let $c \in \mathcal{OT}_{temp}$ be a class and $id \in \mathcal{OI}$ be an identifier of an object instance of c. Let $t \in \mathbb{N}$ be a time instant such that $G_I(t) = \{t\}$, then the following condition holds:*

$$id \in \pi_{temp}(c, t) \text{ if and only if } id \in c.history(t). \qquad \Box$$

Finally, we can state that the definition of a T_ODMG class external specification is analogous to the one for ODMG classes (see Definition 2.12) where the object type identifier i belongs to $\mathcal{OT}_{temp}$. We give now an example of a T_ODMG class storing temporal and non-temporal information.

Example 3.8 Let `Course` be a class storing the information concerning university courses. For each course we are interested in representing the code of the course, `C_code`, the `name`, the `year` in which the course has to be taken, that is, an integer between 1 and 5, and the `room` in which the course takes place. In addition, the professor (`T_professor`), the teaching assistant (`T_assistant`) and the students taking the course (`P_students`) information are stored with respect to different granularities. For example, the professor teaching the course is the same during one year, whereas the teaching assistant is the same during a semester. Moreover, for internal statistics, for each course the set of students attending the course during one year is stored. In the following let `Professor`, `Researcher`, `Student` and `Person`

[8] Usually c is an overloaded symbol that denotes both the class and the class identifier. That is, $c.i = c$.

be classes and let *days*, *weeks*, *months*, *semesters*, and *years* be granularities such that $days \preceq months \preceq semesters \preceq years$, and $days \preceq weeks$. Thus, the set of granularities $\mathcal{G}$ is $\{days, weeks, months, semesters, years, G_I\}$. The following is a class definition for a class `Course`, in ODL syntax enriched with temporal types. In case of temporal properties when nothing is specified, then the property is mutable. Immutable properties are explicitly denoted through the keyword `immutable` before the property definition.

```
class Course
  (extent courses, key ((C_code),absolute),((name),relative))
  {
  immutable attribute temporal_years(string) C_code;
  attribute string name;
  attribute short year;
  attribute temporal_years(short) room;
  relationship temporal_years(Professor) T_professor
     inverse Professor::teaches;
  relationship temporal_semesters(Researcher) T_assistant
     inverse Researcher::assists;
  relationship temporal_years(set<Student>) P_students
     inverse Student::attends;
  }
```

Let $\mathtt{i}_1$, $\mathtt{i}_2$, and $\mathtt{i}_3$ be objects belonging to class `Course`. The following is an example of class `Course` external specification:

$ext\_spec = (i, attr, rel)$ where:

> i = `Course`;
>
> $attr = \{(\mathtt{temporal}_{years}(\mathtt{string}),\ \mathtt{C\_code},\ \mathtt{true})\ (\mathtt{string},\ \mathtt{name},\ \mathtt{false}),\ (\mathtt{short},\ \mathtt{year},\ \mathtt{false}),\ (\mathtt{temporal}_{years}(\mathtt{short}),\ \mathtt{room},\ \mathtt{false})\}$
>
> $rel = \{(\mathtt{temporal}_{years}(\mathtt{Professor}),\ \mathtt{T\_professor}, (\mathtt{Professor}, \mathtt{teaches}),\ \mathtt{false}),$
> $(\mathtt{temporal}_{semesters}(\mathtt{Researcher}),\ \mathtt{T\_assistant}, (\mathtt{Researcher}, \mathtt{assists}),\ \mathtt{false})$
> $(\mathtt{temporal}_{years}(\mathtt{set{<}Student{>}}),\ \mathtt{P\_students},\ (\mathtt{Student}, \mathtt{attends}),\ \mathtt{false})\}$;

$charact = (\emptyset, extent, keys, history, lifespan)$

> $extent$ = `courses`;
>
> $keys = \{\langle(\mathtt{C\_code}), \mathtt{absolute}\rangle, \langle(\mathtt{name}), \mathtt{relative}\rangle\}$;
>
> $history = \{\langle[1, 99]^{G_I}, \{\mathtt{i}_2, \mathtt{i}_3\}\rangle, \langle[100, \infty]^{G_I}, \{\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3\}\rangle\}$;
>
> $lifespan = [1, \infty]^{G_I}$.

◊

3.5 T_ODMG Objects

Analogously to ODMG (cf. Section 2.6), in T_ODMG each object is characterized by an object identifier, a set of names, and a state. The state of an object is characterized by the set of properties (attributes and relationships) of the most specific object type to which the object belongs and the corresponding values. In T_ODMG each object has an associated *lifespan* representing the set of instants during which the object exists in reality.[9] As for classes, the object lifespan is expressed with respect to the chronon granularity G_I. We assume the lifespan of an object to be contiguous. Finally, for each object we maintain information on the most specific class to which it belongs to. In accordance to the ODMG model, which do not support objects having more than one most specific, in T_ODMG objects have a unique most specific class to which they belong to. In addition, in T_ODMG, as in ODMG, objects cannot migrate along the inheritance hierarchy. Thus such class is the same for the whole life of the object. Actually, in ODMG an additional component has been considered for objects: the lifetime. The lifetime of an object determines whether the objects is transient or persistent. We believe that storing temporal information is meaningful only for persistent objects that persist after the termination of the application which creates them. Thus, in T_ODMG we consider only persistent objects and we drop the lifetime component from objects to simplify the notation.

The formal definition of a T_ODMG object is very similar to the one of ODMG object (cf. Definition 2.13) where attribute and relationship values belong to $\mathcal{V}_{temp}$ and the lifespan component is added. Thus, we do not report the formal definition of T_ODMG object, rather we report the formal definition of the lifespan of a T_ODMG object.

Definition 3.16 (*Object Lifespan*). Given an object o the $lifespan$ of o is a temporal interval with respect to G_I denoting the set of instants in which o exists. □

Summarizing, a T_ODMG object is a 6-tuple $(id, N, v, r, c, [i,j]^{G_I})$ where id is the identifier of the object, N is the set of object names, v and r are the attribute and relationship values, respectively, c is the most specific class to which the object belongs, and, finally, $[i,j]^{G_I}$ is a temporal interval representing the object lifespan. In what follows an example of a T_ODMG object is given.

Example 3.9 Consider the class `Course` of Example 3.8. Suppose that $\mathtt{p}_1$, $\mathtt{p}_2$ are object identifiers of type `Professor`, $\mathtt{r}_1$ and $\mathtt{r}_2$ are object identifiers of type `Researcher`, $\mathtt{s}_1, \ldots, \mathtt{s}_{20}$ are object identifiers of type `Student`, and $\mathtt{i}_1$ is an object identifier of type `Course`. The following is an example of object belonging to class `Course`:

[9] This is in accordance with our model which refers to valid time.

$(\mathtt{i}_1, \{temporal\_db\}, v, r, \mathtt{Course}, [100, \infty]^{G_I})$,

where:

$v = (\mathtt{C\_code}\texttt{:}\{\langle [i_{1999}, \infty]^{years}, \text{`}\mathtt{c0112}\text{'}\rangle\}, \mathtt{name}\texttt{:}\text{`}\mathtt{Temporal\ Databases}\text{'}, \mathtt{year}\texttt{:}5,$
$\quad \mathtt{room}\texttt{:}\{\langle i_{1999}, 235\rangle, \langle i_{2000}, 312\rangle\})$

$r = (\mathtt{T\_professor}\texttt{:} \quad \{\langle i_{1999}, \mathtt{p}_2\rangle, \langle i_{2000}, \mathtt{p}_1\rangle\},$
$\quad \mathtt{T\_assistant}\texttt{:}\{\langle i_{1st\_sem\_1999}, \mathtt{r}_1\rangle, \langle i_{2nd\_sem\_1999}, \mathtt{r}_2\rangle,$
$\quad\quad \langle i_{1st\_sem\_2000}, \mathtt{r}_2\rangle, \langle i_{2nd\_sem\_2000}, \mathtt{r}_2\rangle\},$
$\quad \mathtt{P\_students}\texttt{:}\{\langle i_{1999}, \{\mathtt{s}_1, \ldots, \mathtt{s}_{10}\}\rangle, \langle i_{2000}, \{\mathtt{s}_{11}, \ldots, \mathtt{s}_{20}\}\rangle\})$.

◊

In ODMG we have formally defined the notion of consistent instance (cf. Definition 2.14). We recall that each object must be a consistent instance of its most specific class c. Intuitively, an object is a consistent instance of a class c if its state matches the body of c. That is, each attribute value is a legal value for the corresponding attribute type and each relationship value is a legal value for the corresponding relationship type. In addition, the object must not have additional properties with respect to the properties specified in c. Since now we are in a temporal context, additional constraints have to be devised. Indeed, for instance, the lifespan of the object must be included in the lifespan of c. Additionally, constraints must be imposed on the relationships occurring between the lifespan of an object and the domain of the temporal properties it contains, in that the domain of a temporal properties should not exceed the lifespan boundary. The following example clarifies the discussion.

Example 3.10 Consider an object modeling a professor and suppose that this object contains an attribute `salary` of type $\mathtt{temporal}_{months}(\mathtt{short})$ which records the monthly salary of the professor. Suppose that a professor, say *Bob*, is hired on March, 15th 1999 and fired on June, 15th 2000. We have to make sure that the attribute `salary` in the professor object corresponding to `Bob` does not contain values for months which are not included in the interval $[i_{03/1999}, i_{06/2000}]^{months}$. ◊

We report the formal definition of *T*_ODMG consistent instance which includes both non-temporal and temporal constraints.

Definition 3.17 (*T_ODMG Consistent Instance*). An object $o = (id, N, (a_1 : v_1^a, \ldots, a_n : v_n^a), (r_1 : v_1^r, \ldots, r_m : v_m^r), c, [i_o, j_o]^{G_I})$ is a consistent instance of a class $\bar{c}$ whose type external specification body is the triple $(\bar{c}, attr, rel)$, and whose type characteristics are the 5-tuple $(super, extent, keys, history, [i_c, j_c]^{G_I})$ if the following conditions hold:

- the lifespan of o is included into the lifespan of $\bar{c}$, that is, $[i_o, j_o]^{G_I} \subseteq [i_c, j_c]^{G_I}$;
- $\{t\} \in [i_o, j_o]^{G_I}$ if and only if $id \in history(t)$, where $G_I(t) = \{t\}$;
- for each $(\tau, a, im) \in attr$, then k exists, $k \in [1, n]$, such that $a = a_k$ and the following conditions are verified:
  1. if $\tau \in \mathcal{ST}$: $v_k^a \in [\![\tau]\!]_{now}$;
  2. if $\tau \in \mathcal{TT}$, $\tau = temporal_G(\tau')$ all the following conditions hold:
     - $v_k^a \in [\![\tau]\!]$;
     - $\forall i \in \mathcal{IS}$ such that $v_k^a(i)$ is defined, $G(i) \cap (\bigcup_{i_o \leq k \leq j_o} \{G_I(k) \mid k \in \mathcal{IS}\}) \neq \emptyset$;
     - if $im = true$, then $\forall i, j \in \mathcal{IS}$ such that $v_k^a(i)$ and $v_k^a(j)$ are defined, $v_k^a(i) = v_k^a(j)$, and $\forall h \in \mathcal{IS}$, $i \leq h \leq j$, $v_k^a(h) = v_k^a(i)$;
- for each $(\tau, r, r\_inv, im) \in rel$, then k exists, $k \in [1, m]$, such that $r = r_k$ and the following conditions are verified:
  1. if $\tau \in \mathcal{ST}$: $v_k^r \in [\![\tau]\!]_{now}$;
  2. if $\tau \in \mathcal{TT}$, $\tau = temporal_G(\tau')$, all the following conditions hold:
     - $v_k^r \in [\![\tau]\!]$;
     - $\forall i \in \mathcal{IS}$ such that $v_k^r(i)$ is defined, $G(i) \cap (\bigcup_{i_o \leq k \leq j_o} \{G_I(k) \mid k \in \mathcal{IS}\}) \neq \emptyset$;
     - if $im = true$, then $\forall i, j \in \mathcal{IS}$ such that $v_k^r(i)$ and $v_k^r(j)$ are defined, $v_k^r(i) = v_k^r(j)$, and $\forall h \in \mathcal{IS}$, $i \leq h \leq j$, $v_k^r(h) = v_k^r(i)$;
- for each a_i, $i \in [1, n]$, $(\tau, a, im) \in attr$ exists such that $a = a_i$;
- for each r_i, $i \in [1, m]$, $(\tau, r, r\_inv, im) \in rel$ exists such that $r = r_i$. $\square$

Note that in the previous definition the consistency of static properties is checked with respect to time instant *now*. Indeed, a static property denotes a temporal information related to current time. By contrast, for each temporal property three conditions are checked. The first condition establishes the correctness of the value with respect to the property domain. The second condition establishes that the property is not defined outside the object lifespan. Finally, the last condition establishes the fact that immutable properties are really immutable, that is, they are defined and equal for a set of contiguous granules. In addition, the latter two conditions in Definitions 3.17 check that the object does not have extra components with respect to the ones specified for the class.

Consider again Example 3.10. The conditions imposed by Definition 3.17 ensure that the domain of the `salary` attribute would not contain granules preceding the one corresponding to March 1999 and following the one corresponding to June 2000. This ensures that queries such as: *Retrieve the salary of Bob on August 2001* would result in an empty answer.

Additionally, we have to make sure that also queries such as: *Retrieve the salary of Bob on March, 12th 1999* would result in an empty answer, since Bob has been hired on March 15th. These additional checks are performed at the level of the query language in Chapter 4, that ensures that no object property can be queried at a time instant which is not included into the object lifespan. Let $G(i)$ be a granule in which a property is defined and $[i_o, j_o]^{G_I}$ be the object lifespan. Then, such constraint is formulated in Definition 3.17 as follows:

$$G(i) \cap (\bigcup_{i_o \leq k \leq j_o} \{G_I(k) \mid k \in \mathcal{IS}\}) \neq \emptyset$$

where $\bigcup_{i_o \leq k \leq j_o} \{G_I(k) \mid k \in \mathcal{IS}\}$ converts the lifespan of the object, that is, a set of granules belonging to G_I into a set of time instants, so that the set operation $\cap$ can be correctly applied.

Example 3.11 Consider the class `Course` of Example 3.8 and the object o of Example 3.9, and suppose that time instant 100 corresponds to the first instant of `January 1999`. Object o is consistent since it is a consistent instance of class `Course`, in that:

- $[100, \infty]^{G_I} \subseteq [1, \infty]^{G_I}$, thus the lifespan of o is included into the lifespan of class `Course`;
- $\{t\} \in [100, \infty]^{G_I}$ if and only if $\mathtt{i}_1 \in$ `Course`.$history(t)$, where $G_I(t) = \{t\}$;
- for each $(\tau, a, im) \in \{$(`temporal`$_{years}$(`string`),`C_code`,`true`),(`string`,`name`,`false`), (`short`,`year`,`false`),(`temporal`$_{years}$(`short`),`room`,`false`)$\}$ the corresponding attribute value is defined and its type is correct with respect to the attribute component of `Course` (cf. Example 3.9);
- in addition, `C_code`, which is the unique immutable property is really immutable, in particular, since 100 corresponds to the first instant of `January 1999`, its value $\{\langle [i_{01/1999}, \infty]^{years}$,'c0112'$\rangle\}$ is constant for all the object lifespan $[100, \infty]^{G_I}$;[10]
- for each $(\tau, r, r\_inv, im) \in \{$(`temporal`$_{years}$(`Professor`), `T_professor`, (`Professor`, `teaches`),`false`),(`temporal`$_{semesters}$(`Researcher`), `T_assistant`, (`Researcher`, `assists`),`false`),(`temporal`$_{years}$(`set<Student>`),`P_students`,(`Student`,`attends`), `false`)$\}$ the corresponding relationship value is defined and its type is correct with respect to the relationship component of `Course` (cf. Example 3.9);
- each granule in which a temporal property of o is defined has a non-empty intersection with the lifespan of o;

[10]Note that, since 100 corresponds to the first instant of `January 1999`, $[i_{01/1999}, \infty]^{years}$ and $[100, \infty]^{G_I}$ denote the same portion of the time domain.

- finally, the object does not have extra components with respect to the ones specified in the class.

$\diamond$

3.6 Impact of Temporal Types on Inheritance

In this section we first consider the impact of temporal types on the inheritance relationship, then we investigate the issue related to the refinement of properties along the inheritance relationship.

3.6.1 Subtyping and Temporal Types

In ODMG two inheritance relationships are considered: ISA and EXTENDS. Characteristics and related issues of these two inheritance relationships have been extensively discussed in the previous chapter (cf. Section 2.7). As we have already pointed out, in T_ODMG we restrict ourselves to class object types. This fact implies that only the EXTENDS relationship is considered. Thus, subtyping relationship for static types in our model is defined as in Definition 2.18 where the items related to the ISA (second item) and named types (the last three items) are dropped.

First we extend Rule 2.6 to function π_{temp} (cf. Definition 3.8).

Rule 3.3 *Let τ_1 and $\tau_2 \in \mathcal{OT}_{temp}$ be two object types such that $\tau_2 \leq_T \tau_1$, then for each $t \in \mathbb{N}$, $\pi_{temp}(\tau_2, t) \subseteq \pi_{temp}(\tau_1, t)$.*

We extend now Proposition 2.1 to the set of values of a given type with respect to a time instant.

Proposition 3.1 *Let τ_1 and $\tau_2 \in \mathcal{ST}$ be two types such that $\tau_2 \leq_T \tau_1$ and $t \in \mathbb{N}$ be a time instant, then $[\![\tau_2]\!]_t \subseteq [\![\tau_1]\!]_t$.*

$\bigcirc$

Proof 3.1 The set of static types $\mathcal{ST}$ is defined as the union of literal and object types, that is, $\mathcal{ST} = \mathcal{LT}_{temp} \cup \mathcal{OT}_{temp}$. If $\tau_1, \tau_2 \in \mathcal{ST}$ such that $\tau_2 \leq_T \tau_1$, then either $\tau_1, \tau_2 \in \mathcal{LT}_{temp}$ or $\tau_1, \tau_2 \in \mathcal{OT}_{temp}$.[11]

If $\tau_1, \tau_2 \in \mathcal{LT}_{temp}$, then, according to Definition 3.10, $[\![\tau_1]\!]_t = [\![\tau_1]\!]$, $\forall t \in \mathbb{N}$, and $[\![\tau_2]\!]_t = [\![\tau_2]\!]$, $\forall t \in \mathbb{N}$. Proposition 2.1 states that if $\tau_2 \leq_T \tau_1$, then $[\![\tau_2]\!] \subseteq [\![\tau_1]\!]$, which implies $[\![\tau_2]\!]_t \subseteq [\![\tau_1]\!]_t$, $\forall t \in \mathbb{N}$.

[11]Note that no subtyping relationship can be established between a literal type and an object type.

If $\tau_1, \tau_2 \in \mathcal{OT}_{temp}$, then, according to Definition 3.10, $[\![\,\tau_1\,]\!]_t = \pi_{temp}(\tau_1, t)$, and $[\![\,\tau_2\,]\!]_t = \pi_{temp}(\tau_2, t)$. Since π_{temp} satisfies Rule 3.3, then the proposition is trivially proved.

$\bigcirc$

The following definition extends Definition 2.18 to temporal types.

Definition 3.18 (*Temporal Subtypes*). Let τ_1, $\tau_2 \in \mathcal{TT}$ be two temporal types, such that $\tau_1 = temporal_G(\tau_1')$ and $\tau_2 = temporal_G(\tau_2')$. Then, τ_2 is a subtype of τ_1 (denoted as $\tau_2 \leq_T \tau_1$) if and only if $\tau_2' \leq_T \tau_1'$. $\square$

Thus, the subtype relationship of Definition 2.18 is extended adding the pairs of temporal types of Definition 3.18. No other subtyping relationship can be devised. For instance, neither temporal types are subtypes of static types, nor viceversa. Similarly, the extent inclusion property presented in Proposition 2.1 is extended to temporal types as follows.

Proposition 3.2 *Let τ_1, $\tau_2 \in \mathcal{TT}$ be two temporal types such that $\tau_2 \leq_T \tau_1$, then $[\![\,\tau_2\,]\!] \subseteq [\![\,\tau_1\,]\!]$.* $\bigcirc$

Proof 3.2 Let $\tau_2 = temporal_G(\tau_2')$ and $\tau_1 = temporal_G(\tau_1')$. To prove the proposition we should prove that the set of values of type $temporal_G(\tau_2')$ is a subset of the set of values of type $temporal_G(\tau_1')$, that is, $[\![\,temporal_G(\tau_2')\,]\!] \subseteq [\![\,temporal_G(\tau_1')\,]\!]$. According to Definition 3.12:

$$
\begin{aligned}
&[\![\,temporal_G(\tau_2')\,]\!] = \\
&\qquad \{f \mid f : \mathcal{IS} \to \textstyle\bigcup_{i \in \mathcal{IS}} [\![\,\tau_2'\,]\!]_i^G \text{ is a partial function such that} \\
&\qquad \text{for each } i \in \mathcal{IS} \text{ if } f(i) \text{ is defined then } f(i) \in [\![\,\tau_2'\,]\!]_i^G\}
\end{aligned}
$$

and

$$
\begin{aligned}
&[\![\,temporal_G(\tau_1')\,]\!] = \\
&\qquad \{f \mid f : \mathcal{IS} \to \textstyle\bigcup_{i \in \mathcal{IS}} [\![\,\tau_1'\,]\!]_i^G \text{ is a partial function such that} \\
&\qquad \text{for each } i \in \mathcal{IS} \text{ if } f(i) \text{ is defined then } f(i) \in [\![\,\tau_1'\,]\!]_i^G\}
\end{aligned}
$$

First, we prove that the codomain of functions in $[\![\,temporal_G(\tau_2')\,]\!]$ is included in the codomain of functions in $[\![\,temporal_G(\tau_1')\,]\!]$, which implies that more functions can be built in $[\![\,temporal_G(\tau_1')\,]\!]$ with respect to $[\![\,temporal_G(\tau_2')\,]\!]$. Thus, we have to prove that:

$$\textstyle\bigcup_{i \in \mathcal{IS}} [\![\,\tau_2'\,]\!]_i^G \subseteq \bigcup_{i \in \mathcal{IS}} [\![\,\tau_1'\,]\!]_i^G.$$

To prove this we prove that for each index $i \in \mathcal{IS}$, an index $j \in \mathcal{IS}$ exists such that:

$$[\![\, \tau_2' \,]\!]_i^G \subseteq [\![\, \tau_1' \,]\!]_j^G.$$

Let $i = j$, then:

$$[\![\, \tau_2' \,]\!]_i^G =^a \bigcap_{t \in G(i)} [\![\, \tau_2' \,]\!]_t \subseteq^b \bigcap_{t \in G(i)} [\![\, \tau_1' \,]\!]_t =^c [\![\, \tau_1' \,]\!]_i^G.$$

Where a holds for Definition 3.11, b is implied by Proposition 3.1, and, finally, c holds for Definition 3.11.

In addition, this implies that, given $f \in [\![\, temporal_G(\tau_2') \,]\!]$, that is, $f : \mathcal{IS} \to \bigcup_{i \in \mathcal{IS}} [\![\, \tau_2' \,]\!]_i^G$ a partial function such that for each $i \in \mathcal{IS}$ if $f(i)$ is defined then $f(i) \in [\![\, \tau_2' \,]\!]_i^G$, $f(i) \in [\![\, \tau_1' \,]\!]_i^G$, that is, $f \in [\![\, temporal_G(\tau_1') \,]\!]$. $\bigcirc$

The extent inclusion property is automatically satisfied by two classes related by the EXTENDS relationship, whereas to ensure substitutability some conditions must be satisfied. Those conditions are related to the fact that each subclass must contain all properties (attributes and relationships) of its superclasses. Apart from the inherited features, additional features can be introduced in a subclass. In T_ODMG we allow inherited features to be redefined (overwritten) in a subclass under a number of restrictions. In the next section we investigate the redefinition of property domains along the inheritance hierarchy.

3.6.2 Property Refinement

In ODMG neither property nor method signature redefinition along the inheritance hierarchy are allowed.[12] In order to investigate this issue in an object-oriented temporal model supporting multiple granularity management, we drop this restriction in T_ODMG and allow the redefinition of a property domain by specializing the domain of the property, performing what is known as *covariant* redefinition [AC96]. In the thesis, since we focus on the structural part of classes, we do not deal with method signature refinement, but our approach can be easily extended to the refinement of method signatures.

We recall that, even in a non-temporal context, property domain redefinition along the inheritance hierarchy must be considered carefully. The covariant redefinition of properties reflects what is usually needed when creating a taxonomy of classes. Indeed, when specializing a class the designer usually needs to add new properties or to specialize the existing

[12]We recall that in ODMG method implementation redefinition along the inheritance hierarchy is allowed.

ones. The problems arising when properties are refined in a covariant way along the inheritance hierarchy are well-known and have been firstly recognized by Cardelli [Car88]. They can be summarized by saying that statically type correct assignments to values of covariantly redefined properties may cause run-time type errors. Many different solutions have been proposed [ACO85, Car88, CM92, Mei92]. Our approach is similar to the one proposed for O_2 [LRV90] and Chimera [GBB98]. Such an approach consists in considering property domains 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 property, we dinamically check that the value is appropriate for the domain.

In the context of temporal types, property refinement means, for example, that a property defined with granularity *months* can be refined in a property with granularity *years* on the same domain or on a most specific one. This situation does not pose problems with respect to substitutability, since thanks to downward hereditary property once we have stored a value with respect a year we can access the value in each month of that year.

However, we would also like to allow a property defined with respect to a given granularity to be refined in a property with respect to a finer granularity on the same domain or on a most specific one. An example is to refine a property defined with granularirty *years* in a property with granularity *months*. The reason is that, according to our point of view, property refinement is a meaningful way for adding information in the refined property. More precisely, by refining the granularity associated with a property, one can specify more detailed information for a given time interval. For example, if a property p defined with granularity *years* is specialized in a subclass in a property with granularity *months*, more information for that property can be stored in the subclass. In the superclass, only values for each year can be stored for p, whereas in the subclass values for each month of each year can be stored for p.

However, to ensure substitutability, in case of property refinement where the refining property domain has a granularity finer than the refined property domain, we need to introduce a *coercion* function to compute the value of the refined property with respect to the coarser granularity. The following example clarifies these concepts.

Example 3.12 Let $\mathtt{c}_1$ and $\mathtt{c}_2$ be two classes such that $\mathtt{c}_2 \leq_T \mathtt{c}_1$, that is, $\mathtt{c}_1 = EXTENDS(\mathtt{c}_2)$. Moreover, let $\mathtt{a}$ be a temporal attribute whose domain is $temporal_G(\tau)$ in $\mathtt{c}_1$ and whose refined domain in $\mathtt{c}_2$ is $temporal_H(\tau)$. For simplicity, we assume the inner type τ to be the same, thus we can focus on temporal granularities. In what follows we denote through the keyword `ref` the refinement of a property in subclass. We can distinguish two cases:

<u>Coarser granularity in the refined domain.</u> $G \preceq H$, for example $G = months$ and $H = years$:

```
class c₁{...attribute temporal_months(τ) a;...}
class c₂extends c₁{...ref attribute temporal_years(τ) a;...}
```

In this case, no problems arise. When an object of type $\mathtt{c}_1$ is accessed, we expect the type of attribute a to be $temporal_{months}(\tau)$, but if such object is an instance of $\mathtt{c}_2$ we will have an object whose attribute a is of type $temporal_{years}(\tau)$. However, since we have assumed downward hereditary property, if attribute a has a value for a given year, such value is the same for each month of the year. As we have already discussed in Section 3.3, indeed, a value of type $temporal_H(\tau)$ can be accessed in a granule of granularity G, $G \preceq H$. Thus, we do not need any additional information to map one granularity onto the other while accessing objects of type $\mathtt{c}_1$.

<u>Finer granularity in the refined domain.</u> $H \preceq G$, for example $H = months$ and $G = years$:

```
class c₁{...attribute temporal_years(τ) a;...}
class c₂extends c₁{...ref attribute temporal_months(τ) a;...}
```

In this case for an instance of type $\mathtt{c}_2$ we may know the value of the attribute just in some months of the year, but not in all the months of the year. Moreover, the attribute may take different values in different months of the year. Thus, the problem is to determine which value among the ones in which the attribute is defined must be considered for the coarser granularity. Suppose for example that one is interested in knowing the value of attribute a corresponding to year 1994. Then, all the values corresponding to the months of such year will be retrieved and the value related to year 1994 will be computed according to a *coercion function*. $\diamond$

Our idea is to add information to the refined property which allows one to choose the value. The general idea is that if a property of type $temporal_G(\tau_1')$ is refined in a subclass into a property of type $temporal_H(\tau_2')$, with $\tau_2' \leq_T \tau_1'$ and $H \preceq G$, a *coercion function* must be defined for the property, according to which the property value is computed. When the property is accessed, with respect to a specific granule j of granularity G, the coercion function is applied to all the granules $i_1, \ldots, i_n$ such that $H(i_k) \subseteq G(j)$, $k \in [1, n]$. The following definition formalizes the notion of coercion function.

Definition 3.19 (*Coercion Function*). Let $\tau_1 = temporal_G(\tau_1')$ and $\tau_2 = temporal_H(\tau_2')$ be two temporal types such that $\tau_2' \leq_T \tau_1'$ and $H \preceq G$. A coercion function:

$$C : [\![\, temporal_H(\tau_2') \,]\!] \rightarrow [\![\, temporal_G(\tau_1') \,]\!]$$

```
⟨prop_spec⟩ ::= ⟨attr_spec⟩ | ⟨rel_spec⟩

⟨attr_spec⟩ ::= [immutable] attribute ⟨domain_type attribute_name⟩ |
              ref attribute ⟨domain_type attribute_name [⟨ coerc_func ⟩]⟩

⟨rel_spec⟩ ::= [immutable] relationship ⟨domain_type relationship_name⟩
                inverse ⟨inv_domain_type::inv_relationship_name⟩ |
              ref relationship ⟨domain_type relationship_name⟩
                inverse ⟨inv_domain_type::inv_relationship_name [⟨ coerc_func ⟩]⟩

⟨coerc_func⟩ ::= ⟨selective_coerc_func⟩ | ⟨aggregate_coerc_func⟩ | ⟨user-def_coerc_func⟩

⟨selective_coerc_func⟩ ::= first | last | Proj(index) | main | all

⟨aggregate_coerc_func⟩ ::= min | max | avg | sum

⟨user-def_coerc_func⟩ ::= meth_inv
```

Figure 3.2: Extension of the ODL syntax for defining properties with coercion functions

is a partial function that maps values of type $temporal_H(\tau_2')$ into values of type $temporal_G(\tau_1')$. □

A large variety of coercion functions could be devised. We have developed a simple language for defining coercion functions. In our approach, a coercion function is associated with a property definition, since such function establishes how values of the properties have to be coerced to the coarser granularity. The syntax in BNF form of property specifications extended to allow the specification of coercion functions, whose semantics will be clarified in the remainder of the section, is given in Figure 3.2. With reference to Figure 3.2 terminal symbol `index` denotes an element in $\mathcal{IS}$, `meth_inv` denotes a method invocation. As specified in the BNF grammar of Figure 3.2, depending on how the value of a granule $G(j)$ is computed with respect to the values of the granules $H(i)$, such that $H(i) \subseteq G(j)$, coercion functions can be classified into three categories: *selective*, *aggregate*, and *user-defined* coercion functions, whose meaning is formally defined in Definition 3.20. In Figure 3.2 terminal symbols `first`, `last` and `Proj(index)` denote selective coercion functions of obvious meaning. Coercion function `main`, for each granule in the coarser granularity, returns the value which appears most frequently in the included granules of the finer one. Coercion function `all`, for each granule in the coarser granularity, returns the value which always appears in the included granules of the finer one if this value exists, the null value otherwise. Finally, `min`, `max`, `avg`, and `sum` denote the well-known SQL aggregate functions.

Definition 3.20 (*Coercion Function Classification*). Let v be a value of type $temporal_H(\tau_2')$ and v' be a value of type $temporal_G(\tau_1')$, such that $\tau_2' \leq_T \tau_1'$ and $H \preceq G$. A coercion function $C : [\![temporal_H(\tau_2')]\!] \rightarrow [\![temporal_G(\tau_1')]\!]$ can be classified as follows.

- Let $\{i_1, \ldots, i_n\}$ be the set of indexes such that $H(i_k) \subseteq G(j)$ and $v(i_k) = v_k$, $\forall k$, $k \in [1, n]$, then C is a *selective* coercion function if $C(v) = v'$ where $v'(j) = v_{\bar{k}}$, $\bar{k} \in [1, n]$.
- Let $\{i_1, \ldots, i_n\}$ be the set of indexes such that $H(i_k) \subseteq G(j)$ and $v(i_k) = v_k$, $\forall k$, $k \in [1, n]$, then C is an *aggregate* coercion function if $C(v) = v'$ where $v'(j) = \bar{v}$ and $\bar{v}$ is computed as function of values $\{v_1, \ldots, v_n\}$.
- A *user-defined* coercion function C is such that $C(v) = v'$ where v' is computed by a user-defined method. □

Intuitively, in case of selective coercion functions, one of the possible values among $\{v_1, \ldots, v_n\}$ is chosen for a generic granule j. In case of aggregate coercion functions, an aggregate function, such as the average or the sum, is applied to the values $\{v_1, \ldots, v_n\}$ to compute the value of granule j.[13] In case of user-defined coercion functions the method to convert from one granularity to the other is completely left to the user.

Note that temporal values are partial functions, thus given v a temporal value, there may exist $i \in \mathcal{IS}$ such that $v(i) = \perp$. In computing aggregate coercion functions we consider such *unknown* information as *null* values in OQL.

Example 3.13 Let $\tau_2 = temporal_{months}(\texttt{short})$ such that $\{\langle i_{02/1998}, 4\rangle, \langle i_{04/1998}, 5\rangle, \langle i_{09/1998}, 3\rangle\} \in [\![\tau_2]\!]$ where $i_{02/1998}$ is the index corresponding to the granule 02/1998 and so on. Moreover, let $\tau_1 = temporal_{years}(\texttt{short})$. An example of aggregate coercion function which maps values of type τ_2 into values of type τ_1 is $\{\langle i_{1998}, 4\rangle\} = \texttt{avg}(\{\langle i_{02/1998}, 4\rangle, \langle i_{04/1998}, 5\rangle, \langle i_{09/1998}, 3\rangle\})$. Moreover an example of selective coercion function which maps values of type τ_2 into values of type τ_1 is $\{\langle i_{1998}, 3\rangle\} = \texttt{last}(\{\langle i_{02/1998}, 4\rangle, \langle i_{04/1998}, 5\rangle, \langle i_{09/1998}, 3\rangle\})$. ◊

Let $<_T$ be the anti-reflexive subtyping relationship, that is, given two types $\tau_1, \tau_2 \in \mathcal{T}_{temp}$, $\tau_2 <_T \tau_1$ if $\tau_2 \leq_T \tau_1$ and $\tau_2 \neq \tau_1$.

In order to ensure substitutability some restrictions are imposed on the EXTENDS relationship, formalized by the following property refinement rule.

[13] Obviously these functions apply in case of set of values for which such functions are defined, such as, for example integers.

Rule 3.4 *Let c_1 and c_2 be two classes such that $c_2 <_T c_1$.*[14] *Moreover, let p be a property of class c_1 whose domain is type τ_1 in c_1. Then, p can be refined in c_2 by a property of type τ_2 if and only if one of the following conditions holds.*

1. $\tau_2 \leq_T \tau_1$;
2. $\tau_1 = temporal_G(\tau_1')$, $\tau_2 = temporal_H(\tau_2')$, $\tau_2' \leq_T \tau_1'$, $G \preceq H$, *and for each* $i \in \mathcal{IS}$, *such that* $H(i) \neq \emptyset$, $j \in \mathcal{IS}$ *exists, such that* $G(j) \subseteq H(i)$;
3. $\tau_1 = temporal_G(\tau_1')$, $\tau_2 = temporal_H(\tau_2')$, $\tau_2' \leq_T \tau_1'$, $H \preceq G$ *and a coercion function* $C : [\![temporal_H(\tau_2')]\!] \rightarrow [\![temporal_G(\tau_1')]\!]$ *is associated to property p in c_2.*

Note that once a property is refined in a subclass only its type can vary. The kind of the property remains the same. That is, if the property is static remains static, if the property is mutable remains mutable, and, finally, if the property is immutable remains immutable. We leave for future work the investigation of changing the kind of a property in refining it.

The following proposition proves that Rule 3.4 ensures substitutability.

Proposition 3.3 *Given an EXTENDS relationship, if Rule 3.4 holds, then type substitutability is ensured.* ○

Proof 3.3 Rule 3.4 holds for each pair of types $\tau_1, \tau_2 \in \mathcal{T}_{temp}$. In order to focus on temporal types, we do not report in this context the proof that Rule 3.4 ensures substitutability for static types. We refer the interested reader to [AC96] for a formal treatment of this case.

In case of temporal types, $\tau_1, \tau_2 \in \mathcal{TT}$, if condition 1 holds, τ_2 is a subtype of τ_1. Then, because of Proposition 3.2, $[\![\tau_2]\!] \subseteq [\![\tau_1]\!]$. Thus, the values of property p of instances of c_2 are legal values for property p of instances of c_1, and substitutability is ensured.

If condition 2 holds, since $G \preceq H$ for each index $i \in \mathcal{IS}$, there exists an index $j \in \mathcal{IS}$ such that $G(i) \subseteq H(j)$. Thus, since we assume the downward hereditary property, once the value of property p is accessed in c_1 with respect to the ith granule of G the value of p in c_2 in the jth granule of H is returned. We denote such value as $p^G(i)$. Note that $p^G(i) \in \bigcup_{i \in \mathcal{IS}} [\![\tau_2']\!]_i^H$, then, we have to prove that $\tau_2' \leq_T \tau_1'$ implies $p^G(i) \in \bigcup_{i \in \mathcal{IS}} [\![\tau_1']\!]_i^G$. This is equivalent to prove that $\bigcup_{i \in \mathcal{IS}} [\![\tau_2']\!]_i^H \subseteq \bigcup_{i \in \mathcal{IS}} [\![\tau_1']\!]_i^G$. To prove this we prove that for each i j exists such that $[\![\tau_2']\!]_i^H \subseteq [\![\tau_1']\!]_j^G$. According to Definition 3.11 $[\![\tau_2']\!]_i^H = \bigcap_{t \in H(i)} [\![\tau_2']\!]_t$. Since, according to the second condition for each $i \in \mathcal{IS}$, such that $H(i) \neq \emptyset$, $j \in \mathcal{IS}$ exists, such that $G(j) \subseteq H(i)$, we have that $\bigcap_{t \in H(i)} [\![\tau_2']\!]_t \subseteq \bigcap_{t \in G(j)} [\![\tau_2']\!]_t$. Then since $\tau_2' \leq_T \tau_1'$, $\bigcap_{t \in G(j)} [\![\tau_2']\!]_t \subseteq \bigcap_{t \in G(j)} [\![\tau_1']\!]_t$. According to Definition 3.11 again we have

[14]This is equivalent to $c_1 = EXTENDS(c_2)$.

that $\bigcap_{t \in G(j)} [\![\tau_1']\!]_t = [\![\tau_1']\!]_i^G$. Thus we have proved that for each i j exists such that $[\![\tau_2']\!]_i^H \subseteq [\![\tau_1']\!]_j^G$.

If condition 3 holds, for each value $v \in [\![\tau_2]\!]$ in class c_2, a corresponding value $C(v) \in [\![\tau_1]\!]$ exists, where C is the coercion function associated to the property. Such value is returned once value of property p in c_2 are accessed as values of property p in c_1. Thus, the coercion function ensures substitutability of instances of type c_2 with respect to instances of type c_1. ○

In what follows we present an example of an inheritance hierarchy including several temporal granularities.

Example 3.14 Consider the class `Course` of Example 3.8. Courses are partitioned into three types: introductory courses (`I_Course`), experimental courses (`E_Course`) and theoretical courses (`T_Course`). Each class representing a course type is a subclass of the class `Course`, that is, $EXTENDS($`I_Course`$) = EXTENDS($`E_Course`$) = EXTENDS($`T_Course`$) =$ `Course`. Each class inherits all the properties of the class `Course` and redefines some of them according to Rule 3.4. Moreover, they introduce proper attributes and relationships.

The class definitions are presented in Figure 3.3, whereas the corresponding class hierarchy is depicted in Figure 3.4. In Figure 3.3 only redefined or proper attributes and relationships are reported in subclasses.

Let us discuss each class separately. Class `Course` has been already discussed in Example 3.8. As far as introductory courses are concerned, the teaching assistant is the same for the whole year, thus `I_Course` redefines the relationship `T_assistant` whose type is now `temporal`$_{years}$`(Researcher)`. Type substitutability is obviously ensured since for the year the teaching assistant is the same for each semester. Moreover, for internal statistics the students participating to each daily class must be stored. Since *days* is finer than *years* a coercion function must be defined for the refined relationship in class `I_Course`, here the user-defined coercion function `count_students` is defined. A possible implementation for `count_students` could be a method computing the percentage of days in which the student has participated to the course, for each student in the union of the value of `P_students` over the days of the year. If this percentage is greater than 50%, the student is returned in the set given as result. Similarly, the class `E_Course` redefines the `T_assistant` relationship with a granularity finer than the one in the superclass ($months \preceq semesters$). Thus, the coercion function `main` is associated with the relationship. Such a function for each semester returns the researcher who did most of lectures for the course. Class `T_Course` refines the attribute `room` of class `Course` with a finer granularity attribute ($months \preceq years$). In this case the associated coercion function is `all` which returns a room number if the room is the same for a whole year, null otherwise.

```
class Course (extent courses, key ((C_code),absolute)),((name),relative)) {
          immutable attribute temporal_{years}(string) C_code;
          attribute string name;
          attribute short year;
          attribute temporal_{years}(short) room;
          relationship temporal_{years}(Professor) T_professor
                     inverse Professor::teaches;
          relationship temporal_{semesters}(Researcher) T_assistant
                     inverse Researcher::assists;
          relationship temporal_{years}(set<Student>) P_students
                     inverse Student::attends;
          }

class I_Course extends Course {
          ref relationship temporal_{years}(Researcher) T_assistant
                     inverse Researcher::assists;
          ref relationship temporal_{days}(set<Student>) P_students
                     inverse Student::attends count_students;
          attribute temporal_{weeks}(string) lab;
          }

class E_Course extends Course {
          ref relationship temporal_{months}(Researcher) T_assistant
                     inverse Researcher::assists main;
          attribute temporal_{days}(Person) expert;
          attribute temporal_{months}(string) lab;
          }

class T_Course extends Course {
          ref attribute temporal_{months}(short) room all;
          }
```

Figure 3.3: Example of database schema (I)

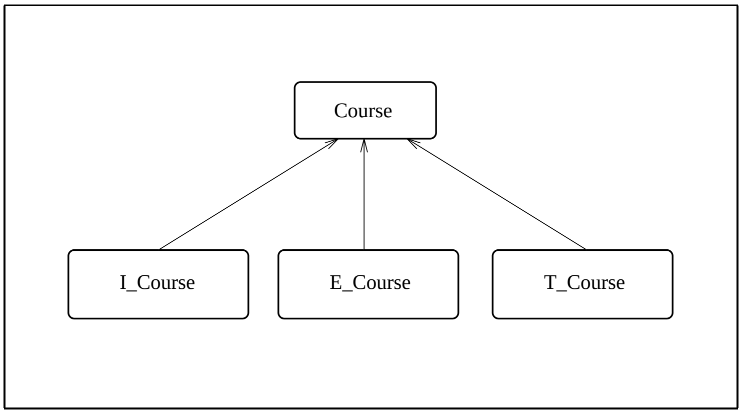

Figure 3.4: Course inheritance hierarchy

Finally, the classes `I_Course` and `E_Course` introduce some proper attributes. For introductory courses a laboratory (`lab`) is needed and it can change each week. Experimental courses have a laboratory as well and it is the same for each month. Moreover an expert can give daily seminars to students. $\diamond$

3.7 T_ODMG Schema and Database

In this section we investigate how temporal types impact on the notions of schema and database formalized in the previous chapter (see Section 2.8).

A T_ODMG database schema is an ODMG database schema as defined in Definition 2.19 where temporal properties can be defined through temporal types. In addition, with respect to an ODMG database, in a T_ODMG database interfaces do not appear, since we have restricted ourselves to object types defined through class declarations, and property domain refinement is allowed. Note that this implies that Rule 2.5, preventing property refinement in subclasses, has not to be satisfied by a T_ODMG schema. Such rule is replaced in T_ODMG by Rule 3.4 which establishes under which restrictions property refinement can be performed. In addition Rule 3.1 has to be satisfied.

Example 3.15 The classes in Figure 3.3 whose inheritance relationship is graphically depicted in Figure 3.4, together with the classes in Figure 3.5, whose inheritance relationship is graphically depicted in Figure 3.6 are an example of database schema. $\diamond$

Before introducing the notion of T_ODMG database, we first formalize the key constraints informally introduced in Section 3.4. In what follows, given an object o and a property p, $o.p$ denotes the value of property p of the object o.

```
class Person (extent Persons, key ((P_code), absolute)) {
            immutable attribute temporal_{G_I}(string) P_code;
            attribute string name;
            attribute temporal_{months}(string) address;
            };

class UnivEmployee extends Person
            (extent UnivEmployees, key ((emp#), absolute) ((room), relative)) {
            attribute short emp#;
            attribute temporal_{months}(short) salary;
            };

class Student extends Person (extent Students) {
            ref attribute temporal_{years}(string) address;
            attribute temporal_{years}(float) avg_grade;
            relationship temporal_{years}(set<Course>) attends
                        inverse Course::P_students;
            relationship temporal_{years}(Professor) supervisor
                        inverse Professor::supervises;
            };

class Professor extends UnivEmployee (extent Professors) {
            attribute temporal_{years}(string) business_hours;
            relationship temporal_{years}(Course) teaches
                        inverse Course::T_professor;
            relationship temporal_{years}(set<Student>) supervises
                        inverse Student::supervisor;
            };

class Researcher extends UnivEmployee (extent Researchers) {
            ref attribute temporal_{months}(short) room main;
            attribute temporal_{weeks}(string) supervised_lab;
            relationship temporal_{months}(set<Course>) assists
                        inverse Course::T_assistant;
            };
```

Figure 3.5: Example of database schema (II)

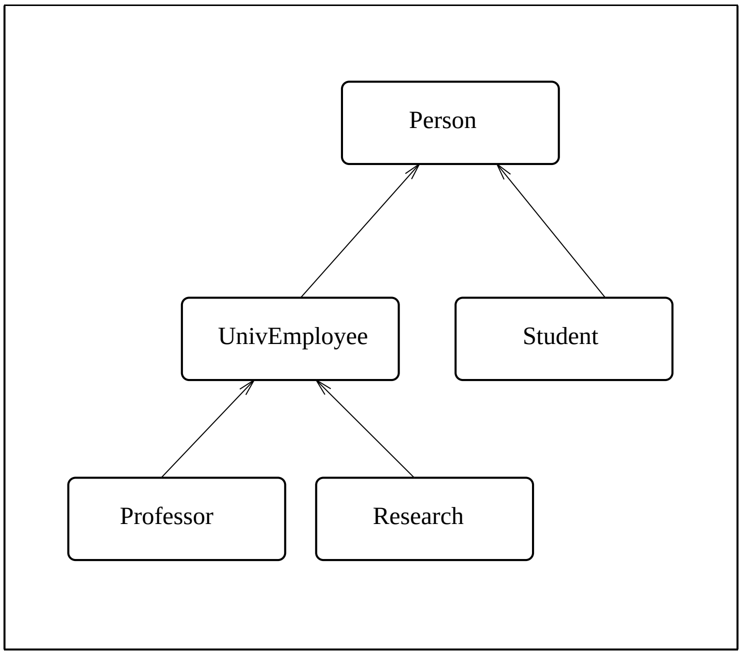

Figure 3.6: Person inheritance hierarchy

Rule 3.5 *Let c be a class and $p_1, \ldots, p_n$ be a list of property names of c, defined as a key for that class, such that $\tau_1, \ldots, \tau_n$ are the types of $p_1, \ldots, p_n$ in c, respectively.*

- *If the key is static then the following conditions must hold:*
  - $\tau_1, \ldots, \tau_n$ *are static types, that is,* $\tau_1, \ldots, \tau_n \in \mathcal{ST}$, *and*
  - *for each pair of objects* o, $o' \in c.history(i_{now})$, k *exists,* $k \in [1, n]$, *such that* $o.p_k \neq o'.p_k$.
- *If the key is relative then the following conditions must hold:*
  - $\tau_1, \ldots, \tau_n$ *are temporal types, that is,* $\tau_1, \ldots, \tau_n \in \mathcal{TT}$,
  - *let* $\tau_1 = temporal_{G_1}(\tau'_1), \ldots, \tau_n = temporal_{G_n}(\tau'_n)$, *then a granularity* $G' \in \mathcal{G}$ *exists such that* $G' \preceq G_i$ $\forall i$, $i \in [1, n]$, *and*
  - *for each pair of objects* o, $o' \in c.history(t)$, $t \in \mathbb{N}$, *and for each* $i \in \mathcal{IS}$, k *exists,* $k \in [1, n]$, *such that* $o.p_k^{G'}(i) \neq o'.p_k^{G'}(i)$.[15]

[15] We recall that, given a property p of type $temporal_G(\tau)$, a granularity $H \preceq G$, and an index $j \in \mathcal{IS}$ such that $H(j) \subseteq G(i)$, $p^H(j)$ is the value of the property p with respect to the jth granule of H and such a value is equal to $p(i)$.

- *If the key is absolute then the following conditions must hold:*
  - $\tau_1, \ldots, \tau_n$ *are temporal types, that is,* $\tau_1, \ldots, \tau_n \in \mathcal{TT}$,
  - *let* $\tau_1 = temporal_{G_1}(\tau'_1), \ldots, \tau_n = temporal_{G_n}(\tau'_n)$, *then a granularity* $G' \in \mathcal{G}$ *exists such that* $G' \preceq G_i$ $\forall i,\ i \in [1, n]$, *and*
  - *for each pair of objects* $o,\ o' \in \bigcup_{t \in \mathbb{N}} c.history(t)$ *and for each* $i, j \in \mathcal{IS}$, k *exists,* $k \in [1, n]$, *such that* $o.p_k^{G'}(i) \neq o'.p_k^{G'}(j)$. □

Then, we extend Definition 2.20, which establishes when a set of objects is coherent with respect to static relationships, to temporal relationship.

Definition 3.21 (*Set of T_ODMG Objects Satisfying the Coherence of Relationships*).
Let $S = (S_C, EXTENDS)$ be a T_ODMG database schema. A set of objects OBJ satisfies the coherence of relationships in S if OBJ satisfies the conditions established in Definition 2.20, and for each temporal relationship r_j, $j \in [1, n]$, of each object $o \in OBJ$, $o.r = (r_1 : v_1, \ldots, r_n : v_n)$, then:

- $(temporal_G(\tau_j), r_j, (\tau_j^{inv}, r_j^{inv})) \in o.c.rel$, $\tau_j \in Id(S_C)$,[16] and
  - if the type of r_j^{inv} in τ_j^{inv}[17] is $temporal_H(\tau_k)$, then the following conditions must hold:
    - ∗ a granularity $G' \in \mathcal{G}$ exists such that $G' \preceq G$ and $G' \preceq H$, and
    - ∗ for each $i \in \mathcal{IS}$ such that $v_j^{G'}(i) \neq \perp$, an object $o' \in OBJ$ exists such that $v_j^{G'}(i) = o'.id$, and let $o'.r = (r'_1 : v'_1, \ldots, r'_m : v'_m)$, then $k \in [1, m]$ exists such that $r'_k = r_j^{inv}$ and $o.id = {v'}_k^{G'}(i)$ or $o.id \in {v'}_k^{G'}(i)$;
  - if the type of r_j^{inv} in τ_j^{inv} is static, then the following condition must hold:
    - ∗ for each $i \in \mathcal{IS}$ such that $now \in G(i)$ and $v_j(i) \neq \perp$, an object $o' \in OBJ$ exists such that $v_j(i) = o'.id$;
- $(temporal_G(coll < \tau_j >), r_j, (\tau_j^{inv}, r_j^{inv})) \in o.c.rel$, $\tau_j \in Id(S_C)$, $coll \in \{$`set`, `list`, `bag`$\}$, then the following conditions must hold:
  - if the type of r_j^{inv} in τ_j^{inv}[18] is $temporal_H(\tau_k)$, then the following conditions must hold:

[16] We recall that $Id(S_C)$ denotes the set of object type identifiers of the object type external specifications in S_C.
[17] Note that in $\tau_j = \tau_j^{inv}$.
[18] Note that $\tau_j = \tau_j^{inv}$.

* a granularity $G' \in \mathcal{G}$ exists such that $G' \preceq G$ and $G' \preceq H$, and
* for each $i \in \mathcal{IS}$ such that $v_j^{G'}(i) \neq \perp$ and an object $o' \in OBJ$ exists such that $o'.id \in v_j^{G'}(i)$, and let $o'.r = (r'_1 : v'_1, \ldots, r'_m : v'_m)$, then $k \in [1, m]$ exists such that $r'_k = r_j^{inv}$ and $o.id = {v'}_k^{G'}(i)$ or $o.id \in {v'}_k^{G'}(i)$;

– if the type of r_j^{inv} in τ_j^{inv} is static then the following condition must hold:

* for each $i \in \mathcal{IS}$ such that $now \in G(i)$ and $v_j(i) \neq \perp$, an object $o' \in OBJ$ exists such that $o'.id \in v_j(i)$.

□

We recall that, similarly to an ODMG database, a T_ODMG database has to satisfy two other important properties: oid-uniqueness and referential integrity.

The notion of T_ODMG database differs from the one of ODMG database of Definition 2.21 with many respects, thus, we have decided to report it here. This is in accordance with the intuition, since in T_ODMG the schema is the time invariant component, whereas the database is the time varying one.

Definition 3.22 *(T_ODMG Database).* Let $S = (S_C, EXTENDS)$ be a T_ODMG database schema. A T_ODMG database over S is a pair (OBJ, π_{temp}), where:

1. for each pair of objects o_1 and $o_2 \in OBJ$: if $o_1.id = o_2.id$, then $o_1.N = o_2.N$, $o_1.v = o_2.v$, $o_1.r = o_2.r$, $o_1.c = o_2.c$, $o_1.lifespan = o_2.lifespan$;

2. each object $o \in OBJ$ is a consistent instance of $o.c$ according to Definition 3.17 and $o.c \in Id(S_C)$;

3. Rule 3.5 on key constraint is satisfied;

4. OBJ satisfies the coherence of relationships (cf. Definition 3.21);

5. each object $o \in OBJ$ satisfies Rule 3.2 with respect to its class $o.c$ for each time instant $t \in \mathbb{N}$ such that $\{t\} \in o.lifespan$.

□

A T_ODMG database state is modeled similarly to an ODMG database state (cf. Definition 2.22). That is, a state S of a T_ODMG database (cf. Definition 3.22) is a pair (π^+_{temp}, ν) where π^+_{temp} is the function defined in Definition 3.9 and function ν applied to an object oid returns the state of the object identified by the oid, that is, the value of its properties.

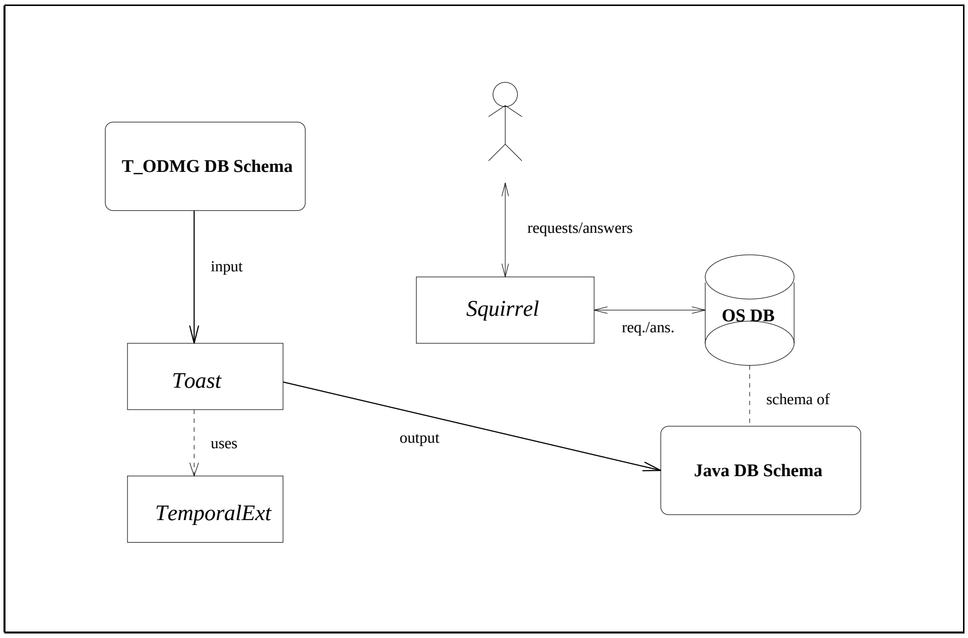

Figure 3.7: System architecture

3.8 Implementation Issues

The model presented in this thesis is being currently implemented on top of ObjectStore PSE Pro [Des98], an ODMG compliant DBMS. ObjectStore supports both C++ and Java object-oriented programming languages. We have chosen Java [AG98] for our prototype, since Java is establishing itself as one of the most successful and promising object-oriented languages. In addition, the ODMG object model is closer to the Java one than to the C++ one, and it seems to go more and more in that direction.

The implementation is composed by three components:

1. *Toast.java* (*T_ODMG to JAva Simple Translator*), which takes as input a text file containing the definition of a database schema including temporal types, as it has been defined in Section 3.7 and produces as output the Java translation of the database schema;

2. *Squirrel.java*, which implements the graphical user interface;

3. *TemporalExt*, a Java package, which is a Java library implementing multiple granularity temporal data.

Because of the application *Squirrel.java*, the management of temporal values and temporal granularities is completely transparent to the user, which does not see any difference between temporal and static properties. The architecture of our prototype is presented in Figure 3.7.

The application *Squirrel.java* is general, that is, it works on every set of Java classes translated by *Toast.java* from a T_ODMG schema. This is achieved thanks to a heavy use of Java *reflection*, that is, the capability of the language of reifying the definition of classes in order to invoke their methods.

In this section we do not go into details on the interface implementation, rather we focus on how three very important features of our model are implemented: temporal granularities, temporal attributes, and attribute refinement. The current version of the implementation does not support relationships, neither static nor temporal. The reason is that neither ObjectStore, nor Java support relationships. In the future we plan to extend the implementation in order to support relationships.

The remainder of this section is organized as follows. In Subsection 3.8.1 the implementation of temporal granularities is presented. In Subsection 3.8.2 the implementation of temporal properties is introduced. In Subsection 3.8.3 the implementation of attribute refinement is presented. Finally, in Subsection 3.8.4 the functionalities of the application and extensions to the implementation are presented.

3.8.1 Temporal Granularities

The main idea on which our implementation of temporal granularities is based is that, since a granularity is a mapping, we do not need to create new objects for each granule, rather we need to "implement" such a mapping through methods. Thus, for each granularity two methods are defined:

- *indexToInterval*, which takes as input an index $i \in \mathcal{IS}$ and returns the corresponding time interval on the time domain; this method corresponds to function G;
- *labelToIndex*, which takes as input the textual representation of a granule $G(i)$, i.e., the label, and returns the corresponding index i.

Note that method *labelToIndex* can be regarded, from an intuitive point of view, as the reverse operation of method *indexToInterval*. However, it does not implement the *inverse* function according to the mathematical meaning of inverse. In fact, method *indexToInterval* returns time intervals, whereas method *labelToIndex* takes as input labels.

In addition, since our model is based on the finer than relationship, we need to store information related to this relationship. Thus, for each granularity two attributes are defined:

- *finer*, storing the set of granularities finer than the considered one;
- *coarser*, storing the set of granularities coarser than the considered one.

Temporal granularities are classes without instances, which corresponds in Java to *abstract classes*. Each abstract class implementing a temporal granularity is an implementation of the following common interface.

```
interface granularity {
  public static String[] get_finer();
  public static String[] get_coarser();
  public static Interval indexToInterval(Long i);
  public static long convert_to_index(String label);}
```

Methods *get_finer* and *get_coarser* are the accessor methods corresponding to the attributes *finer* and *coarser*, respectively. Note that for each attribute a set of strings is stored, more precisely an array, which represents the set of granularity names of the abstract classes representing the granularities finer or coarser than the considered one.

Each abstract class representing a granularity is stored in a Java file, named *granularity_name.java* (e.g., *months.java*), which is manipulated by the application *Squirrel.java*. In the current prototype, the implemented granularities are the ones of the gregorian calendar. Note that the approach we have used to implement temporal granularities is very flexible since it allows the user to define his/her own granularities. In fact, to insert a new granularity the user has to define the proper abstract class having the common interface *granularity* described above. In this way the set of granularities can be regarded as an on-line library which can be updated by the user. The interface manipulates the granularity classes through the Java reflective features.

3.8.2 Implementation of Temporal Attributes

For each definition of a temporal attribute a in the input schema, a corresponding Java attribute with the same name and with the class *temporal_attribute* as domain is created. The definition of the class *temporal_attribute* is the following:

```
class temporal_attribute {
  private String att_type;
  private String att_gran;
  private list_temporal att_value;}
```

Let $temporal_G(\tau)$ be the type of an attribute a, then each component of class *temporal_attribute* associated to a has the following meaning:

- *att_type* stores the inner type τ;
- *att_gran* stores the granularity G;
- *att_value* stores the attribute temporal value, which is a value in $[\![\, temporal_G(\tau) \,]\!]$.

Note that the attributes *att_type* and *att_gran* storing the granularity and the inner type of the temporal attribute a are of type *String*. Thus, Java reflective features are used to invoke methods of the class whose name is the one stored in the type or granularity component of the temporal attribute.

In the current prototype, the management of temporal attributes is very simple. The application builds the list representing a temporal value of type $temporal_G(\tau)$, that is, a partial function from granules of G to values of type τ. The definition of class *list_temporal* is the following:

```
class list_temporal {
  private Object value;
  private interval value_interval;
  private list_temporal next;}
```

Note that we do not create objects representing granules, rather we store for each granule the interval it represents in the time domain, which corresponds to Java milliseconds. Such information is stored in the attribute *value_interval*. We do not report here the definition of class *interval*, which represents intervals of the time domain, since it simply consists of the two bounds of the interval.

Consider, for instance, a temporal attribute of type $temporal_{years}(\texttt{string})$, for each granule of granularity *years* we store the interval of the time domain starting from the first instant of January, 1st to the last instant of December, 31st of that year. In addition, if a value is constant for some consecutive granules, then a single value is stored in the list, whose corresponding temporal interval is the union of the consecutive granules.

Figure 3.8 illustrates an example of the management of the insertion of values in a list of type $temporal_{years}(\texttt{string})$ storing, for instance, the city where a person lives.

Note that lists are stored with the most recent value in the head, in order to get them quickly. It is part of our future work to provide an efficient management of temporal values through allocations of suitable indexes.

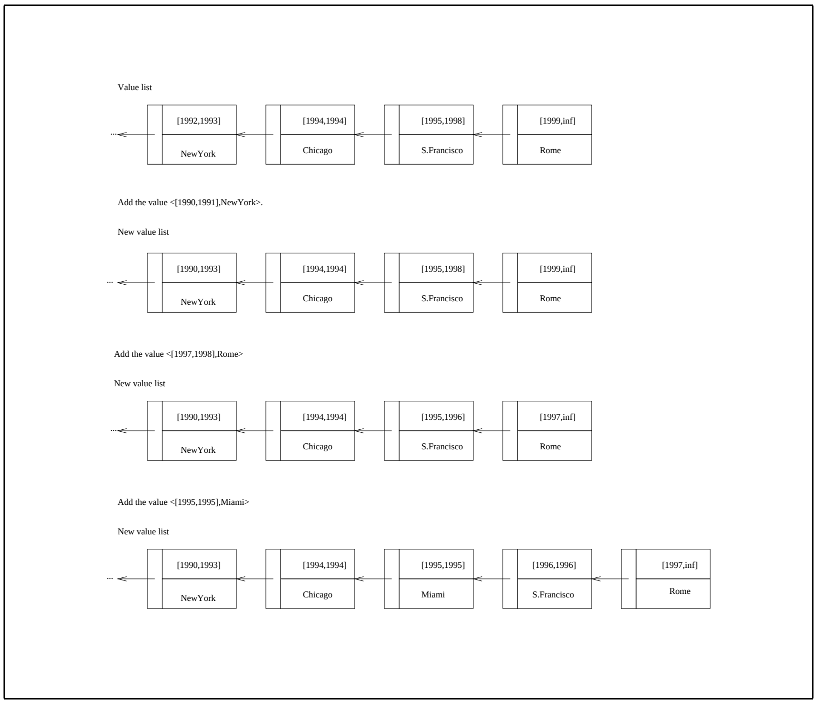

Figure 3.8: Example of list management

3.8.3 Attribute Refinement

Java does not support attribute refinement. To redeclare attributes in subclasses it supports a *hiding* mechanism. Through such mechanism, an attribute with the same name and a different type, can be declared in a subclass, however, both the two attributes are kept and can be accessed.[19] Thus, attribute refinement is not performed.

Attribute refinement in T_ODMG has been implemented through two additional classes:

- *Redefinition.java* which implements the data structure storing the information related to attribute refinement;
- *CFunction.java* which implements the pre-defined coercion functions which have been presented in Figure 3.2.

In the current implementation, temporal attribute refinement is performed leaving the inner type unaltered. Thus, with respect to the second and the third item of Rule 3.4, $\tau'_2 = \tau'_1$. A refined attribute can be characterized by three components: the attribute

[19]This is performed through the use of *super*.

current granularity, the granularity of the attribute in the superclass, and, if needed, the associated coercion function. Thus, the structure of class *Redefinition* is as follows:

```
class Redefinition {
  private String g;
  private String gSuper;
  private String cf;}
```

where g is the attribute current granularity, *gSuper* is the granularity of the attribute in the superclass, and *cf* is the associated coercion function. In addition, class *Redefinition* has several methods which are used by the methods of the class in which the attribute has been refined to implement substitutability. The main method of this class is *convert* which converts the value of an attribute related to a granularity G into a value of a granularity H, that is, the one in the superclass. If $H \prec G$ then such conversion is simply performed using the downward hereditary property. If $G \prec H$ then the associated coercion function is used.

Class *CFunction* implements the pre-defined coercion functions which have been presented in Figure 3.2. Class *CFunction* is presented in what follows:

```
class CFunction {
  public static Object sum(Object[][] v){...};
  public static Object avg(Object[][] v){...};
  public static Object min(Object[][] v){...};
  public static Object max(Object[][] v){...};
  ...
  }
```

Each coercion function is a class (static) method which takes as input the list representing the value of a temporal attribute and returns as output the coerced value.

3.8.4 Supported Functions

The program *Squirrel.java* is a general interface which works on every T_ODMG schema which has been translated into java by the application *Toast.java*. It supports the following functions:

1. creation of a new object;
2. deletion of an existing object;

3. set of the current object;

4. update of the attributes of the current object;

5. update of the lifespan of the current object;

6. display of the attribute values of the current object;

7. display of the value of a temporal attribute of the current object.

Among the previous functions, the latter two, that is, the ones concerning the display of attribute values, are examples of very simple queries. The theoretical foundations of the query language will be presented in Chapter 4. To date, the extension of the presented implementation to querying objects is under development and we have achieved a good point. However, since this is not yet accomplished, we have preferred not to include it in the thesis.

In the current prototype, the visual interface of the system is very primitive, its improvement is currently under designing phase. We plan to use Java Swing libraries in order to work in a complete integrated framework, thanks to the Java programming language.

Summarizing, the improvement of the system that are currently under implementations are the support of the query language facilities that will be presented in next chapter, the extension to relationships, efficiency issues, and, finally, the visual interface.

3.9 Concluding Remarks

In handling temporal information, the need often arises of managing temporal data expressed according to different granularities. Those data should in many cases be compared and used by the same application program. A temporal data model should thus offer the possibility of handling multiple temporal granularities. In this chapter, we have formally defined T_ODMG, a temporal ODMG compliant data model supporting multiple granularities, and we have revisited on a formal basis issues related to inheritance, type refinement, and substitutability. We have described how the proposed model is being implemented.

Chapter 4

Querying Temporal Data at Multiple Granularities

In this chapter we present how temporal data expressed with respect to different granularities can be queried. Querying object data in OQL is based on two main notions: combining and comparing values of properties and navigating through objects. In T_ODMG the values of properties can be temporal values expressed with respect to different granularities. Thus, first we have to establish how such values can be combined [MBF$^+$00]. In OQL, navigation through objects is performed via path expressions [BNPS92]. The extension of path expressions to a temporal model has already been investigated in [BFG98]. *Temporal path expressions* [BFG98] are obtained as an extension of classical path expressions of object-oriented languages, in that for each property access a time can be specified, in addition to the property name. Here, the notion of temporal path expression is extended to a multi-granularity context.

An interesting issue in temporal query languages is how the time with respect to which a query is evaluated can be denoted. Generally, in a non-temporal object-oriented database, a query selects objects based on the evaluation of boolean conditions involving property values, method invocation results, and so on. In a temporal context, queries must support the retrieval of objects satisfying a given boolean condition for a specific set of time instants, that can be denoted either explicitly or implicitly. Actually, since in T_ODMG we support multiple temporal granularities, the queries we consider must support the retrieval of objects satisfying a given boolean condition for a specific set of granules. The time with respect to which the query is evaluated can be expressed either *explicitly*, by specifying a granule or a set of granules, or *implicitly*, by means of a symbolic formalism. To symbolically denote a set of time instants we use a formalism based on the notion of *temporal expression*. A temporal expression is an expression that, evaluated on an object (the one through which we are navigating), returns a set of granules.

In this chapter we do not aim at presenting a complete query language, rather what we present can be considered as the *core* of a language which extends OQL to query data expressed with respect to different granularities.

The presented chapter is organized as follows. In Section 4.1 we present the main features of OQL. In Section 4.2 some preliminary notions that will be extensively used throughout this chapter are introduced. In Section 4.3 we discuss how temporal values related to different temporal granularities can be combined. Section 4.4 presents the syntax and semantics of temporal expressions. Section 4.5 presents path expressions in T_ODMG. Finally, Section 4.6 concludes the chapter.

4.1 The ODMG Query Language: OQL

ODMG provides a declarative language for querying objects, OQL, that is simple, not computationally complete, and provides high level primitives to deal with objects. OQL is an SQL-like language, thus, it is based on the `select from where` clause.

Example 4.1 Consider the database schema presented in Example 2.15.[1] The following OQL query retrieves the names of the teaching assistants living in room 511.

```
select t.stud_name
  from Teaching_Assistants t
  where t.dorm_address.room_number="511"
```

◊

We do not report here the entire specification of OQL. We refer the interested reader to [CBB+99] for the complete language specification. By contrast, we report the main features of OQL which makes it a real object-oriented query language.

Even though OQL is very close to SQL, it differs from SQL in many respects. First, the result of a query in SQL is always a relation, whereas in OQL the type of the result of a query depends on the query itself. For instance, in the query presented in Example 4.1 the type of the result is `bag<string>` since the type of the attribute `stud_name` of class `Teaching_Assistant` is `string` and no `distinct` clause has been specified. In case of `distinct` clause the type of the result would be `set<string>`. In addition, OQL allows one to return objects as results of a query, as shown by the following example.

[1] We refer to the database schema presented in Example 2.15 and not the one presented in Example 3.15 since the former is an ODMG schema, whereas the later is a T_ODMG schema. In this section we present OQL queries on an ODMG schema.

Example 4.2 Consider the database schema presented in Example 2.15. Suppose that attribute `year` of class `Teaching_Assistant` stores the university year a student attends. The following OQL query retrieves the `Teaching_Assistant` objects corresponding to students attending at least the third university year.

```
select t
  from Teaching_Assistants t
  where t.year >= 3
```

◊

Finally, OQL offers the possibility to structure the result of queries, returning for instance `struct` types.

The second feature of OQL which differentiates it from SQL is the use of path expressions. Path expressions allow one to navigate through objects via object identifiers. The notation used by OQL is the dot notation and allows one to follow both references contained in attributes and traversal paths corresponding to relationships. We recall that in OQL path navigation across multivalued properties is not allowed. The following example presents some OQL queries involving path expressions.

Example 4.3 Consider the database schema presented in Example 2.15. Suppose that class `Section` is equipped with an attribute `of_course` which stores the course to which the section refers. In addition, suppose that class `Course` is equipped with an attribute `name` which stores the course name. The following OQL query retrieves the names of the teaching assistants of database courses.

```
select t.stud_name
  from Teaching_Assistants t
  where t.assists.of_course.name="Database"
```

The following OQL query retrieves the set of structs composed by the name and the identifier of the teaching assistants which take sections of database courses.

```
select struct(t_name: m.stud_name, t_id: t.student_id)
  from Teaching_Assistants t, t.takes s
  where s.of_course.name="Database"
```

Note that in the previous query each multivalued relationship is traversed via aliases. ◊

Note that path expressions appear in SQL:1999 to support navigation through reference types and through the ADT components. However, in SQL:1999 they are an alternative to the usual join mechanism, whereas in OQL they are the standard mechanism to navigate through references. In OQL, indeed, object oids cannot be explicitly referenced in queries. In addition, in OQL method invocations can appear in path expressions. The notation for calling a method is exactly the same as for accessing an attribute or traversing a relationship, in the case the method has no parameters. If it has parameters, these are given between parentheses. The following example simply illustrates the use of methods in OQL path expressions.

Example 4.4 Suppose that the method `boolean reg_for_course` which class `Teaching_Assistant` inherits from interface `Student` in Example 2.15 returns returns true if the teaching assistant attends that course in that section, false otherwise. The following OQL query retrieves the set of objects of class `Teaching_Assistant` representing students of a database course.

```
select t
  from from Teaching_Assistants t, Course c, Sections s
  where t.reg_for_course(c,s) and c.name="Database"
```

$\diamond$

An important feature of OQL is the support of the late binding mechanism. Thus, for each method invocation the most specific implementation with respect to the type of the object is associated with the method for execution. That is, for instance, suppose that method `reg_for_course` of Example 4.4, is overridden in the class `Senior_Teaching_Assistant` which is subclass of `Teaching_Assistant`, then for the `Teaching_Assistant` objects whose most specific class is `Senior_Teaching_Assistant` the method implementation in class `Senior_Teaching_Assistant` is invoked.

The use of methods in queries opens many relevant issues that are not fully addressed in OQL. For instance, though it is not explicitly stated in the OQL specification, it seems reasonable to allow only side-effect free methods in queries, that is, they should not modify the database state [Clu98]. Such a restriction is reasonable in order to keep queries side-effect free.

4.2 Preliminaries: Temporal Elements

In Section 3.1 we have formally defined the notion of temporal interval (cf. Definition 3.3) with respect to a given granularity as follows. Let $G \in \mathcal{G}$ be a granularity and $i, j \in \mathcal{IS}$ be two indexes such that $i \leq j$. Then $[i, j]^G = \{G(k) \mid i \leq k \leq j, k \in \mathcal{IS}\}$ is called *temporal* *interval*, with respect to granularity G. An alternative way to define a temporal interval is as a set of contiguous granules (cf. Definition 3.4).

In this chapter we will frequently deal with *temporal elements*. The notion of temporal element has been first formally introduced by Gadia in [Gad88]. Intuitively, every subset of the time domain is a temporal element. We extend the notion presented in [Gad88] to a multi-granularity model.

Definition 4.1 (*Temporal Element*). Let $G \in \mathcal{G}$ be a granularity, then every subset of the set of granules associated to G, $\{G(i) \mid i \in \mathcal{IS}\}$, is called temporal element with respect to granularity G. $\square$

Example 4.5 For instance, $\{months(i) \mid i \in \mathcal{IS}$ and $i_{01/1999} \leq i \leq i_{12/1999}\}$ is a temporal element with respect to granularity *months* representing the set of the months of year 1999.

$\diamond$

Every temporal element can be represented as a finite union of intervals. That is, let $[i_1, j_1]^G, \ldots, [i_n, j_n]^G$ be temporal intervals with respect to granularity G. Then, $[i_1, j_1]^G \cup \ldots \cup [i_n, j_n]^G$ represents the temporal element which includes all the granules included in each interval.

Example 4.6 Let $[i_{01/1999}, i_{07/1999}]^{months}$ and $[i_{10/1999}, i_{12/1999}]^{months}$ be two temporal intervals with respect to granularity *months*, then $[i_{01/1999}, i_{07/1999}]^{months} \cup [i_{10/1999}, i_{12/1999}]^{months}$ represents the temporal element $\{months(i) \mid i_{01/1999} \leq i \leq i_{07/1999},\ i_{10/1999} \leq i \leq i_{12/1999}$ and $i \in \mathcal{IS}\}$ with respect to granularity *months*. $\diamond$

Note that the representation of a temporal element as a finite union of intervals is not unique. For instance, the temporal element of the previous example can be represented as $[i_{01/1999}, i_{07/1999}]^{months} \cup [i_{10/1999}, i_{12/1999}]^{months}$ or, similarly, as $[i_{01/1999}, i_{04/1999}]^{months} \cup [i_{05/1999}, i_{07/1999}]^{months} \cup [i_{10/1999}, i_{12/1999}]^{months}$. In what follows we consider always the *minimal* representation of a temporal element as a finite union of intervals. With minimal we mean that contiguous intervals are merged in a unique one. Thus, $[i_{01/1999}, i_{04/1999}]^{months} \cup [i_{05/1999}, i_{07/1999}]^{months} \cup [i_{10/1999}, i_{12/1999}]^{months}$ is not a minimal representation of a temporal element since $[i_{01/1999}, i_{04/1999}]^{months}$ and $[i_{05/1999}, i_{07/1999}]^{months}$ can be merged.

A granule itself is obviously a temporal element. Indeed, the ith granule of a granularity G can be represented through the temporal interval $[i, i]^G$, thus it can be regarded as a temporal element.

In what follows temporal elements will be frequently denoted through the symbol Υ^G where G is the granularity of the granules which compose the temporal element. Note indeed that in a temporal element all the granules are expressed with respect to the same granularity. The set of all temporal elements with respect to a granularity $G \in \mathcal{G}$ is denoted by $\mathcal{TELEM}_G$. Finally, $\mathcal{TELEM}$ denotes the union of all the set of temporal elements with respect to any granularity G, that is, $\mathcal{TELEM} = \bigcup_{G \in \mathcal{G}} \mathcal{TELEM}_G$.

In [Gad88], where temporal elements are subsets of the time domain, it is proved that the set of all temporal elements is closed under union ($\cup$), intersection ($\cap$), difference ($\setminus$), and complementation and thus forms a boolean algebra. The operations union, intersection, and complementation[2] are the counterparts of "or", "and", and "not" in natural languages. Thus, the closure with respect to such operations makes substantial simplifications in expressing temporal queries [Gad88].

Such a result applies similarly to the set of all temporal elements of a given granularity G, $\mathcal{TELEM}_G$. Indeed, let Υ_1^G and Υ_2^G be two temporal elements, since they are expressed with respect to the same granularity G the operations of union ($\Upsilon_1^G \cup \Upsilon_2^G$), intersection ($\Upsilon_1^G \cap \Upsilon_2^G$), difference ($\Upsilon_1^G \setminus \Upsilon_2^G$), and complement ($(\Upsilon^G)^c$)[3] have the usual semantics of set operations when applied to sets of granules at the same granularity. Since we deal with multiple granularities we often need to apply these operations to temporal elements at different granularities. In this case the sets are first *converted* to a common granularity: the greatest lower bound *glb* of their granularities (see Section 3.1).[4] Thus, we formalize now the notion of *conversion* of a temporal interval from a granularity to another, and then we extend such definition to temporal elements.

Definition 4.2 (*Conversion of a Temporal Interval*). Let $H \in \mathcal{G}$ be a granularity and $i, j \in \mathcal{IS}$ be two indexes such that $i \leq j$. Then $G([i,j]^H)$ denotes the conversion of the temporal interval $[i,j]^H$ to granularity G. It is defined as follows: $G([i,j]^H) = [h,k]^G$ such that $\bigcup_{i \leq p \leq j} H(p) \supseteq \bigcup_{h \leq p \leq k} G(p)$ and $\nexists [h',k']^G$ such that $\bigcup_{i \leq p \leq j} H(p) \supseteq \bigcup_{h' \leq p \leq k'} G(p) \supseteq \bigcup_{h \leq p \leq k} G(p)$. □

Note that in the previous definition if $G = H$ obviously $G([i,j]^G) = [i,j]^G$. The conversion can be applied if either $G \preceq H$ or $H \prec G$. In both cases, the set of time instants denoted by the initial interval includes the set of time instants denoted by the converted one. Such a constraint is expressed through the condition $\bigcup_{i \leq p \leq j} H(p) \supseteq \bigcup_{h \leq p \leq k} G(p)$. In Definition 4.2, if $G \preceq H$ we refer to the conversion as *downward conversion*. By contrast, if $H \prec G$ we refer to the conversion as *upward conversion*.

[2] We recall that the difference can be expressed through these operations. Indeed, let A and B two sets, then $A \setminus B = A \cap B^c$.

[3] The universe with respect to which the complement is computed will be clarified in what follows.

[4] We recall that, since we do not consider the empty granularity in our set $\mathcal{G}$, two granularities are not guaranteed to have a *glb* in $\mathcal{G}$. Here and in the remainder of this chapter the approach is that two granularities can be "used together" only if they admit a *glb* in $\mathcal{G}$.

Example 4.7 Let $[i_{1995}, i_{2000}]^{years}$ be e temporal interval with respect to granularity years. Then, $days([i_{1995}, i_{2000}]^{years}) = [i_{01/01/1995}, i_{31/12/2000}]^{days}$ is an example of downward conversion, whereas $years([i_{01/01/1995}, i_{31/12/2000}]^{days}) = [i_{1995}, i_{2000}]^{years}$ is an example of upward conversion. Another example of upward conversion is $years([i_{01/1999}, i_{05/2000}]^{months}) = [i_{1999}, i_{1999}]^{years}$. In this case the set of instants included in the converted interval is strictly included in the set of instants represented by the initial interval. $\diamond$

The conversion of a temporal element Υ^H is computed by repeatedly applying the conversion of each temporal interval composing the temporal element.

Definition 4.3 (*Conversion of a Temporal Element*). Let $\Upsilon^H = [i_1, j_1]^H \cup \ldots \cup [i_n, j_n]^H$ be a temporal element. Then $G(\Upsilon^H)$ denotes the conversion of the temporal element Υ^H to granularity G. It is defined as $G(\Upsilon^H) = G([i_1, j_1]^H) \cup \ldots \cup G([i_n, j_n]^H)$. $\square$

In what follows given a set of granularities $\{G_1, \ldots, G_n\}$, $glb(G_1, \ldots, G_n)$ denotes the granularity which is the greatest lower bound of $G_1, \ldots, G_n$ in $\mathcal{G}$ with respect to the finer that relationship. Similarly, $lub(G_1, \ldots, G_n)$ denotes the granularity which is the lowest upper bound of $G_1, \ldots, G_n$ in $\mathcal{G}$ with respect to the finer that relationship. The following example presents how the conversion of temporal elements is used in operations between temporal elements.

Example 4.8 Consider the following temporal elements: $[i_{1995}, i_{2000}]^{years}$ and $[i_{01/1996}, i_{03/1996}]^{months} \cup [i_{08/1999}, i_{07/2000}]^{months}$. If we want to compute

$$[i_{1995}, i_{2000}]^{years} \setminus ([i_{01/1996}, i_{03/1996}]^{months} \cup [i_{08/1999}, i_{07/2000}]^{months})$$

then, first the temporal element with respect to the coarser granularity is converted to the finer one, that is, $months([i_{1995}, i_{2000}]^{years}) = [i_{01/1995}, i_{12/2000}]^{months}$. Then the operation is computed:

$$[i_{01/1995}, i_{12/2000}]^{months} \setminus ([i_{01/1996}, i_{03/1996}]^{months} \cup [i_{08/1999}, i_{07/2000}]^{months}) =$$
$$[i_{01/1995}, i_{12/1995}]^{months} \cup [i_{04/1996}, i_{07/1999}]^{months} \cup [i_{08/2000}, i_{12/2000}]^{months}.$$

Note that in this case $glb(years, months) = months$.

$\diamond$

We introduce now some notations that will be used in what follows. Given a generic set S, $\mid S \mid$ denotes the cardinality of S.

Given an interval $[i, j]^G$, $min([i, j]^G)$ denotes the lower bound i, whereas $max([i, j]^G)$ denotes the upper bound j.

In addition, we define a projection operation $\Pi(\Upsilon^G, n)$, that takes as input a temporal element Υ^G and a natural number n. Function Π orders the elements in Υ^G in increasing order, with respect to their upper bound, and returns the n-th interval in the ordering. If the cardinality of Υ^G is lesser than n, $\Pi(\Upsilon^G, n)$ is undefined.

Example 4.9 Let $[i_{01/1995}, i_{04/1995}]^{months} \cup [i_{04/1996}, i_{07/1997}]^{months} \cup [i_{09/1999}, i_{03/2000}]^{months}$ be a temporal element with respect to granularity *months*. Then,

$\Pi([i_{01/1995}, i_{04/1995}]^{months} \cup [i_{04/1996}, i_{07/1997}]^{months} \cup [i_{09/1999}, i_{03/2000}]^{months}, 2) =$
$[i_{04/1996}, i_{07/1997}]^{months}$,

whereas

$\Pi([i_{01/1995}, i_{04/1995}]^{months} \cup [i_{04/1996}, i_{07/1997}]^{months} \cup [i_{09/1999}, i_{03/2000}]^{months}, 4) = \perp$. $\diamond$

We recall that Table A.1 (cf. Appendix A) summarizes the meaning of the symbols most frequently used in the thesis.

In a temporal context, queries must support the retrieval of objects satisfying a given boolean condition (or a set of boolean conditions) for a specific set of time instants, that can be denoted either *explicitly* or *implicitly*. Temporal elements are used in queries for explicitly denoting the set of granules with respect to which queries have to be evaluated.

Example 4.10 The query:

> *select the professors that taught courses attended by more than 20 students in the time interval [1995,2000]*

is an example of query in which the set of granules in which the query condition must be verified is explicitly given. By contrast, the query:

> *select the assistant of the temporal database course when it was taught by Prof. Snodgrass*

is an example of query in which the set of granules is implicitly specified. $\diamond$

The fact that in *T*_ODMG we support multiple granularities entails addressing several issues. For instance, consider the first query of the previous example and suppose that the information related to the course taught by a professor is stored with respect to granularity *semesters*. Then, the granularity of the stored information and the granularity of the given time interval is different. Indeed, the time interval $[1995, 2000]$ refers to granularity *years*. Thus, the management of these cases has to be treated.

As we have said, the time with respect to which the query is evaluated can be expressed either explicitly, by specifying a granule or a set of granules, or implicitly, by means of a symbolic formalism.

For what concerns explicitly denoting the time with respect to which the query has to be evaluated, in the considered queries granules are denoted through their *labels* in the set $\mathcal{LABEL} = \bigcup_{G \in \mathcal{G}} \mathcal{LABEL}_G$, that is, through their "textual representation" as introduced in Section 3.1, rather than through the (less descriptive) granule index. We recall that given a label we know the granularity and the granule index associated with it. That is, each label identifies a unique granule of a single granularity. Similarly temporal intervals in queries are denoted through pairs of granule labels between square brackets and temporal elements are denoted by sequences of temporal intervals separated by commas. For instance, in the first query of Example 4.10 above the set of granules of granularity $years$, with respect to which the query has to be evaluated, is explicitly denoted though a temporal interval at granularity $years$ denoted through their labels, that is, $[1995, 2000]$.

To symbolically denote a set of time instants we use a formalism based on the notion of *temporal expression*, which extends the notion of temporal expression presented in [BFG98], which has been in turn inspired by the one proposed by Gadia and Nair in [GN93]. A temporal expression is a symbolic representation of a set of granules. A temporal expression is an expression that, evaluated on an object (the one through which we are navigating), returns a set of granules, that is, a temporal element.

In what follows we first introduce how temporal values expressed with respect to different granularities can be combined and compared. Then, we present the syntax and semantics of boolean expressions which are the basis for temporal expressions. Afterwards, temporal expression syntax and semantics are illustrated and temporal path expressions are presented.

4.3 Combining Temporal Values

In this section we discuss how temporal values related to different temporal granularities can be combined. An approach to combine and compare multiple granularity data has been previously proposed by us in [MBF+00]. We recall that (cf. Definition 3.12) temporal values are partial functions from the index set to the set of values of the inner type. That is, let $temporal_G(\tau) \in \mathcal{TT}$ be a temporal type, $[\![temporal_G(\tau)]\!]$ denotes the set of legal values of type $temporal_G(\tau)$:

$$\begin{aligned}&[\![temporal_G(\tau)]\!] = \\ &\qquad \{f \mid f : \mathcal{IS} \rightarrow \textstyle\bigcup_{i \in \mathcal{IS}} [\![\tau]\!]_i^G \text{ is a partial function such that} \\ &\qquad \text{for each } i \in \mathcal{IS} \text{ if } f(i) \text{ is defined then } f(i) \in [\![\tau]\!]_i^G\}\end{aligned}$$

Combining temporal values expressed with respect to different granularities raises several interesting issues.

Example 4.11 Let v_1 and v_2 be two values in $[\![\, temporal_G(\texttt{short}) \,]\!]$. Is the expression $v_1 + v_2$ meaningful? If it is, what is the resulting value? In addition, is the comparison of v_1 and v_2 possible? For instance, what is the result of $v_1 > v_2$? Suppose, in addition, that v_1 is a value of type $[\![\, temporal_G(\texttt{short}) \,]\!]$, whereas v_2 is a value of type $[\![\, temporal_H(\texttt{short}) \,]\!]$, $G \neq H$. What is the result of the previous expressions in this case?

$\diamondsuit$

The intuitive meaning of an operator *op* applied to two temporal values, v_1 and v_2, that is, the resulting value of the expression v_1 *op* v_2, is the evaluation "point to point", that is, "granule to granule" in our context, of the operation denoted by *op*.

Reasoning on temporal values we have to consider that they can be defined with respect to different granularities. One idea could be to force temporal values appearing in an expression to be expressed with respect to the same granularity. We believe that this is too restrictive. Usually, in programming languages, values belonging to different types can appear in the same expression. In addition, temporal information expressed with respect to different granularities can be compared in several interesting ways. Thus, we believe that expressions involving temporal values related to different granularities have to be managed properly. If the two values are expressed with respect to different granularities and the granularities associated with the types of the two values are comparable through the finer than relationship, the downward hereditary property is used to compute the resulting value. We recall that, downward hereditary property establishes that, given $v \in [\![\, temporal_G(\tau) \,]\!]$ for each granularity H such that $H \preceq G$, and for each $i, j \in \mathcal{IS}$ such that $H(j) \subseteq G(i)$, the value of v in granule j of H is the one in granule i of G.

Let, for instance, $\tau_1 = temporal_G(\tau_1')$ be the type of a value v_1 and $\tau_2 = temporal_H(\tau_2')$ be the type of another value v_2, such that $G \preceq H$. To compute the result of v_1 *op* v_2, the value with respect to the coarser granularity, that is, v_2, is converted, thanks to the downward hereditary property, to the finer granularity. We denote the converted value as v_2^H (cf. Definition 3.13). Then, the operation is computed "granule per granule" as in the previous case.

In case the two values are expressed with respect to different granularities that are in some way comparable, that is, there exists a granularity K that is finer than both then two value granularities, the two values are converted to such granularity K.[5] If none of the previous conditions is verified an undefined value is returned, that corresponds to an error detection.

The previous concepts are formalized by the following definition.

Definition 4.4 (*Semantics of Operations on Temporal Values*). Let v_1, $v_2 \in \mathcal{V}_{temp}$ be

[5]If K exists, then it is the greatest lower bound of H and G in $\mathcal{G}$, which we denote as $glb(H, G) = K$.

two values. In addition, let *op* be an operator defined between values of type τ then the semantics of v_1 *op* v_2 is defined as follows:

$$v_1 \ op \ v_2 = \begin{cases} \{\langle i, v_1(i) \ op \ v_2(i)\rangle \mid i \in \mathcal{IS}\} & \text{if } v_1, v_2 \in [\![\, temporal_G(\tau) \,]\!] \\ \{\langle i, v_1(i) \ op \ v_2\rangle \mid i \in \mathcal{IS}\} & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!] \text{ and } v_2 \in [\![\, \tau \,]\!] \\ \{\langle i, v_1 \ op \ v_2(i)\rangle \mid i \in \mathcal{IS}\} & \text{if } v_1 \in [\![\, \tau \,]\!] \text{ and } v_2 \in [\![\, temporal_G(\tau) \,]\!] \\ v_1 \ op \ v_2^G & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_2 \in [\![\, temporal_H(\tau) \,]\!], \text{ and } G \preceq H \\ v_1^H \ op \ v_2 & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_2 \in [\![\, temporal_H(\tau) \,]\!], \text{ and } H \preceq G \\ v_1^K \ op \ v_2^K & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_2 \in [\![\, temporal_H(\tau) \,]\!], \\ & K = glb(G, H) \\ \perp & \text{otherwise} \end{cases}$$

□

Let v be a temporal value. Since v is a partial function, there may exist $i \in \mathcal{IS}$ such that $v(i) = \perp$. We consider such *unknown* information as *null* values in OQL. Thus, they are considered as zero in integer expressions and three-values logics is adopted for boolean expressions.

Let in the previous definition τ be the type `short`, then the operation *op* can be an integer operator such as +, -, *, and so on, or a comparison operator such as =, $\neq$, >, >=, and so on.

The following example clarifies the previous definition.

Example 4.12 Let $v_1 = \{\langle i_{01/1999}, 100\rangle, \langle i_{02/1999}, 150\rangle, \langle i_{03/1999}, 300\rangle\}$ and $v_2 = \{\langle i_{01/1999}, 400\rangle, \langle i_{03/1999}, 50\rangle\}$ be two temporal values of type $temporal_{months}$(`short`) representing, for instance, the salary of two married persons. If we are interested in knowing the family income, that is, the sum of the two salaries, we have to compute the value of the integer expression $v_1 + v_2$. If we are interested in knowing which one of the two makes more money for each month, we have to compute $v_1 > v_2$. The result of the previous expressions according to the semantics presented in Definition 4.4 is:

$v_1 + v_2 = \{\langle i_{01/1999}, 500\rangle, \langle i_{02/1999}, 150\rangle, \langle i_{03/1999}, 350\rangle\}$

$v_1 > v_2 = \{\langle i_{01/1999}, false\rangle, \langle i_{03/1999}, true\rangle\}$. ◊

For each comparison operator *op* we introduce a *temporal* variation (op_T), whose intuitive meaning is to answer the following question: "when the relationship denoted by *op* holds?" These operators are the extension to a temporal context of comparison operators and the result of their evaluation is a temporal element, thus, the expressions in which such operators appear are called *temporal expressions* (cf. Section 4.4). The result of a temporal comparison of two temporal values is formally defined by the following definition.

Definition 4.5 (*Semantics of Temporal Comparison Operations on Temporal Values*). Let $v_1, v_4 \in \mathcal{V}_{temp}$ be two values. In addition, let *op* be a comparison operator defined between values of type τ then the semantics of $v_1\ op_T\ v_2$, where op_T denotes the temporal variation of *op* is defined as follows:

$$v_1\ op_T\ v_2 = \begin{cases} \{G(i) \mid v_1(i)\ op\ v_2(i) = true\} & \text{if } v_1,\, v_2 \in [\![\, temporal_G(\tau) \,]\!] \\ \{G(i) \mid v_1(i)\ op\ v_2 = true\} & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!] \text{ and } v_2 \in [\![\, \tau \,]\!] \\ \{G(i) \mid v_1\ op\ v_2(i) = true\} & \text{if } v_1 \in [\![\, \tau \,]\!] \text{ and } v_2 \in [\![\, temporal_G(\tau) \,]\!] \\ v_1\ op_T\ v_2^G & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_2 \in [\![\, temporal_H(\tau) \,]\!], \text{ and } G \preceq H \\ v_1^H\ op_T\ v_2 & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_1 \in [\![\, temporal_H(\tau) \,]\!], \text{ and } H \preceq G \\ v_1^K\ op_T\ v_2^K & \text{if } v_1 \in [\![\, temporal_G(\tau) \,]\!], \\ & v_2 \in [\![\, temporal_H(\tau) \,]\!], \\ & K = glb(G, H) \\ \perp & \text{otherwise} \end{cases}$$

□

Example 4.13 Consider again the two temporal values v_1 and v_2 presented in Example 4.12 representing the salary of two married persons. If we are interested in knowing when v_1 made more money than v_2 we have to compute $v_1 >_T v_2$. The resulting value is

$v_1 >_T v_2 = [i_{03/1999}, i_{03/1999}]^{months}$

Note that the returned temporal element represents a single granule, that is, the one denoted by label 03/1999 of granularity *months*.

Consider now two values: $v_1 \in [\![\, temporal_{months}(\texttt{short}) \,]\!]$ and $v_2 \in [\![\, temporal_{days}(\texttt{short}) \,]\!]$, such that $v_1 = \{\langle i_{12/1999}, 5\rangle, \langle i_{01/2000}, 7\rangle\}$ and $v_2 = \{\langle i_{01/12/1999}, 5\rangle, \langle i_{02/12/1999}, 10\rangle, \langle i_{03/12/1999}, 8\rangle, \langle i_{04/12/1999}, 1\rangle, ...\}$.[6] Suppose that we are interested in knowing when v_2 is greater than v_1, we should compute $v_1 <_T v_2$, whose semantics is

$$v_1 <_T v_2 = v_1^{days} <_T v_2 =$$
$$\{days(i) \mid v_1(i) < v_2(i) = true\} = \{days(i_{02/12/1999}), days(i_{03/12/1999}), ...\}$$

[6]We do not report all the values since they do not add any contribution to the example.

where $v_1^{days} = \{\langle i_{01/12/1999}, 5\rangle, \langle i_{02/12/1999}, 5\rangle, \langle i_{03/12/1999}, 5\rangle, \langle i_{04/12/1999}, 5\rangle, ...,$ $\langle i_{01/01/2000}, 7\rangle, \langle i_{02/01/2000}, 7\rangle, ...\}$. $\diamond$

Note that, in computing when a relationship denoted by a comparison operator holds between two values, we consider the intersection of the domains of the functions representing the values, that is, when both of them are defined, and we do not force the domain to be the same. This is in accordance with our intuition. If one asks: have John and his son always lived in the same house? The answer is yes if they have lived in the same house after John's son was born. In this case the domain of the temporal function corresponding to the value storing the history of the address of John's son is strictly included in the domain of the temporal function corresponding to the value storing the history of the address of John.

4.4 Temporal Expressions

In this section we present the syntax and semantics of temporal expressions. Temporal expressions are the mean by which the set of granules, that is, the temporal element, with respect to which a query is evaluated, are implicitly specified.

4.4.1 Syntax

Temporal expressions evaluated on an object, return the temporal element in which a boolean condition is satisfied on the object. They are somehow the temporal counterpart of boolean expressions. That is, a boolean expression answers to the query: is a certain condition verified? Whereas, temporal expressions answers the query: *when* is a certain condition verified? The idea of using temporal expressions to implicitly denote the instants in path expressions has been first introduced in [BFG98]. Another paper considering navigation access is by Cheng and Gadia [CG93]. Their language OOTempSQL provides a sublanguage for associative navigation relying on notions similar to our concept of temporal expression.

Temporal expressions are built by combining simple expressions which denote values belonging to $\mathcal{V}_{temp}$. Temporal expressions are specified according to the language whose syntax in BNF form is reported in Figure 4.1. Terminal symbol `nat` represents a natural number, `granularity` represents a granularity belonging to $\mathcal{G}$, `value` represents an element of $\mathcal{V}_{temp}$, `count` denotes the usual aggregate function, `attribute_name` represents an element of $\mathcal{AN}$ and `relationship_name` represents an element of $\mathcal{RN}$.

⟨temp_expr⟩ ::= ⟨simple_expr⟩ ⟨temp_comp_op⟩ ⟨simple_expr⟩ |
⟨temp_expr⟩ ⟨bool_op⟩ ⟨temp_expr⟩ | ⟨single_op⟩ (⟨temp_expr⟩) |
⟨slice_op⟩ (⟨temp_expr⟩, nat) | granularity(⟨temp_expr⟩)

⟨simple_expr⟩ ::= value | ⟨path⟩ | count(⟨simpl_expr⟩) |
⟨simpl_expr⟩ ⟨op⟩ ⟨simpl_expr⟩

⟨path⟩ ::= attribute_name | relationship_name |
⟨path⟩.attribute_name |
⟨path⟩.relationship_name

⟨temp_comp_op⟩ ::= $>_T$ | $<_T$ | $>=_T$ | $<=_T$ | $=_T$ | $!=_T$ | $\in_T$ | $\notin_T$

⟨op⟩ ::= + | - | * | / | $\cup$ | $\cap$ | \

⟨bool_op⟩ ::= and | or

⟨single_op⟩ ::= not|first|last|first_instant|last_instant

⟨slice_op⟩ ::= slice|inst_slice

Figure 4.1: Syntax of the language for expressing temporal expressions

Example 4.14 The following are examples of temporal expressions which can be evaluated with respect to an object of class `Course` of the database schema of Example 3.15:

- `room` $=_T$ `107 and T_assistant.room` $=_T$ `207`,
- `first(count(P_students)` $>_T$ `25 or T_professor.salary` $>_T$ `70)`,
- $years$(`T_assistant.salary` $>_T$ `55k`),

whereas, the following temporal expression:

`count(attends)` $>_T$ `5 and not(supervisor.teaches` $\in_T$ `attends)`

is an example of a temporal expression which can be evaluated on an object of class `Student` of the same schema. $\diamond$

4.4.2 Semantics

For giving the semantics of temporal expressions, we start from the semantics of simple expressions which are defined according to the second rule of Figure 4.1.

A simple expression is an expression that evaluated on an object denotes a value. It can be a value itself, the navigation through a path, the count operator applied to an expression, and, finally, an operation applied to two expressions.

Example 4.15 The following are simple expressions which can be evaluated on an object of class `Course` of the schema of Example 3.15.

- `T_professor.salary`, is a simple expression which evaluated on an object of class `Course` returns a value storing the history of the salaries of the professors who have ever taught that course;
- `count(P_students)`, is a simple expression which evaluated on an object of class `Course` returns a value storing the history of the number of students participating to that course;
- `count(P_students)+5`, is a simple expression which evaluated on an object of class `Course` returns a value storing the history of the number of students participating to that course increased by 5. $\diamond$

In case of paths, which represent the navigation through objects, the evaluation of a path expression $p_1.p_2.\cdots.p_n$ is the value of property p_n starting to navigate from the given object through properties $p_1,\ldots,p_{n-1}$. In this case one has to take into account the fact that each property p_i, $i \in [1,n]$, through which the navigation is performed, can be expressed with respect to a different granularity. Consider indeed a path $p_1.p_2.\cdots.p_n$ such that p_1 is a property of the class of the object on which the expression is evaluated whose domain is $temporal_{G_1}(\tau_1)$ and p_i, for $i \in [2,n]$, is a property of class τ_i whose domain is $temporal_{G_i}(\tau_i)$. The value denoted by this expression is a temporal value of granularity $glb(G_1,\ldots,G_n)$ whose value in each granule is obtained by evaluating the value of the path at that granule. If $glb(G_1,\ldots,G_n)$ in $\mathcal{G}$ does not exist, then the value of such expression is not defined. In defining the semantics, we restrict our attention to temporal properties since the navigation through static properties does not raise new problems.

Example 4.16 Consider again the database schema of Example 3.15. The path `supervisor.teaches.P_students`, evaluated on an object of class `Student` of that schema whose supervisor has ever taught an introductory course, denotes a temporal value of type $temporal_{days}$(`set` $<$ `Student` $>$). Indeed, the type of property `supervisor` in class `Student` is $temporal_{years}$(`Professor`), the type of property `teaches` in class `Professor` is $temporal_{years}$(`Course`), and, finally, the type of property `P_students` in class `I_Course` is $temporal_{days}$(`set` $<$ `Student` $>$). Thus, $glb(years, years, days) = days$, which implies that $days$ is the granularity of the returned value. $\diamond$

In addition, we impose the restriction that path navigation across multivalued properties is not allowed. Such a restriction is in agreement with the OQL approach which forbids multivalued property navigation. The semantics of simple expressions is formally defined in Definition 4.6.

Definition 4.6 (*Semantics of Simple Expressions*). Let $S = (\pi^+_{temp}, \nu)$ be the state of a T_ODMG database (cf. Definition 2.21), $\mathcal{OI}$ be a set of object identifiers, and $\mathcal{V}_{temp}$ be the set of T_ODMG values. In addition, let Exp_{simple} be the set of well-formed simple expressions as specified in Figure 4.1. Then the semantics of simple expressions is defined through function:

$$\mathcal{E}_{simple} : Exp_{simple} \rightarrow (\mathcal{OI} \rightarrow \mathcal{V}_{temp})$$

such that, given an expression $e \in Exp_{simple}$ and an object obj such that $obj = (id, N, v, r, c, [i,j]^{G_I})$[7] (cf. Section 3.5) and $obj.id = o \in \mathcal{OI}$:

[7] We recall that each object component is accessed through dot notation. That is, *obj.id* denotes the object identifier, *obj.N* denotes the set of object names, and so on.

- if $e = v$, $v \in \mathcal{V}_{temp}$, then $\mathcal{E}_{simple} [\![v]\!] o = v$;
- if $e = p_1.p_2.\cdots.p_n$, with $p_1, p_2, \ldots, p_n$ properties, $\mathcal{E}_{simple} [\![p_1.p_2.\cdots.p_n]\!] o = \{\langle i, (\mathcal{E}_{simple} [\![p_2.\cdots.p_n]\!] \nu(o).p_1^K(i))^K(i)\rangle \mid K = glb(G_1, \ldots, G_n), i \in \mathcal{IS}\}$ if the type of property p_1 in *obj.c* is $temporal_{G_1}(c_1) \in \mathcal{TT}$ and the type of property p_i in c_{i-1} is $temporal_{G_i}(c_i) \in \mathcal{TT}$, $i \in [2, n]$;
- if $e = \mathtt{count}(e')$, $e' \in Exp_{simple}$, then $\mathcal{E}_{simple} [\![\mathtt{count}(e)]\!] o = \mid \mathcal{E}_{simple} [\![e]\!] o \mid$[8]
- if $e = e_1\ op\ e_2$, then $\mathcal{E}_{simple} [\![e_1\ op\ e_2]\!] o = (\mathcal{E}_{simple} [\![e_1]\!] o)\ op\ (\mathcal{E}_{simple} [\![e_2]\!] o)$.

□

Note that in the last case of Definition 4.6 the result of the evaluation of each expression returns a value in $\mathcal{V}_{temp}$. In case of temporal values we refer to the semantics presented in Section 4.3 to obtain the resulting value.

Intuitively, the resulting value of a temporal expression is the temporal element in which all the conditions expressed by each expression composing it are verified. An expression composing it can be a comparison of temporal values according to a temporal comparison operator op_T. How to compare temporal values according to the temporal counterpart of comparison operators has been specified in Section 4.3. In that section issues related to the comparison of values expressed with respect to different granularities have been considered. The following example presents the granularity of the temporal elements of the temporal comparisons involved in temporal expressions presented in Example 4.14.

Example 4.17 Referring to the boolean expressions of Example 4.14, evaluated on an object of class `I_Course`:

- `room` $=_T$ `107` returns a temporal element of granularity *years*,
- `T_assistant.room` $=_T$ `207`, `T_professor.salary` $>_T$ `70`, and `T_assistant.salary` $>_T$ `55` return a temporal element of granularity *months*,
- `count(P_students)` $>_T$ `25` returns a temporal element of granularity *days*.

Each temporal element corresponds to the set of time granules in which the specified condition holds on the database, according to the semantics of comparison operators presented in Section 4.3. ◊

Intuitively, each temporal expression uniquely denotes a set of granules, that is, a temporal element.

[8]We recall that given a generic set S, $\mid S \mid$ denotes the cardinality of S.

Example 4.18 Referring to the temporal expressions of Example 4.14, evaluated on an object of class `I_Course`:

- `room` $=_T$ `107 and T_assistant.room` $=_T$ `207` returns a set of temporal intervals of granularity *months*, obtained as the intersection of the intervals in which the two conditions are satisfied;

- $first$(`count(P_students)` $>_T$ `25 or T_professor.salary` $>_T$ `70`) returns a temporal interval of granularity *days*, obtained as the first interval in the union of the intervals in which the two conditions are satisfied;

- $years$(`T_assistant.salary` $>_T$ `55`) returns a set of temporal intervals of granularity *years* if the set of intervals of granularity *months* in which the condition is satisfied exactly corresponds to a set of years, it is undefined otherwise.

Moreover, `count(attends)` $>_T$ `5 and not(supervisor.teaches` $\in_T$ `attends)` evaluated on an object of class `Student` returns a set of temporal intervals of granularity *years* obtained by eliminating from the years in which the student attended at least five courses the ones in which he/she attended the course taught by her supervisor. $\diamond$

The semantics of a temporal expression, that is, the temporal element it denotes, is formalized by means of function

$$\mathcal{E}_{temp} : Exp_{temp} \rightarrow (\mathcal{OI} \rightarrow \mathcal{TELEM})$$

defined in Figure 4.2.

In Figure 4.2 symbols e, e_1, and e_2 denote temporal expressions. Symbol o denotes an object identifier, that is, $o \in \mathcal{OI}$, n denotes a natural number, and, finally, G denotes a granularity in $\mathcal{G}$.

In case of temporal expressions involving the negation operator `not`, the semantics is defined using the set operation complement, that is $\mathcal{E}_{temp}[\![\, \mathtt{not}(e) \,]\!]\, o = (\mathcal{E}_{temp}[\![\, e \,]\!]\, o)^c$. The universe with respect to which the complement is computed is the set of granules in which expression e on object o is defined. Let $def(e)$ be the set of granules in which e is defined, then $(\mathcal{E}_{temp}[\![\, e \,]\!]\, o)^c = def(e) \setminus \mathcal{E}_{temp}[\![\, e \,]\!]\, o$.

Note that the only meaningful use of the conversion operator G in temporal expressions is for upward conversion, that is, to convert to a coarser granularity, to obtain a granule-valued expression (whose usefulness will be clarified below). According to the definitions presented in Section 4.2 of conversion of temporal interval (cf. Definition 4.2) and temporal element (cf. Definition 4.3), the semantics of upward conversion we consider, converts

$$\mathcal{E}_{bool}\llbracket e_1 op_T e_2 \rrbracket o = (\mathcal{E}_{simple}\llbracket e_1 \rrbracket o)\ op_T\ (\mathcal{E}_{simple}\llbracket e_2 \rrbracket o)$$

$$\mathcal{E}_{temp}\llbracket e_1 op e_2 \rrbracket o = \begin{cases} \mathcal{E}_{temp}\llbracket e_1 \rrbracket o \cup \mathcal{E}_{temp}\llbracket e_2 \rrbracket o & \text{if } op\text{= } \texttt{or} \\ \mathcal{E}_{temp}\llbracket e_1 \rrbracket o \cap \mathcal{E}_{temp}\llbracket e_2 \rrbracket o & \text{if } op\text{= } \texttt{and} \end{cases}$$

$$\mathcal{E}_{temp}\llbracket op(e) \rrbracket o = \begin{cases} (\mathcal{E}_{temp}\llbracket e \rrbracket o)^c & \text{if } op\text{= } \texttt{not} \\ \Pi(\mathcal{E}_{temp}\llbracket e \rrbracket o, 1) & \text{if } op\text{= } \texttt{first} \\ \Pi(\mathcal{E}_{temp}\llbracket e \rrbracket o, m) & \text{if } op\text{= } \texttt{last} \text{ and } m = \mid \mathcal{E}_{temp}\llbracket e \rrbracket o \mid; \\ [i,i]^G & \text{if } op = \texttt{first\_instant} \text{ and } i = min(\Pi(\mathcal{E}_{temp}\llbracket e \rrbracket o, 1)) \\ [i,i]^G & \text{if } op = \texttt{last\_instant}, i = max(\Pi(\mathcal{E}_{temp}\llbracket e \rrbracket o, m)), \text{ and} \\ & \text{if } m = \mid \mathcal{E}_{temp}\llbracket e \rrbracket \mid \end{cases}$$

$$\mathcal{E}_{temp}\llbracket op(e,n) \rrbracket o = \begin{cases} \Pi(\mathcal{E}_{temp}\llbracket e \rrbracket o, n) & \text{if } op\text{= } \texttt{slice} \\ [i,i]^G & \text{if } op = \texttt{instant\_slice} \\ & \text{and } i \text{ is the } i\text{th granule in } \mathcal{E}_{temp}\llbracket e \rrbracket o \end{cases}$$

$$\mathcal{E}_{temp}\llbracket G(e) \rrbracket o = G(\mathcal{E}_{temp}\llbracket e \rrbracket o)$$

Figure 4.2: Semantics of temporal expressions

expressions that denote a superset of a set of granules of the coarser granularity. Actually, upward conversion could be extended to arbitrary intervals, with different alternative semantics. Consider for instance the expression $years$(`T_assistant.salary` $>_T$ `55`) on the schema of Example 3.15, where the granularity of attribute `salary` is *months*, possible semantics are:

- return the years in which the condition is always verified (i.e., it is verified in all the months of the year), even if it is verified also in months outside the year;
- return the years in which the condition is always verified (i.e., it is verified in all the months of the year), if it is verified for a set of months which exactly denotes a set of years;
- return the years in which the condition is sometimes verified (i.e., it is verified in a month of the year);
- convert through a coercion function the value of `T_assistant.salary` to granularity *years* and then evaluate the condition.

The current semantics correspond to the first one. That is, testing the condition, obtaining a set of months, then, if a part of them, possibly all, corresponds to a set of years, return those years, otherwise the value of the expression is undefined. A possibility is to support all the four semantics described, corresponding to four different syntactic notations. We leave the support of all the four semantics for future work.

In the following we refer to temporal expressions denoting a single granule as *granule-valued* temporal expressions.

Definition 4.7 (*Granule-valued Temporal Expression*). Let $e \in Exp_{temp}$ be a temporal expression, then e is a granule-valued temporal expression with respect to an object identified by oid o if and only if $\mathcal{E}_{temp}[\![\, e \,]\!]\, o = [i, i]^G$, $i \in \mathcal{IS}$.[9] □

Note that interval conversions can be exploited to obtain granule-valued temporal expression.

Example 4.19 Let $e \in Exp_{temp}$ be a temporal expression such that $\mathcal{E}_{temp}[\![\, e \,]\!]\, o = [i_{01/01/1995}, i_{31/12/1995}]^{days}$. Then e is not granule-valued with respect to granularity $days$, but $years(e)$ is granule-valued, since $\mathcal{E}_{temp}[\![\, years(e) \,]\!]\, o = years(\mathcal{E}_{temp}[\![\, e \,]\!])o = years([i_{01/01/1995}, i_{31/12/1995}]^{days}) = [i_{1995}, i_{1995}]^{years}$. ◊

4.5 Temporal Path Expressions

Temporal path expressions [BFG98] are obtained as an extension of classical path expressions of object-oriented languages [BNPS92], in that for each attribute access a time can be specified, in addition to the attribute name. In such a way, the value of the attribute at a specified time is denoted. In [BFG98] the time can be expressed either explicitly, by specifying a time instant or a set of time instants, that is, a temporal element, or implicitly, by means of a temporal expression. The language to express path expressions presented in [BFG98] allows the nesting of attribute accesses expressed by means of the postfix dot notation. The restriction is imposed that all attribute accesses in the path expression, but the last one, are qualified with a time instant, and not with a set of time intervals, thus denoting a non-temporal value, on which a further attribute access can be specified. In the thesis, we extend that notion to a multi-granularity data model in which properties traversed in the path can be expressed at different granularities and the time instants are in this context granules of different granularities. Since, moreover, we refer to the ODMG data model which supports both attributes and relationships, our path expressions can traverse both attributes and relationships. In [BFG98] path expressions allowing a set of time intervals to qualify an intermediate attribute access in the path are also discussed.

[9] The usefulness of granule-valued temporal expressions will be made clear in the remaining part of the chapter.

However, since that extension complicates the presentation of all notions related to path expressions and makes them more difficult to understand, we do not consider them in the thesis, for the sake of clarity.[10]

As we have mentioned above, internal nodes of a path expression must produce a single non-temporal value (an object identifier) to which subsequent accesses are applied. In a multi-granularity context, to ensure that property, whichever the granularities of the accessed property and of the granule are, we rely on the use of *coercion function* (cf. Definition 3.19). In what follows, we first discuss why we need coercion functions and then how they are used.

Consider an access to a property p of an object o at a granule identified by label l_i^G. Let the property be stored at granularity H, that is, its domain is $temporal_H(\tau)$. If $G \preceq H$, as discussed in Section 3.3, we can access the value of property p at granule identified by l_i^G, and this access uniquely denotes a value. Let $S = (\pi^+_{temp}, \nu)$ be a database state, we denote such access as $\nu(o).p^G(i)$. If, by contrast, $H \prec G$, property p may assume different values in the granule identified by l_i^G. For instance, consider the class `Person` whose definition has been presented in Figure 3.5. If we access property `address` of a `Person` object specifying a year, since that property may vary every month, the property may take different values over the year, corresponding to the fact that the person has moved during that year. In this case we make use of coercion functions to obtain a single value starting from those taken by property p in the granule l_i^G.

Coercion functions have been defined by Definition 3.19. Here we allow one to attach such functions to property accesses, and not only to property definitions. If a coercion function is specified for a property access this function is used to evaluate the expression, even if a different coercion function has been specified in the schema for the accessed property. If, by contrast, no coercion function is specified for a property access, the one associated with the property in the schema is employed.

Wherever they are specified, coercion functions are employed to convert a temporal value to a coarser granularity. Not all the coercion functions introduced in Figure 3.2 are meaningful in this context, in particular, aggregate coercion functions do not make sense if applied to object-valued properties, as those accessed at intermediate nodes of a path expression. Thus, only selective and user-defined coercion functions are considered for internal nodes.

[10]For the same reason, we do not consider here property accesses for which no time qualifier is specified nor those for which the *all* qualifier introduced in [BFG98] is specified. Since they are introduced only for syntactic convenience, their extension to a multi-granularity context does not bring in any relevant issue. Similarly, we consider here only the weak interpretation of temporal path expressions, according to the terminolgy of [BFG98].

$$
\begin{array}{l}
\langle \texttt{path\_expr} \rangle ::= \langle \texttt{simple\_path\_expr} \rangle \mid \langle \texttt{term\_path\_expr} \rangle \\[1em]
\langle \texttt{simple\_path\_expr} \rangle ::= \texttt{var} \mid \texttt{var.}\langle \texttt{path} \rangle \mid \\
\qquad\qquad \langle \texttt{simple\_path\_expr} \rangle \downarrow \texttt{granule} \mid \langle \texttt{simple\_path\_expr} \rangle \downarrow^{\langle coerc\_func \rangle} \texttt{granule} \mid \\
\qquad\qquad \langle \texttt{simple\_path\_expr} \rangle \downarrow \langle \texttt{temp\_expr} \rangle \mid \langle \texttt{simple\_path\_expr} \rangle \downarrow^{\langle coerc\_func \rangle} \langle \texttt{temp\_expr} \rangle \\[1em]
\langle \texttt{term\_path\_expr} \rangle ::= \langle \texttt{simple\_path\_expr} \rangle \downarrow \texttt{temp\_element} \mid \\
\qquad\qquad \langle \texttt{simple\_path\_expr} \rangle \downarrow^{\langle coerc\_func \rangle} \texttt{temp\_element}
\end{array}
$$

Figure 4.3: Syntax of the language for expressing temporal path expressions

4.5.1 Syntax

The syntax of temporal path expressions is specified according to the BNF presented in Figure 4.3. Terminal symbols of Figure 4.3 have the following meaning: `var` denotes a variable, `granule` denotes a granule label, and, finally, `temp_element` denotes a temporal element. In addition, the rule of the non-terminal symbol ⟨`coerc_func`⟩ can be found in Figure 3.2, and the rules of the non-terminal symbols ⟨`temp_expr`⟩ and ⟨`path`⟩ can be found in Figure 4.3.

Example 4.20 Consider the schemas of Figures 3.3 and 3.5, and let `X` be a variable of type `Course`.

- $\texttt{X.T\_assistant} \downarrow \texttt{1st\_sem\_1999}$,
- $\texttt{X.T\_professor} \downarrow \texttt{first\_instant(count(P\_students)} >_T 25\texttt{).business\_hours} \downarrow$
  $\downarrow \texttt{(count(supervises)} >_T 5\texttt{)}$,
- $\texttt{X.T\_assistant} \downarrow^{\texttt{all}} \texttt{(room}=_T\texttt{107).supervised\_lab} \downarrow \texttt{(count(assists)} >_T 2\texttt{)}$

are examples of path expressions. The first one denotes the assistant of the course during the first semester of 1999. The second one denotes the business hours, in time intervals in which he/she supervised more than five students, of the professor who was teaching the course in the first instant in which it was attended by more than 25 students. Finally, the third one denotes the laboratory supervised, in a period in which he/she was the assistant of more than two courses, by the researcher that was the assistant of the course when it was located in room 107. ◊

4.5.2 Semantics

The semantics of a path expression (i.e., the value it denotes) can only be specified starting from an oid-assignment, that is, a function assigning oids to object-denoting variables. However, the value denoted by a path expression depends on the temporal specifications it contains.

Consider first simple path expressions, for which a granule l_i^G is specified (either implicitly or explicitly), that is, path expressions of the form $e.p \downarrow l_i^G$ with a coercion function optionally specified ($e.p \downarrow^C l_i^G$). Let o be an object identifier to which e evaluates, and let the granularity of property p for object o be H. In addition, let obj be the object identified by o, that is, $obj.id = o$.

Moreover, given a temporal property p and a granule identified by l_i^G, $p(i)$ denotes the value of p in the granule l_i^G. Finally, let $K \preceq G$, $p^K(i)$ denotes the value of property p with respect to granularity K, such value is equal to $p(j)$ where $K(i) \subseteq G(j)$.

The value denoted by the path expression with respect to a database state $S = (\pi_{temp}^{+}, \nu)$ and an object o is:

- $\nu(o).p(i)$, if $G = H$ (in this case, the eventually specified coercion function is irrelevant);
- $\nu(o).p^G(i)$, if $G \prec H$ and $G(i) \subseteq obj.lifespan$, whereas it is undefined if $G \prec H$ and $G(i) \not\subseteq obj.lifespan$ (see the discussion in Section 2.6 concerning the control on the lifespan, and note that, also in this case, the eventually specified coercion function is irrelevant);
- $C(\nu(o).p)(i)$ if $H \prec G$, where C is the coercion function specified for the access, if any, and the one associated with property p in the class to which o belongs otherwise;[11]
- $C(\nu(o).p)^K(j)$ if H and G are not comparable under $\preceq$ but they have a least upper bound K, that is, $K = lub(G, H)$, $G(i) \subseteq K(j)$, and C is the coercion function determined as in the case above. In this case, function C is used to coerce values of property p to granularity K and the value of the property in the K-granule containing $G(i)$ is accessed.

Example 4.21 Let $\mathtt{o_i}, \mathtt{o_e}, \mathtt{o_t}$ be identifiers of objects of classes `I_Course`, `E_Course`, `T_Course` of the schema of Figure 3.3, respectively.

[11] If $H \prec G$ and no coercion function is specified for the access, nor attached to property p in the class to which o belongs, the value of the expression is undefined.

- the value denoted by $\texttt{o}_\texttt{t}$.T_assistant $\downarrow$ 1st_sem_1999 is simply the value stored for the relationship in that granule;

- the value denoted by $\texttt{o}_\texttt{i}$.T_assistant $\downarrow$ 1st_sem_1999 is simply the value stored for the relationship in the *year*-granule 1999, provided that $\texttt{o}_\texttt{i}$ exists in the first semester of 1999;

- the value denoted by $\texttt{o}_\texttt{e}$.T_assistant $\downarrow$ 1st_sem_1999 is the value obtained by applying coercion function main to the values stored for the relationship in the *months*-granules corresponding to the first semester of 1999, that is, the researcher that served as assistant for most months;

- the value denoted by $\texttt{o}_\texttt{e}$.T_assistant $\downarrow^{\texttt{first}}$ 1st_sem_1999 is the value obtained by applying coercion function first to the values stored for the relationship in the *months*-granules corresponding to the first semester of 1999, that is, the researcher that served as assistant in the first month of the semester. $\diamond$

Consider now the terminal path expression $e.p \downarrow \Upsilon^G$, where Υ^G is a temporal element (either explicitly or implicitly denoted). The value associated with this expression is the restriction of the temporal value $\nu(o).p$ to the time instants in Υ^G. Different interpretations of this path expression are however possible if the value and the temporal element are expressed at different granularities. Let o be the oid of the object to which e evaluates and the granularity of property p for object o be H. Different cases can be distinguished:

- $G = H$: the value of the expression is a temporal value of granularity G which is the restriction of $\nu(o).p$ to the time instants in Υ^G (as formally specified below);

- $G \prec H$: the value of the expression is a temporal value of granularity G which is $\nu(o).p$ seen as a value of granularity G and restricted to the time instants in Υ^G that also belong to the lifespan of o;

- $H \prec G$: two alternative interpretations can be given to the path expressions:
  - if no coercion function is specified for the access, the expression denotes a temporal value of granularity H obtained by restricting $\nu(o).p$ to the H-granules contained in Υ^G;
  - if a coercion function C is specified for the access, the expression denotes a temporal value of granularity G obtaining by restricting $C(\nu(o).p)$ to the time instants in Υ^G;

- G and H are not comparable under $\preceq$: two alternative interpretations can be given to the path expressions:

– if no coercion function is specified for the access, the expression denotes a temporal value of granularity $J = glb(G, H)$ which is the restriction of $\nu(o).p$ (seen as a value of granularity J) to the J-granules belonging to Υ^G that also belong to the lifespan of o;

– if a coercion function C is specified for the access, the expression denotes a temporal value of granularity G obtaining by restricting the coercion $C(\nu(o).p)$ of $\nu(o).p$ at granularity $K = lub(G, H)$, to the time instants in Υ^G.

Example 4.22 Consider again the schema of Example 3.15 and let $\mathtt{o_a}$ be the identifier of a `Researcher` object. We recall that, in class `Researcher`, the type of the attribute `room` is $temporal_{months}(\mathtt{short})$, the type of the attribute `supervised_lab` is $temporal_{weeks}(\mathtt{string})$, and the type of the relationship `assists` is $temporal_{months}(\mathtt{set} < \mathtt{Course} >)$. Thus, $\mathtt{count(assists)} >_{\mathtt{T}} 2$ denotes a set of intervals at granularity *months*.

- $\mathtt{o_a.room} \downarrow (\mathtt{count(assists)} >_{\mathtt{T}} 2)$ denotes a temporal value of type $temporal_{months}(\mathtt{short})$ obtained by restricting the value of the `room` attribute to those days belonging to months in which the researcher was the assistant of more than two courses;

- $\mathtt{o_a.room} \downarrow [\mathtt{01/01/99}, \mathtt{18/01/99}]$ denotes a temporal value of type $temporal_{days}(\mathtt{short})$ obtained by restricting the value of the `room` attribute to those days belonging to specified temporal interval and to the lifespan of $\mathtt{o_a}$;

- $\mathtt{o_a.room} \downarrow [\mathtt{1999}, \mathtt{2000}]$ denotes a temporal value of type $temporal_{months}(\mathtt{short})$ obtained by restricting the value of the `room` attribute to the months of 1999 and 2000;

- $\mathtt{o_a.room} \downarrow^{\mathtt{first}} [\mathtt{1999}, \mathtt{2000}]$ denotes a temporal value of type $temporal_{years}(\mathtt{string})$ obtained by restricting the value of the `room` attribute corced to the *years* granularity by means of the `first` function, to years 1999 and 2000; thus, it associates to the 1999 granule the assistant in the first month of 1999 and to the 2000 granule the assistant in the first month of 2000;

- $\mathtt{o_a.supervised\_lab} \downarrow (\mathtt{count(assists)} >_{\mathtt{T}} 2)$ denotes a temporal value of type $temporal_{days}(\mathtt{string})$ obtained by restricting the value of the `supervised_lab` attribute to those days belonging to months in which the researcher was the assistant of more than two courses, and belonging to the lifespan of $\mathtt{o_a}$. $\diamond$

We now formalize these notions. Consider first the restriction of a temporal value to a temporal element, as specified by the following definition.

Definition 4.8 *(Temporal Value Restriction).* Let $v \in [\![temporal_H(\tau)]\!]$ be a temporal value and let $\Upsilon^G \in \mathcal{TELEM}_G$ be a temporal element. The restriction of v to Υ^G, denoted as $v_{|\Upsilon^G}$, is defined as follows:

- if $G \preceq H$, $v_{|\Upsilon^G}$ is a temporal value of type $temporal_G(\tau)$ such that:

$$v_{|\Upsilon^G}(i) = \begin{cases} v(j) & \text{if } G(i) \in \Upsilon^G \text{ and } G(i) \subseteq H(j) \\ undefined & \text{otherwise} \end{cases}$$

- if $H \prec G$, $v_{|\Upsilon^G}$ is a temporal value of type $temporal_H(\tau)$ such that:

$$v_{|\Upsilon^G}(i) = \begin{cases} v(i) & \text{if } G(j) \in \Upsilon^G \text{ exists such that } H(i) \subseteq G(j) \\ undefined & \text{otherwise} \end{cases}$$

□

We are now ready to define the semantics of a path expression. Let Exp_{path} be the set of well-formed path expressions built according to the BNF grammar specified in Figure 4.3, $\vartheta : Var \rightarrow \mathcal{OI}$ be an oid-assignment, and Θ be the set of all oid-assignments. The semantics of a path expression e under oid-assignment ϑ and with respect a database state $S = (\pi^{+}_{temp}, \nu)$ is defined through the following semantic function

$$\mathcal{E}_{path} : Exp_{path} \rightarrow (\Theta \rightarrow \mathcal{V}_{temp})$$

which is formally specified in Figure 4.4.

In Figure 4.4 we use function γ to denote the granularity of a temporal value, that is, if v is a temporal value of type $temporal_G(\tau)$, $\gamma(v) = G$.

In addition, in Figure 4.4, X denotes a variable, p denotes an object property, that is, $p \in \mathcal{AN} \cup \mathcal{RN}$, l_i^G denotes a granule label, that is, $l_i^G \in \mathcal{LABEL}_{\mathcal{G}}$, te denotes a temporal expression, that is, $te \in Exp_{temp}$, C denotes a coercion function, and, finally, Υ^G denotes a temporal element, that is, $\Upsilon^G \in \mathcal{TELEM}_G$.

4.6 Concluding Remarks

In this chapter we have investigated how values expressed with respect to different granularities can be combined and compared. In addition, we have investigated navigation through objects whose property values are expressed with respect to several granularities. This can be considered as the *core* of a language which extends OQL to query data expressed with respect to different granularities.

$$\mathcal{E}_{path}\llbracket X \rrbracket \vartheta = \vartheta(X)$$

$$\mathcal{E}_{path}\llbracket X.p \rrbracket \vartheta = \nu(\mathcal{E}_{path}\llbracket X \rrbracket \vartheta).p = \nu(\vartheta(X)).p$$

$$\mathcal{E}_{path}\llbracket X.path.p \rrbracket \vartheta = \nu(\mathcal{E}_{path}\llbracket X.path \rrbracket \vartheta).p$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow l_i^G \rrbracket \vartheta = \begin{cases} \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p(i) & \text{if } G = \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \text{ or} \\ & (G \prec \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \text{ and} \\ & \ G(i) \subseteq \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).lifespan) \\ \mathcal{E}_{path}\llbracket e.p \downarrow^C l_i^G \rrbracket \vartheta & \text{if } \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \prec G \\ & \text{and } C \text{ is the coerc. func. associated with } p \\ undefined & \text{otherwise} \end{cases}$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow^C l_i^G \rrbracket \vartheta = \begin{cases} \mathcal{E}_{path}\llbracket e.p \downarrow l_i^G \rrbracket \vartheta & \text{if } G \preceq \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \\ C(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p)^K(j) & \text{otherwise,} \\ & K = lub(G, \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p)), \text{ and} \\ & \ G(i) \subseteq K(j) \end{cases}$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow te \rrbracket \vartheta = \mathcal{E}_{path}\llbracket e.p \downarrow (\mathcal{E}_{temp}\llbracket te \rrbracket \vartheta(X)) \rrbracket \vartheta$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow^C te \rrbracket \vartheta = \mathcal{E}_{path}\llbracket e.p \downarrow^C (\mathcal{E}_{temp}\llbracket te \rrbracket \vartheta(X)) \rrbracket \vartheta$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow \Upsilon^G \rrbracket \vartheta = \begin{cases} \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p_{|\Upsilon^G} & \text{if } \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \preceq G \\ \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p_{|\Upsilon'^G} & \text{if } G \preceq \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \\ & \Upsilon'^G = \{G(i) \mid G(i) \in \Upsilon^G \text{ and} \\ & G(i) \cap \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).lifespan \neq \emptyset\} \\ \nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p_{|J(\Upsilon^G)} & \text{otherwise (i.e., } \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \text{ and } G \\ & \ \text{are not comparable under } \preceq), \\ & J = glb(\gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p)), G) \end{cases}$$

$$\mathcal{E}_{path}\llbracket e.p \downarrow^C \Upsilon^G \rrbracket \vartheta = \begin{cases} \mathcal{E}_{path}\llbracket e.p \downarrow^C \Upsilon^G \rrbracket \vartheta & \text{if } G \preceq \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p) \\ C(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p)_{|K(\Upsilon^G)} & \text{otherwise,} \\ & K = lub(G, \gamma(\nu(\mathcal{E}_{path}\llbracket e \rrbracket \vartheta).p)) \end{cases}$$

Figure 4.4: Semantics of path expressions

Several different semantics could be devised in evaluating the presented expressions. Among the different alternatives we have chosen what is, according to our point of view, the most intuitive. We leave for future work the introduction in the language of syntactic constructs according to which all the possible semantics could be supported. In addition, our future works include the definition of a full extension to OQL to query temporal data and query optimization.

Chapter 5

A_ODMG: Extending the ODMG Object Model with Triggers

In this chapter, we extend the standard for object-oriented databases, ODMG, with reactive features, by proposing a language for specifying triggers and defining its semantics. The extended model is termed A_ODMG. The model considered for extension is the complete ODMG object model presented in Chapter 2.

This extension makes three different specific contributions. First, the definition of a declarative data manipulation language for ODMG, which is missing in the current version of the standard. Such a definition requires revisiting data manipulation in ODMG and also addressing issues related to set-oriented vs instance-oriented computation. Then, the definition of a trigger language for ODMG, unifying also the SQL:1999 proposal and providing support for trigger inheritance and overriding. We recall that SQL triggers have been described in Section 1.3.2 and in Appendix B. Finally, the development of a formal semantics for the proposed data manipulation and trigger languages.

This chapter is organized as follows. In Section 5.1 we present the running example schema we refer to throughout this chapter. Section 5.2 is devoted to data manipulation in the ODMG model. Section 5.3 defines our trigger language, and Section 5.4 presents its semantics. Finally, Section 5.5 concludes the chapter.

5.1 Running Example Schema

This chapter refers to a running example schema for movies, adapted from [UW97], whose ODMG schema is presented in Figure 5.1.

```
class Movie (extent Movies key (title, year)) {
              attribute string title;
              attribute integer year;
              attribute integer length;
              attribute enum Film {color,blackAndWhite} filmType;
              relationship set<Star> stars
                           inverse Star::starredIn;
              relationship Studio ownedBy
                           inverse Studio::owns;
              relationship MovieExec producedBy
                           inverse MovieExec::produces;
              void changeTypeOrYear();
              };
class Star (extent Stars) {
              attribute string name;
              attribute struct Addr {string street, string city} address;
              attribute enum Gend {male, female} gender;
              relationship set<Movie> starredIn
                           inverse Movie::stars;
              relationship set<Cartoon> voiceIn
                           inverse Cartoon::voices;
              void confirmDel();
              };
class Studio (extent Studios keys address, name) {
              attribute string name;
              attribute string address;
              relationship MovieExec president
                           inverse MovieExec::manages;
              relationship set<Movie> owns
                           inverse Movie::ownedBy;
              };
class MovieExec (extent MovieExecs) {
              attribute string name;
              attribute struct Addr {string street, string city} address;
              attribute integer netWorth;
              relationship set<Movie> produces
                           inverse Movie::producedBy;
              relationship Studio manages
                           inverse Studio::president;
              };
class Cartoon extends Movie (extent Cartoons) {
              relationship set<Star> voices
                           inverse Star::voiceIn;
              };
```

Figure 5.1: Example of ODMG schema

5.2 Data Manipulation in ODMG

In ODMG 3.0 no data manipulation language is defined. The only data manipulation primitives considered apart from language bindings are the `Object new()` operation associated with the `ObjectFactory` interface, which is used to create a new object, and the `void delete()` operation associated with the `Object` interface, which is used to delete an object. Moreover operations `form`, `drop`, `add` and `remove` are considered for relationships. All language bindings support a `new` operation, allow one to assign a new value to an object attribute, but the Java and Smalltalk bindings do not support explicit deletion. However, most Java-based OODBMSs, such as POET [Sof99] and ObjectStore [Des98], do provide explicit delete operations. In what follows, we refer to a basic instance-oriented data manipulation language consisting of the primitives for creating and deleting objects, and for updating their properties. Primitives are also provided for manipulating multivalued properties. For the sake of simplicity, we do not consider ad-hoc operations for manipulating relationships, rather operations for updating properties apply to both attributes and relationships.

In addition to such basic instance-oriented data manipulation language, we also provide set-oriented data manipulation. The data manipulation primitives we consider are the set-oriented counterpart of those discussed above for creating, updating, and deleting an object. Rather than acting on a single object (instance), they work on a set of objects at-a-time, where this set is determined by the objects satisfying a given condition (query). This is similar to the approach of SQL data manipulation statements.

The update operations we consider are presented in Figure 5.2, expressed in BNF. They are partitioned into instance-oriented[1] and set-oriented update operations; the latter very closely resemble SQL data manipulation statements. Table 5.1 reports the meaning of the nonterminal symbols used in Figure 5.2.

We now briefly discuss set-oriented data manipulation primitives. First of all, these primitives can be applied to generic object collections, that is, to classes, extents, as well as to multivalued properties. Therefore, our primitives are a clear generalization of SQL DML primitives. They provide a uniform syntax whose semantics depends on the arguments to which they are applied. As in SQL and in OQL an alias can be associated with a class; this alias allows one to refer to the object of that class currently under examination in the condition of the statement.

Insert. The effect of the primitive `insert into` c *expr* is the insertion in the collection c of the objects resulting from the evaluation of *expr*. Two cases must, however, be distinguished. If c is a class or an extent, then *expr* must evaluate to a set of struct

[1] Note that for the sake of simplicity we consider instance-oriented update operations whose paths are composed by just one property. The extension to arbitrarily long paths can be easily done.

Nonterminal	Meaning
`var`	variable denoting an object
`class`	class identifier
`property`	property (either attribute or relationship) name
`i_expr`	expression that evaluates to a single value or object (built by means of path expressions, arithmetic operations, and so on)
`s_expr`	OQL query (whose result is either a set of objects or a set of literals)
`cond`	boolean predicate (`where` clause of OQL query)
`extent`	extent identifier
`identifier`	alias identifier

Table 5.1: Nonterminal symbols

literals.[2] The effect of the operation is to create a new object of class c, with which a new object identifier is associated, for each of the literals obtained by the evaluation of *expr*. With each of those newly created objects, the literal obtained from the corresponding evaluation of *expr* is associated as initial state. Attributes of class c for which no value is specified in *expr* are initialized to the *null* value. If, by contrast, c is a multivalued property, then *expr* evaluates to a set of values (either oids or literals), and the effect of the operation is of inserting values resulting from the evaluation of *expr* in c.

Delete. The effect of the primitive `delete from` c `where` F is the removal of the objects satisfying condition F from the collection c to which they belong. If c is a class or extent identifier the objects are also deleted from the database. If the condition is omitted, it is assumed to be `true`, and thus all objects in c are deleted.

Update. The primitive `update` c `set` p `=` *expr* `where` F sets the value of property p[3] to the value to which *expr* evaluates for all the objects satisfying condition F. If the condition is omitted, it is assumed `true`, thus the update is applied to all the objects in c.

Iterate. Finally, the primitive `iterate on` c x `do` u allows one to apply an instance-oriented operation u on the objects x in a collection c, by explicitly iterating on objects in the collection through an *iterator* [CBB+99]. This functionality is already provided in the standard through the explicit definition of an object instance of the `Iterator` class. For the sake of conciseness, however, we consider the proposed primitive as a shorthand.

There are some type constraints for a primitive to be correct. Typing rules for update operations, making those constraints explicit, can be found in Figure 5.3. There, a generic collection type belonging to $\mathcal{CLT}$ is denoted by $coll < \tau >$ where *coll* is the collection

[2]Note that these struct literals must have as components a subset of the class attributes.
[3]Note that p must be a property of objects in collection c.

```
<update> ::  = <i_update> | <s_update>
<i_update> ::  = <var> = new <class> [(<pars>)] |
            <var>.delete() |
            <var>.<property> = <i_expr> |
            <var>.<property>.insert_element(<i_expr>) |
            <var>.<property>.remove_element(<i_expr>)
<pars> ::= <i_expr> | <i_expr>, <pars>
<s_update> ::  = insert into <obj_coll> <s_expr> |
            delete from <obj_coll> [where <cond>] |
            update <obj_coll> set <property> =
                 <s_expr> [where <cond>]
            iterate on <obj_coll> <identifier> do <i_update>
<obj_coll> ::= <class> | <extent> | <class>[[<identifier>]].<path>
<path> ::= <property> [<identifier>] |
            <property>[[<identifier>]].<path>
```

Figure 5.2: Data manipulation primitives proposed for ODMG

type constructor, e.g. `set`, and τ is the inner type. In addition, let $\tau \in \mathcal{OT}$ be an ODMG object type, and p be a property name, then we denote the fact that p is a property of the object type τ as $p \bar{\in} \tau$, whereas we denote the type of property p in τ as $dom(p, \tau)$. Finally, given an object type τ such that the object type external specification body of τ is $(i, attr, rel, meth)$ (cf. Definition 2.8), $stype(\tau)$ denotes the type `struct` representing the set of properties of τ and their types.

Example 5.1 Given a variable `s` declared of type `Studio`, a variable `e` of type `MovieExec` and a variable `m` of type `Movie`, the following are examples of data manipulations on the schema of Figure 5.1.

```
s =new Studio('mgm', 'losangeles');
s.president = e;
s.owns.insert_element(m);
s.delete();
insert into Studio[s].owns
   select m from Movies m where m.producedBy = s.president;
delete from MovieExec where isempty(produces);
delete from Movie.stars s where s.gender = male;
update Movie set filmType = blackAndWhite where year < 1939;
update MovieExec set netWorth=1.10 * netWorth
   where not null manages or count(produces) > 10;
```

$$\frac{X : \tau}{X = \texttt{new}\ \tau} \quad \tau \in \mathcal{OT}_{class}$$

$$\frac{X : \tau,\ i\_expr_i : \tau'_i,\ 1 \leq i \leq m}{X = \texttt{new}\ \tau(i\_expr_1, \ldots, i\_expr_m)} \quad \tau \in \mathcal{OT}_{class},\ p_i \bar{\in} \tau,\ dom(p_i, \tau) = \tau_i,\ \tau'_i \leq_T \tau_i,\ 1 \leq i \leq m$$

$$\frac{X : \bar{\tau},\ i\_expr : \tau'}{X.p = i\_expr} \quad \bar{\tau} \in \mathcal{OT},\ p \bar{\in} \bar{\tau},\ dom(p, \bar{\tau}) = \tau,\ \tau' \leq_T \tau$$

$$\frac{X : \tau,\ i\_expr : \tau''}{X.p.\texttt{insert\_element}(i\_expr)} \quad \tau \in \mathcal{OT},\ p \bar{\in} \tau,\ dom(p, \tau) = coll < \tau' > \in \mathcal{CLT},\ \tau'' \leq_T \tau'$$

$$\frac{X : \tau,\ i\_expr : \tau''}{X.p.\texttt{remove\_element}(i\_expr)} \quad \tau \in \mathcal{OT},\ p \bar{\in} \tau,\ dom(p, \tau) = coll < \tau' > \in \mathcal{CLT},\ \tau'' \leq_T \tau'$$

$$\frac{obj\_coll : \tau,\ s\_expr : \tau'}{\texttt{insert into}\ obj\_coll\ s\_expr} \quad \tau \in \mathcal{OT}_{class}, stype(\tau) = \tau'',\ \tau' \leq_T \tau''$$

$$\frac{obj\_coll : coll < \tau' >,\ s\_expr : \tau''}{\texttt{insert into}\ obj\_coll\ s\_expr} \quad coll < \tau' > \in \mathcal{CLT},\ \tau'' \leq_T \tau'$$

$$\frac{obj\_coll : \tau}{\texttt{delete from}\ obj\_coll\ \texttt{where}\ F} \quad \tau \in \mathcal{OT}_{class}\ or\ \tau = coll < \tau' > \in \mathcal{CLT}$$

$$\frac{obj\_coll : \tau,\ s\_expr : \tau''}{\texttt{update}\ obj\_coll\ \texttt{set}\ p = s\_expr\ \texttt{where}\ F} \quad \tau \in \mathcal{OT},\ p \bar{\in} \tau,\ dom(p, \tau) = \tau',\ \tau'' \leq_T \tau'$$

$$\frac{obj\_coll : \tau,\ i\_update}{\texttt{iterate on}\ obj\_coll\ X\ \texttt{do}\ i\_update} \quad \tau \in \mathcal{OT}\ or\ \tau = coll < \tau' > \in \mathcal{CLT}$$

Figure 5.3: Typing rules for data manipulation primitives proposed for ODMG

```
iterate on MovieExecs e do e.netWorth = select avg(netWorth) from MovieExecs;
iterate on (select from Movies m where (m.filmtype = color and m.year < 1939)
   do m.changeTypeOrYear();
```

◊

The semantics of the above operations is formally specified in Section 5.4. Note that the execution of the set-oriented update primitives is not semantically equivalent to an iterative execution of the corresponding instance-oriented primitives on all the objects in the set, as shown by the following example.

Example 5.2 Referring to the schema of Figure 5.1, the effect of executing:

```
update MovieExec set netWorth = select avg(netWorth) from MovieExecs
```

is different from that of executing:

```
iterate on MovieExecs e do e.netWorth =
select avg(netWorth) from MovieExecs.
```

In the set-oriented update the average is computed once for all the objects affected by the operation, that is, all the objects belonging to class `MovieExec`, and such a value is associated with the `netWorth` attribute of each object. In the instance-oriented update, by contrast, for each object in class `MovieExec` the average is recomputed and then it is associated with the considered object. Thus, after update execution, not all the objects in class `MovieExec` will have assigned the same value to the `netWorth` attribute. ◊

In an object-oriented context, data manipulation can of course be performed also through methods. Method execution, by contrast, can only be carried on in an instance-oriented way. The reason is that set-oriented method execution poses some problems related to non-determinism [ACPvdB95, BGM97, LS93]. Note that, in OQL, set-oriented method invocation is already present, since method invocations can appear in queries, that have a set-oriented semantics. Though it is not explicitly stated in the OQL specification, it seems reasonable to allow only side-effect free methods in queries, that is, methods that do not modify the database state. For side-effect free methods the effect of iterating the execution of methods on all the objects in a set is equivalent to execute the method in a set-oriented way. For methods with side-effects this property does not hold, and thus the need would arise for a support of set-oriented method execution in ODMG. This would require a re-examination of method execution itself, since each statement in the method implementation should be executed in parallel on the objects in the set. This would mean revisiting the semantics of the programming language in which the method is written (C++, Smalltalk, or Java) to accommodate set-oriented execution. Since that would require an extensive modification of the standard, we do not include set-oriented method execution for methods

```
<trig_dcl> :: = trigger <name>
             {before | after } <event> on <obj_coll>
             [referencing {old as <variable> | new as <variable> |
                  oldset as <variable> | newset as <variable>}]
             [for each {instance | statement}]
             [when <condition>]
             <action>
<event> :: = insert | delete | update[of <property_list>] |
             <operation>
<action> :: = <act> | <act>; <action>
<act> :: = <update> | <s_expr> | <variable>.<operation>([<pars>])
```

Figure 5.4: ODL extension for specifying triggers

with side-effects in our model. As shown in Example 5.1, however, method execution can be iterated on elements in a collection.

5.3 A Language for Defining Triggers

The ODL primitive we propose for defining triggers is presented in Figure 5.4. In what follows we first provide some examples to illustrate the language, then we discuss the main aspects in the definition of a trigger, also pointing out the relevant differences with respect to SQL:1999.

5.3.1 Examples

Figure 5.5 shows some trigger examples defined on the schema in Figure 5.1. Trigger `checkDate` checks, upon movie insertions, the constraint that a movie may not be a color movie if it has been produced before 1939. Method `changeTypeOrYear`, invoked in the trigger action, asks the user whether the movie type or year have to be changed in order to restore consistency.

Trigger `cascDelMovieExec` makes sure that every movie executive always appears as either a studio president, a producer of a movie, or both, by propagating the deletion of a movie to its producer, if necessary. Trigger `checkStudio` reacts to updates to a movie, to make sure that if a producer of a movie is also the president of a studio, then he or she must be the president of the studio that produced the movie. Suppose moreover that the average net worth of movie executives must not drop below $500000. If an update violates this

```
trigger checkDate
   after insert on Movie
   referencing new as nm
   for each instance
   when (nm.filmtype = color and nm.year < 1939)
   nm.changeTypeOrYear()

trigger cascDelMovieExec
   after delete on Movie
   referencing oldset as delMovies
   for each statement
   delete from MovieExec e
     where e in delMovies.producedBy and
     isempty(e.produces) and is null e.manages

trigger checkStudio
   after update on Movie
   referencing new as newMovie
   for each instance
   when is not null newMovie.producedBy.manages
   newMovie.ownedBy=newMovie.producedBy.manages

trigger adjustNetWorth
   after update of netWorth on MovieExec
   referencing newset as newS, oldset as oldS
   when (500000 <= select avg(netWorth) from MovieExecs)
   for each statement
   update MovieExec e
     set netWorth = 1.10 · (select avg(netWorth) from MovieExecs)
     where not null manages or count(produces) > 10 and
       ((select netWorth from oldS o where o = e) >
         (select netWorth from newS n where n = e))
```

Figure 5.5: Examples of triggers

```
trigger bothMaleAndFemaleStars
   before delete on Movie.stars
   referencing oldset as oldStars, old as os
   for each instance
   when (not exists(select s from Movie.stars
         where s not in oldStars and s.gender = male) or
        (not exists(select s from Movie.stars
         where s not in oldStars and s.gender = female)
   os.notifyDel()

trigger delStudio
   after delete on Studio.owns
   referencing old as om, new as nm
   for each instance
   when (nm.isempty())
   delete from Studio s
     where s = om.ownedBy
```

Figure 5.5: Examples of triggers (continued)

constraint, we want to restore it by adding the 10% of that average net worth to the net worth of movie executives that are presidents of some studios or have produced more than 10 films, whose net worth has been decreased by the update. This behavior is realized by trigger `adjustNetWorth`. Trigger `bothMaleAndFemaleStars` alerts the user when the last male or female star of a movie is going to be deleted, notifying the user of the deletion through the invocation of method `notifyDel`. Finally, trigger `delStudio` removes the information related to a studio when it owns no more movies.

5.3.2 Activation Time

As in SQL:1999, it is possible to specify whether a trigger must be executed before or after its triggering operation. This option is specified by the `before` and `after` clauses of the trigger definition statement. These clauses are of course mutually exclusive. For `before` triggers the condition is evaluated and the action executed before the triggering operation is performed, whereas for `after` triggers the condition is evaluated and the action executed after the triggering operation is performed. As we will discuss in Subsection 5.3.9 below, there are some restrictions on actions of `before` triggers, namely they are not allowed to modify the database state.

5.3.3 Event

The *event* specifies what causes the rule to be triggered. As triggering events we consider internal events related to database operations: property value updates, object creations, and deletions. The update event can be generic on the class or specific to a property. In the latter case the trigger would react only to updates on the specified property. These are exactly the same as the events considered in SQL:1999. In addition to those events, we also consider method executions as possible events. In order to be as close as possible to SQL:1999, no *composite* events (combinations of events), are supported, and a single event can be specified for each trigger. More than one event per trigger could be specified, and composite events could be considered, though there is an obvious trade-off between the richness of the event language and the efficiency of event detection.

Note that an `insert` event corresponds to the execution of either a `new` or `insert_element` instance-oriented primitive, as well as to a set-oriented `insert` primitive; similarly, a `delete` event corresponds to both instance and set-oriented `delete` and to `remove_element`. Event `update of` p corresponds to any modification of p, either through assignment, `insert_element`, `remove_element` or their set-oriented counterparts. Event `update` corresponds to the modification of any of the properties of the object. Finally, event *op* corresponds to the invocation of *op* (which can only be instance-oriented).

5.3.4 Monitored Collection

The target of a trigger can be any collection of objects. This means that a trigger can monitor the objects in a class, in an extent, or in the set of values of a multivalued property. Of course, for the trigger to be type correct the property or operation eventually appearing in the event must belong to the type of the objects in the collection to which the trigger is targeted.

In case of triggers targeted to multivalued properties (denoted through a path expression) problems related to efficiency of implementations can arise, as illustrated by the following example.

Example 5.3 Referring to the schema of Figure 5.1, let attribute `address` of class `Star` store the address where a movie star lives. A trigger monitoring the event:

```
update of address on Movie.stars
```

should react to the update:

```
update Star set address = ...  where ...
```

if, for any of the `Star` affected by the update, the property `starredIn` is not empty. Thus, for these objects, the `update` event on `Movie.stars` is raised. If an inverse property of `stars` did not belong to the `Star` class, testing whether the star has been the star in some movie would have been very expensive. In this case, the trigger could however have been expressed as:

```
update of address on Star when

exists(select m from Movie m where old in m.stars)
```
◊

An efficient event detection can be achieved if triggers are targeted to multivalued properties expressed through an *invertible path*, that is, such that for each traversed property an inverse property belongs to the schema. Note that in ODMG this holds for relationships whereas it does not hold for attributes. For achieving an efficient evaluation inverse properties could be maintained by the system even if they do not belong to the schema. Since we are not addressing implementation and optimization issues in this thesis, that possibility will not be discussed further.

5.3.5 Trigger Inheritance and Overriding

Since a single event per rule can be specified, and each event is targeted to a collection (`on` clause), each trigger monitors a single collection. Triggers are however inherited, that is, events are propagated across the class inheritance hierarchy. A trigger monitoring the event E on class c, also reacts to occurrence of E on instances of a subclass c' of c. 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 the proper instances of this class. Similarly, if a trigger monitors an event E on the multivalued property $c.p_1.\cdots.p_n$, it also reacts to occurrence of E on the property $o.p_1.\cdots.p_n$ for objects o that are proper instances of a subclass c' of c.

Triggers targeted to classes can also be overridden along the inheritance hierarchy, like methods. Trigger overriding is based on names. Let r be a trigger monitoring an event on a class c, then r can be overridden by the definition of a trigger r' monitoring the same event, on subclass c' of c, with the same name of that of r.[4] Late binding is supported also for triggers, thus, at execution time, for each object affected by the execution of an operation monitored by a trigger, the most specific implementation for the trigger will be chosen.

[4]Note that in overriding a trigger only the condition and the action components can be redefined.

Example 5.4 Referring to the schema of Figure 5.1, consider the following trigger:

```
trigger upd_prod

after update of producedBy on Movie ...
```

It would react also to updates to the `producedBy` property of objects of class `Cartoon`, unless the following redefinition is specified, overriding the trigger above:

```
trigger upd_prod

after update of producedBy on Cartoon ...
```

◊

5.3.6 Transition Variables

As in SQL:1999, the possibility is given of referencing in the condition (as well as in the action) the objects affected by the operation that triggered the rule. This is accomplished through *transition variables* declared in the `referencing` clause of the trigger definition statement. Both the new and the past states of objects affected by the data manipulation statement (triggering event) can be queried. Affected objects can be seen individually (`old` and `new`) or jointly as a "temporary" extent (`oldset` and `newset`). Individual transition variables can only be used in instance-oriented triggers, whereas set transition variables can be used in both instance-oriented and set-oriented after triggers. Moreover, not every transition variable is supported for every kind of event. In particular, the `old` and `oldset` variables can be used in triggers with delete, update or method invocations as events, whereas the `new` and `newset` variables can be used in triggers with insert, update or method invocations as events. Finally, as in SQL:1999, `before` triggers cannot contain set transition variables.

Note that we allow references to both old and new database states in triggers monitoring method invocations as events. This choice is due to the fact that, because of method dynamic extensibility, it is not possible to statically analyze methods to detect whether they update objects rather than simply creating a new object or deleting an object. Thus, a run-time error could occur in case an undefined transition variable is used, e.g. if a `new` variable is used in a trigger reacting to the invocation of a method that simply deleted an object.

5.3.7 Processing Granularity

Trigger execution can be either *instance-oriented* (`for each instance`) or *set-oriented* (`for each statement`). With an instance-oriented execution the trigger is executed once for each object affected by triggering event. The trigger condition is evaluated on each of those objects, and if it holds, the trigger action is executed on the object. By contrast, trigger execution is set-oriented if a trigger is executed once for all the objects affected by the triggering event. The trigger condition is evaluated once on this set of objects, and if it holds the trigger action is executed in a set-oriented way.

The issue of instance-oriented versus set-oriented execution is obviously related to the granularity of the triggering operation. All the four combinations set/instance-oriented data manipulation operations and set/instance-oriented trigger execution are possible. Thus, we can have a set-oriented data manipulation statement that triggers an instance-oriented trigger, as well as an instance-oriented data manipulation statement that triggers a set-oriented trigger (in this case, the set on which the trigger is executed will consist of the singleton containing the object affected by the triggering operation). Note, moreover, that a set-oriented trigger monitoring as event an operation execution is not very meaningful, since we do not consider set-oriented method execution. Thus the trigger will be always executed on a set containing a single object. However, we do not disallow this case by any syntactic restriction.

Finally, note that the processing granularity is orthogonal with respect to the activation time, thus all kinds of triggers illustrated in Table 5.2 are allowed.

5.3.8 Condition

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. They are expressed as OQL conditions (that is, any construct that can appear in an OQL query `where` clause). The `when` clause of the trigger definition statement is optional. If it is missing, the condition is supposed to be `true` and the trigger action is executed as soon as the trigger event occurs.

5.3.9 Action

The *action* is executed when the rule is triggered and its condition is true. Possible actions include the data manipulation statements discussed in Section 5.2, OQL queries, and method invocations. A sequence of actions can be specified, so that the specified actions are sequentially executed.

The action of a set-oriented trigger can contain only a set-oriented data manipulation statement. This restriction is motivated by the fact that only set transition variables can be declared in statement triggers. Thus, there is no way to delete or update an object which is bound to a variable, or, in case of object creation, to declare a variable that is then bound to a newly created object. The action of an instance-oriented trigger, by contrast, can contain an instance-oriented data manipulation statement or a a set-oriented data manipulation statement addressed to a set transition variable. We recall that, since we do not consider set-oriented method execution, a method can be applied to a set of objects only through the `iterate` primitive.

Another important restriction is related to `before` triggers. As in SQL:1999, the action of `before` triggers cannot update the database state, thus, it is restricted to be a sequence only containing queries or side-effect free method invocations. Before triggers are useful, for instance, for testing method preconditions. Note that the restriction to side-effect free method invocations cannot be detected statically, at trigger definition time, since, due to dynamic method extensibility, we cannot decide, when compiling the trigger, whether the invoked method is side-effect free or not. Thus, a run-time error could occur if a method invoked in the action of a before trigger happens to modify the database state.

Note, finally, that when a method invocation appears in a trigger action, object-specific method selection is applied, that is, objects always exhibit their most specific behavior. Consider a trigger r monitoring an event on a class c and invoking in its action an operation op on the objects affected by the event. Consider moreover a subclass c' of c and suppose that operation op is redefined in c'. Rule r is triggered when the event monitored by r occurs both on objects that are proper instances of c and on objects that are proper instances of c'. For objects that are proper instances of c' the most specialized implementation of op (that is, the implementation in class c') is chosen, without taking into account the class which is monitored by the trigger.

Table 5.2 summarizes the main features of triggers.

5.4 Semantics

In this section we define the semantics of triggers. The intuitive idea behind this semantics is the following. When an update operation or a method is considered for execution, the set of rules triggered by the corresponding event is determined, together with the set of objects (that would be) affected by the operation. In the case where there are both instance-level and statement-level triggers on the same operation and object class or multivalued properties for the same time (either `BEFORE` or `AFTER`), the order of execution of triggers is

	event	transition variables	actions the trigger can contain
before instance trigger	insert	`new`	instance and set-oriented actions that do not update the database (queries, side-effect free method invocations)
	delete	`old`	
	update	`new`, `old`	
	method invocation	`new`, `old`**	
after instance trigger	insert	`new`, `newset`	instance-oriented updates and set-oriented updates
	delete	`old`, `oldset`	
	update	all	
	method invocation	all**	
before statement trigger	insert	-	set-oriented actions that do not update the database (queries, side-effect free method invocations)
	delete	-	
	update	-	
	*method invocation**	-	
after statement trigger	insert	`newset`	set-oriented updates
	delete	`oldset`	
	update	`newset`, `oldset`	
	*method invocation**	`newset`, `oldset`**	

* Allowed, but not very meaningful (method execution always is instance-oriented).

** Both references to old and new states are allowed, however, run-time errors may arise.

Table 5.2: Different kinds of triggers

determined as follows: if the triggers are *before* triggers, then the statement-level triggers precede the instance-level triggers; if, by contrast, the triggers are *after* triggers, then the instance-level triggers precede the statement-level triggers. Thus, the set of triggered rules is partitioned in four different sequences, one for set-oriented before triggers (T_{BS}), one for instance-oriented before triggers (T_{BI}), one for instance-oriented after triggers (T_{AI}) and one for set-oriented after triggers (T_{AS}). Each sequence is ordered with respect to the creation time. Then, each trigger in the set is executed, according to the different execution semantics specified for different triggers. That is:

- first, all triggers in T_{BS} are executed;
- then, for each object in the set of affected objects first, all triggers in T_{BI} are executed, then the original operation and finally all triggers in T_{AI};
- finally all triggers in T_{AS} are executed.

Trigger execution consists of evaluating the condition, and, if it holds, in sequentially executing the operations in the action of the trigger. Thus, trigger execution consists of updates, which may in turn trigger other rules. Note that trigger execution in our approach (as in SQL:1999) is recursive. Thus these triggers are immediately processed before completing the execution of already triggered rules. Clearly, the possibility of infinite rule processing due to chains of active rules that trigger each other exists; techniques and tools for detecting the possible sources of non-termination in a rule set have been developed [CW96].

5.4.1 Preliminaries

Before formalizing trigger semantics, we need to introduce some preliminary definitions. In the following we consider a set TN of trigger names. In addition, we recall that $\mathcal{V}$ denotes the set of ODMG values; $\mathcal{OI}$ denotes a set of all possible oids;[5] $\mathcal{CI}$ denotes a set of class names. We refer to Table A.1 for a summary of the symbols most frequently used in this chapter. Moreover, given a set S, 2^S denotes the powerset of S, whereas L^S denotes the set of all possible sequences with elements in S and $\circ$ denotes sequence concatenation.

Let *Rule* denote the set of triggers defined for a database. Each trigger is modelled as tuple (T, c, e, C, A, t, g, v), where T is the name, c is the target (either a class or a multivalued property denoted through a path expression[6]), e is the event, C is the condition, A is the

[5] Note that $\mathcal{OI} \subseteq \mathcal{V}$.

[6] In our language, the target can also be an extent. However, since the semantics of associating a trigger with an extent is the same of associating it with the corresponding class, we do not consider this option in specifying the semantics.

action, t is the time, that is, either `before` or `after`, g is the processing granularity, that is, either `instance` or `statement`, v is the 4-tuple of transition variables (one component for each kind -`old`, `new`, `oldset`, `newset`- and null where variables are not declared). Because of trigger overriding, there can be several triggers with the same name belonging to set *Rule*. However, some restrictions are imposed on overriding, ensuring that, given two triggers (T, c, e, C, A, t, g, v) and $(T', c', e', C', A', t', g', v')$ such that $T = T'$, then $e = e'$, $t = t'$, $g = g'$, $v = v'$ and c and c' must be related by inheritance.

The notion of database state we consider in giving the semantics is the one presented in Chapter 2 in Definition 2.22. We recall that a database state is a pair $S = (\pi^+, \nu)$ where: π^+ is the function defined in Definition 2.6 and for each $o \in OBJ$, $o = (id, N, l, (a_1 : v_1^a, \ldots, a_n : v_n^a), (r_1 : v_1^r, \ldots, r_m : v_m^r), c)$, $\nu(id)$ returns the state of the object, that is, the value of its properties; then $\nu(id) = [a_1 : v_1^a, \ldots, a_n : v_n^a, r_1 : v_1^r, \ldots, r_m : v_m^r]$.

Note that in *A*_ODMG the set *Rule* is part of the ODMG schema. We now introduce the notion of *path instantiation* [BK89], formalized as follows.

Definition 5.1 (*Path Instantiation*). Let $c.p_1.\cdots.p_n$ be a path, with $c \in \mathcal{CI}$ class name and $p_1, \ldots, p_n$ property names. Moreover, let o_i, $i \in [0, n-1]$, be objects and $O_n \subseteq \mathcal{OI}$ be a set of objects. We say that the sequence $o_0.o_1.\cdots.o_{n-1}.O_n$ is an instantiation of path $c.p_1.\cdots.p_n$ if the following conditions hold:

- o_0 is an instance of class c, that is, $o_0 \in \pi(c)$;
- o_i is the value of property p_i of object o_{i-1}, for $i \in [1, n-1]$;
- O_n is the value of property p_n of object o_{n-1}. □

Note that, according to the definition above, given a path $c.p_1.\cdots.p_n$, the properties $p_1, \ldots, p_{n-1}$ cannot be multivalued. This is in agreement with the OQL approach, where path navigation across multivalued properties is not allowed. By contrast, property p_n is multivalued, thus O_n is a set of objects rather than a single object.

We say that a path instantiation $o_0.o_1.\cdots.o_{n-1}.O_n$ satisfies a formula F in a database state S, if the formula $F[x_i/o_i]$ evaluates to true in S, where x_i is the alias identifier corresponding to position i (that is, the path expression has the form $c[x_0].p_1[x_1].\cdots.p_{n-1}[x_{n-1}].p_n x_n$) and $F[var/obj]$ denotes the formula F in which identifier *var* is instantiated to object *obj*.

The semantic functions we use are illustrated in Table 5.4. For each function we report the signature and the meaning. Note that in giving the semantics we refer to semantic domains as the set of syntactically well-formed language constructs, that is, constructs which meet

Semantic Domain	Meaning
Expr	set of both well-formed single-valued (`i_expr`) and set-valued (`s_expr`) expressions
Update	set of both well-formed set-oriented and instance-oriented updates
Condition	set of well-formed trigger conditions
Action	set of well-formed trigger actions
State	set of possible database states

Table 5.3: Semantic domains

the static constraints and for which the semantics can be defined. The semantics domains we refer to are described in Table 5.3. Semantic function $\mathcal{E}$ models expression evaluation, that is carried on against a database state and produces a set of values. Update execution is modelled by semantic function $\mathcal{U}$, that transforms a database state in a new database state, and is specified relying on an auxiliary semantic function $\mathcal{A}$, of the same functionality, which models update execution as if no trigger was specified. Trigger semantics is specified through semantic function $\mathcal{T}$ that given a database state and a set of objects (those affected by the update we are reacting to) produces a new database state. Note that we model as effects of the execution of a trigger only the changes in the database; indeed, we are interested in the changes on data rather than in outputs given to the user. To capture this aspect an additional parameter, such as a *display*, should have been specified for semantic function $\mathcal{T}$. In particular, we remark that *before* triggers, that in our language are disallowed to update the database state, have no observable effect according to our semantics. Semantic function $\mathcal{T}$ relies on function $\mathcal{C}$, modelling the execution of a trigger condition and action component, which also transforms a database state in a new database state. The remaining semantic functions, namely, $\mathcal{R}_s, \mathcal{R}_i, \mathcal{I}$ allow us to model the execution of a set of triggers triggered by an update operation, which affected a set of objects, according to the different execution options (either set-oriented or instance-oriented).

5.4.2 Update Primitive Semantics

In this section we describe the semantics of updates, disregarding the reactive process. Such semantics is the basis on which trigger semantics is defined. The semantics of updates is modelled by function $\mathcal{A}$. Moreover, in order to define update semantics, semantic function $\mathcal{E}$ is used. In case of single-valued expressions (`i_expr`) the evaluation produces a singleton, whereas in case of set-valued expressions (`s_expr`) the evaluation produces a set of values. In particular, in case of set-valued expressions used in the `insert into` state-

Semantic Function	Meaning
$\mathcal{E} : Expr \rightarrow State \rightarrow 2^{\mathcal{V}}$	expression evaluation
$\mathcal{A} : Update \rightarrow State \rightarrow State$	update exec. without triggers
$\mathcal{U} : Update \rightarrow State \rightarrow State$	update exec.
$\mathcal{R}_s : L^{TN} \rightarrow (2^{\mathcal{OI}} \times State) \rightarrow State$	set-oriented trigger exec.
$\mathcal{R}_i : L^{TN} \circ Update \circ L^{TN} \rightarrow (2^{\mathcal{OI}} \times State) \rightarrow State$	instance-oriented trigger exec.
$\mathcal{I} : L^{TN} \circ Update \circ L^{TN} \rightarrow (\mathcal{OI} \times State) \rightarrow State$	iterates instance-oriented triggers on objects
$\mathcal{T} : TN \rightarrow (2^{\mathcal{OI}} \times State) \rightarrow State$	exec. of a single trigger
$\mathcal{CA} : Condition \times Action \rightarrow (2^{\mathcal{OI}} \times State) \rightarrow State$	exec. of the body of a single trigger

Table 5.4: Semantic functions

ment addressed to a class or an extent, the resulting set is a set of struct values.[7] We do not present here the formal definition of function $\mathcal{E}$ since it is not particularly meaningful in this context.

The semantics of instance-oriented operations is presented in Table 5.5. Note that variable O is an overloaded symbol since it is used to denote both an object-denoting variable in the statements and an object identifier (the oid to which the variable is bound) in giving the semantics. The semantics of set-oriented operations is presented in Table 5.6. In such a table, given a condition F and an object o, $F[o]S = true$ means that the instantiation of F with respect to o evaluates to true in state S.

5.4.3 Trigger Semantics

In defining trigger semantics, first we have to establish, given an event, which is the set of rules triggered by its occurrence.

Definition 5.2 (*Event Instance*). Let u be an update operation, c be the object collection this update is related to, and O be the set of oids of the objects affected by the update. Moreover, let e be the event corresponding to u, then the triple $\langle e, c, O \rangle$ is called event instance. □

[7]Note that in case of OQL queries where the `DISTINCT` clause is omitted, the evaluation result is a bag, thus objects with the same state, but with different oids can be created.

$\mathcal{A}[\![\, O = \texttt{new}\ \bar{c}\ (i\_e_1, \ldots, i\_e_n) \,]\!]\, S = \langle \pi'^{+}, \nu' \rangle$, where:

$$\pi'^{+}(c) = \begin{cases} \pi^{+}(c) \cup \{o^*\} & \text{if } c = \bar{c} \\ \pi^{+}(c) & \text{if } c \neq \bar{c} \end{cases} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \neq o^* \\ v_k & \text{if } o = o^* \text{ and} \\ & \mathcal{E}[\![\, i\_e_k \,]\!]\, S = \{v_k\}, k \in [1, n] \end{cases}$$

where o^* is a newly generated oid, to which variable O gets bound

$\mathcal{A}[\![\, O.\ \texttt{delete}() \,]\!]\, S = \langle \pi'^{+}, \nu' \rangle$, where:

$$\pi'^{+}(c) = \begin{cases} \pi^{+}(c) \setminus \{O\} & \text{if } O \in \pi^{+}(c) \\ \pi^{+}(c) & \text{otherwise} \end{cases} \qquad \nu' = \nu$$

$\mathcal{A}[\![\, O.p = i\_e \,]\!]\, S = \langle \pi'^{+}, \nu' \rangle$, where:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \neq O \text{ or } p_k \neq p \\ v & \text{if } o = O, p_k = p \text{ and } \mathcal{E}[\![\, i\_e \,]\!]\, S = \{v\} \end{cases}$$

$\mathcal{A}[\![\, O.p.\texttt{insert\_element}(i\_e) \,]\!]\, S = \langle \pi'^{+}, \nu' \rangle$, where:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \neq O \text{ or } p_k \neq p \\ \nu(o).p_k \cup \{v\} & \text{if } o = O, p_k = p \text{ and } \mathcal{E}[\![\, i\_e \,]\!]\, S = \{v\} \end{cases}$$

$\mathcal{A}[\![\, O.p.\texttt{remove\_element}(i\_e) \,]\!]\, S = \langle \pi'^{+}, \nu' \rangle$, where:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \neq O \text{ or } p_k \neq p \\ \nu(o).p_k \setminus \{v\} & \text{if } o = O, p_k = p \text{ and } \mathcal{E}[\![\, i\_e \,]\!]\, S = \{v\} \end{cases}$$

Table 5.5: Semantics of instance-oriented operations

$\mathcal{A}[\![\,\texttt{insert into}\ \bar{c}\ s\_e\,]\!]\,S = \langle \pi'^{+}, \nu' \rangle$, where:

- if $\bar{c}$ is a class name, then:
let $\mathcal{E}[\![\,s\_e\,]\!]\,S = \{v_1, \ldots, v_n\}$, then being $NEW\_OID = \{o_1, \ldots, o_n\}$
the set of newly generated oids:

$$\pi'^{+}(c) = \begin{cases} \pi^{+}(c) \cup NEW\_OID & \text{if } c = \bar{c} \\ \pi^{+}(c) & \text{otherwise} \end{cases} \qquad \nu'(o) = \begin{cases} \nu(o) & \text{if } o \notin NEW\_OID \\ v_i & \text{if } o = o_i \text{ and } v_i \in \mathcal{E}[\![\,s\_e\,]\!]\,S \end{cases}$$

- if $\bar{c} = \hat{c}.p_1.\cdots.p_n$ is a path denoting a multivalued property, then:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p = \begin{cases} \nu(o).p \cup \mathcal{E}[\![\,s\_e\,]\!]\,S & \text{if } o.o_1.\cdots.o_{n-1}.O_n \text{ is an instantiation of } \hat{c}.p_1.\cdots.p_n \\ & \text{satisfying the } \texttt{where} \text{ clause of } s\_e \text{ in } S, \\ & o = o_{n-1} \text{ and } p = p_n \\ \nu(o).p & \text{otherwise} \end{cases}$$

$\mathcal{A}[\![\ \texttt{delete from}\ \bar{c}\ \texttt{where}\ F\,]\!]\,S = \langle \pi'^{+}, \nu' \rangle$, where:

- if $\bar{c}$ is a class name, then:

$$\pi'^{+}(c) = \begin{cases} \pi^{+}(c) \setminus \{o \mid o \in \pi^{+}(c) \wedge F[o]S = true\} & \text{if } \bar{c} = EXTENDS(c) \\ \pi^{+}(c) & \text{otherwise} \end{cases} \qquad \nu' = \nu$$

- if $\bar{c} = \hat{c}.p_1.\cdots.p_n$ is a path denoting a multivalued property, then:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p = \begin{cases} \nu(o).p \setminus O_n & \text{if } o.o_1.\cdots.o_{n-1}.O_n \text{ is an instantiation of } \hat{c}.p_1.\cdots.p_n \\ & \text{satisfying } F \text{ in } S,\ o = o_{n-1} \text{ and } p = p_n \\ \nu(o).p & \text{otherwise} \end{cases}$$

Table 5.6: Semantics of set-oriented operations

$\mathcal{A}\,[\![\,\texttt{update}\ \bar{c}\ \texttt{set}\ p = s\_e\ \texttt{where}\ F\,]\!]\,S = \langle \pi'^{+}, \nu' \rangle$, where:

- if $\bar{c}$ is a class name, and O is the set $O = \{o' \mid o' \in \pi(\bar{c})$ and $F[o']S = true\}$, then:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \notin O \text{ or } p_k \neq p \\ \mathcal{E}\,[\![\, s\_e \,]\!]\, S & \text{if } o \in O \text{ and } p_k = p \end{cases}$$

- if $\bar{c} = \hat{c}.p_1.\cdots.p_n$ is a path denoting a multivalued property, and $o.o_1.\cdots.o_{n-1}.O_n$ is an instantiation of $\hat{c}.p_1.\cdots.p_n$ satisfying F in S, then:

$$\pi'^{+} = \pi^{+} \qquad \nu'(o).p_k = \begin{cases} \nu(o).p_k & \text{if } o \notin O_n \text{ or } p_k \neq p \\ \mathcal{E}\,[\![\, s\_e \,]\!]\, S & \text{if } o \in O_n \text{ and } p_k = p \end{cases}$$

$\mathcal{A}\,[\![\ \texttt{iterate on}\ \bar{c}\ x\ \texttt{do}\ u\,]\!]\,\mathrm{S} = \mathcal{A}\,[\![\, u[x/o_n] \,]\!]\,(\mathcal{A}\,[\![\, u[x/o_{n-1}] \,]\!]\,(...(\mathcal{A}\,[\![\, u[x/o_2] \,]\!]\,(\mathcal{A}\,[\![\, u[x/o_1] \,]\!]\,S))))$

where $\bar{c} = \{o_1, o_2, \ldots, o_{n-1}, o_n\}$ and $u[x/o_i]$, $i \in [1, n]$, denotes the instance-oriented operation u where variable x has been bound to object o_i

Table 5.6: Semantics of set-oriented operations (continued)

Example 5.5 The event corresponding to the action `insert into` c *expr*, is *insert* and a corresponding event instance is $\langle \texttt{insert}, c, O\rangle$, where O denotes the set of the oids of the created objects. $\diamond$

For the sake of simplicity, we will often use the word event to denote event instances, when the meaning is clear from the context.

To establish which rules are triggered by an update operation, we must take into account that an operation addressed to a class c can also affect instances of subclasses of c. Thus, for each oid affected by the operation, all the rules monitoring that event on the classes of which the oid is instance must be triggered. Similarly, an update addressed to a multivalued property $c.p_1.\cdots.p_n$ can also affect instances of subclasses of c. To determine the rules to be triggered in that case, we need to inversely traverse the path.

Definition 5.3 (*Set of Rules Triggered by an Update*). Let u be an update, and let $\mathbf{e}$ $= \langle e', c_e, O\rangle$ be its associated event instance. The set of rules triggered by u, denoted by $trig(u)$, is defined as a set of rule names $\{T_1, \ldots, T_n\}$, such that for each T_i, $i \in [1, n]$, a rule $r = (T_i, c, e, C, A, t, g, v)$ exists in *Rule* and either $e = e'$ or $e =$`update` and $e' =$`update(`p`)`. Moreover one of the following conditions must hold:

1. c is a class name and $\exists o \in O$ such that $o \in \pi(c)$;
2. c and c_e are multivalued properties such that $c = c_1.p_1.\cdots.p_n$, $c_e = c_2.p_1.\cdots.p_n$ and $\exists o \in O$ such that a path instantiation $o_0.o_1.\cdots.o_{n-1}.O_n$ of c_e exists such that $o \in O_n$ and $o_0 \in \pi(c_1)$.[8] $\square$

The following definition revisits the semantics of update operations, taking into account trigger execution. In what follows, we denote a sequence of elements as $el_1; \ldots; el_n$ which degenerates in the empty sequence when $n = 0$. The definition refers to the semantic functions modeling trigger execution, $\mathcal{R}_s$ and $\mathcal{R}_i$, that will be specified in Definitions 5.5 and 5.6 below.

Definition 5.4 (*Semantic Function $\mathcal{U}$*). Given a sequence of updates $u_1; \ldots; u_n$, and a database state S the semantic function $\mathcal{U}$ is defined as follows.

- If $n = 0$, $\mathcal{U}\,[\![\, u_1; \ldots; u_n \,]\!]\, S = S$.

[8]Note that this can happen if $c_2 = EXTENDS(c_1)$ and it surely happens if $c_1 = EXTENDS(c_2)$.

- If $n = 1$, let:

  - $trig(u_1) = \{T_{BS}, T_{BI}, T_{AI}, T_{AS}\}$ be the sequence of rules triggered by u_1 (cfr. Definition 5.3) partitioned in different sequences according to processing granularity and execution modality, each one ordered by the creation time;
  - $O = \{o_1, \ldots, o_m\}$ be the set of objects affected by the execution of u_1;

  then:

  $\mathcal{U} [\![u_1]\!] S = \mathcal{R}_s [\![T_{AS}]\!] O(\mathcal{R}_i [\![T_{BI}; u_1; T_{AI}]\!] O(\mathcal{R}_s [\![T_{BS}]\!] OS))$.

- If $n > 1$, then $\mathcal{U} [\![u_1; \ldots; u_n]\!] S = \mathcal{U} [\![u_2; \ldots; u_n]\!] (\mathcal{U} [\![u_1]\!] S)$. $\square$

We can now specify the semantics of set-oriented and instance-oriented trigger execution. In specifying the semantics, the semantic function $\mathcal{T}$ is used. Intuitively, such function, which will be formally defined in Definition 5.8 below, given a trigger T, returns the database state after the execution of T.

Definition 5.5 (*Semantic Function $\mathcal{R}_s$*). Let $T_1; \ldots; T_n$ be a sequence of triggers, O be a set of objects, S be a database state, then:

$$\mathcal{R}_s [\![T_1; \ldots; T_n]\!] OS = \begin{cases} S & \text{if } n = 0 \\ \mathcal{R}_s [\![T_2; \ldots; T_n]\!] O(\mathcal{T} [\![T_1]\!] OS) & \text{otherwise} \end{cases}$$

$\square$

Definition 5.6 (*Semantic Function $\mathcal{R}_i$*). Let $P_1; \ldots; P_n \in L^{TN} \circ \mathit{Update} \circ L^{TN}$ be a sequence of instance-oriented triggers and updates, where only one of the elements is an update, let $O = \{o_1, \ldots, o_m\}$ be a set of objects, and let S be a database state, then:

$$\mathcal{R}_i [\![P_1; \ldots; P_n]\!] \{o_1, \ldots, o_m\} S =$$

$$\begin{cases} S & \text{if } m = 0 \\ \mathcal{I} [\![P_1; \ldots; P_n]\!] o_1 S & \text{if } m = 1 \\ \mathcal{R}_i [\![P_1; \ldots; P_n]\!] \{o_2, \ldots, o_m\}(\mathcal{I} [\![P_1; \ldots; P_n]\!] o_1 S) & \text{otherwise} \end{cases}$$

$\square$

In the execution of instance-oriented triggers, the execution is iterated over the objects in the set. This is modelled by semantic function $\mathcal{I}$ defined below. In the following definition, $conv(P, o, S)$ denotes P itself if P is an instance-oriented update, whereas it denotes the instance-oriented version of P on object o, if P is a set-oriented update. We do not provide a complete definition of *conv* since it does not add any meaningful contribution. The idea is that, given a set-oriented data manipulation statement P, a set of objects O, and a database state S, we can obtain for each object o in O an instance-oriented data manipulation statement $conv(P, o, S)$ on o modelling the actual update executed on o by the execution of P in S. Consider for instance the set-oriented update

$P =$ `update MovieExec set netWorth = select avg(netWorth) from MovieExecs`

and a database state S. Let $\mathcal{E}\,[\![\,\texttt{select avg(netWorth) from MovieExecs}\,]\!]\,S = n$. Then, $conv(P, o, S)$ is o.`netWorth = `n, for each $o \in \pi($`MovieExec`$)$.

Definition 5.7 (*Semantic Function* $\mathcal{I}$). Let $P_1; \ldots; P_n \in L^{TN} \circ Update \circ L^{TN}$ be a sequence of instance-oriented triggers and updates, where only one of the elements is an update, let o be an object, and let S be a database state, then:

$$\mathcal{I}\,[\![\,P_1; \ldots; P_n\,]\!]\,oS = \begin{cases} S & \text{if } n = 0 \\ \mathcal{T}\,[\![\,P\,]\!]\,\{o\}S & \text{if } P \text{ is a trigger and } n = 1 \\ \mathcal{A}\,[\![\,conv(P, o, S)\,]\!]\,S & \text{if } P \text{ is an update and } n = 1 \\ \mathcal{I}\,[\![\,P_2; \ldots; P_n\,]\!]\,o(\mathcal{I}\,[\![\,P_1\,]\!]\,oS) & \text{otherwise} \end{cases}$$

□

We now specify the semantics of the execution of a single trigger through function $\mathcal{T}$. In the definition of the semantics of triggers a fundamental problem is to establish which trigger implementation has to be considered for each object. Our language indeed supports *trigger overriding*, that is, at run-time, for each object, the most specific trigger implementation is executed. In what follows d denotes the trigger *dispatching* function, such that, given an object o and a trigger T, $d(o,\ T){=}c$ where c is the class whose implementation for T is selected for execution. This class is the most specific among the ones to which object o belongs providing a definition of trigger T, that is, $c = min_{EXTENDS}\{\bar{c} \mid o \in \pi(\bar{c}) \wedge (T, \bar{c}, e, C, A, t, g, v) \in Rule\}$. Trigger overriding, however, complicates the semantics of triggers as we discuss in what follows. Suppose rule T triggered on the set of objects $O = \{o_1, \ldots, o_m\}$, then the set O can be partitioned into several disjoint subsets of objects according to the most specific class of each object in which the trigger is re-defined. Given a set of objects $O = \{o_1, \ldots, o_m\}$, the partitioning $P(O,T) = \{O_1 \ldots, O_k\}$ of O is such that, for $j \in [1, k]$, $O_j = \{o_i \mid o_i \in O \text{ and } d(o_i, T) = c_j\}$.[9]

Definition 5.8 (*Semantic Function* $\mathcal{T}$). Given a trigger name T, and a set of objects O, let

[9] Note that objects in the set O belonging to a class in which trigger T is not defined, will not appear in $P(O,T)$.

$P(O,T) = \{O_1, \ldots, O_k\}$ be the partitioning of the set of objects O with respect to T. Moreover, let C_j, A_j be the condition and action of trigger T in class c_j related to partition O_j, $j \in [1,k]$ and S be a database state, then:

$$\mathcal{T}\,[\![\, T \,]\!]\, O_1 \cup \ldots \cup O_k S = \begin{cases} S & \text{if } k = 0 \\ \mathcal{T}\,[\![\, T \,]\!]\, O_2 \cup \ldots \cup O_k(\mathcal{CA}\,[\![\, C_1, A_1 \,]\!]\, O_1 S) & \text{otherwise} \end{cases}$$

□

Note that according to the definition of function $\mathcal{T}$ given above, different database states can be obtained depending on the order in which the different sets of objects in the partition are considered. A possible approach could be that of considering these sets in the order corresponding to the inheritance hierarchy. Finally, we specify the semantics of the execution of the body of a trigger.

Definition 5.9 (*Semantic Function* $\mathcal{CA}$). Let C and A be a trigger condition and action, respectively, O be a set of objects and S be database state, then:

$$\mathcal{CA}\,[\![\, C, A \,]\!]\, OS = \begin{cases} S & \text{if } C \text{ evaluates to } false \text{ in state } S \\ \mathcal{U}\,[\![\, A \,]\!]\, S & \text{if } C \text{ evaluates to } true \text{ in state S} \end{cases}$$

□

Note that in both the evaluation of condition C and in the execution of action A transition variables have been properly bound.[10]

5.5 Concluding Remarks

In this chapter we have proposed an extension of the ODMG data model with reactive capabilities. In particular, our proposal is based on an extension of the SQL:1999 trigger language suitable for the ODMG model. Our proposal, thus, is backward compatible with the SQL:1999 proposal, which can be expressed as a subset of our language. The definition of a trigger model for ODMG has involved the investigation of several related issues: we have revisited data manipulation in ODMG, considering both set-oriented and instance-oriented computations, and we have addressed the problems of trigger inheritance

[10] Actually, to properly evaluate transition variables two additional arguments are needed (the state before the execution of the update and the whole set of objects affected by the update). However, since we do not detail the binding, we have omitted these parameters not to burden the notation.

and overriding, which is not considered in SQL:1999. The formal semantics of our trigger language has also been developed in the thesis. The ideas developed in the thesis are also applicable to object-relational DBMSs [CCN$^+$99, Ora97] and to DBMSs storing XML documents [Des99].

Conclusions

If we reason about the evolution of database systems, we observe a progressive endorsement of object-orientation. This fact is proved by two fundamental standards recently published: ODMG, the standard for purely object-oriented DBMS, and SQL:1999, the standard for object-relational DBMS. ODMG offers to OODBMS vendors the chance to increase their competition with respect to RDBMS vendors because of the definition of a standard model which can be regarded as the SQL of OODBMSs. ODMG contains many interesting ideas, but it lacks a formalization in order to be fully understood and managed.

On the other hand, the relational data model and the object-oriented data models reveal a lack of integration between the data and the semantics they have. This problem is less severe in object-oriented databases where the data include the definition of the operations for manipulating them. However, object-oriented data models still lack some capabilities, such as time management and active behavior. The importance of standards is well-known in database field. Data stored in electronic format should last for years. If one totally depends on specific applications to manage data, effective use of data by users and applications could be severely hindered. Thus, the extension of the ODMG object model with temporal and active capabilities is a very relevant topic of great current interest.

In this thesis we have first provided a comprehensive formalization of the ODMG object model. In our formalization, fundamental notions such as classes, interfaces and objects are precisely defined. All the peculiarities of the model such as the two inheritance relationships, ISA and EXTENDS, relationships among objects, method overloading, and so on are clearly formalized and many interesting properties have been devised. Constraints on the inheritance hierarchies are established in order to be correct with respect to ODMG schema definition. We recall that the aim of the ODMG group was to define portable schemas. In this thesis the notions of schema and database over a given schema are also formalized.

Then, in this thesis we have formally defined a temporal extension of the ODMG object model supporting multiple granularities. In handling temporal information, the need often arises of managing temporal data expressed according to different granularities. Those data should in many cases be compared and used by the same application program. A temporal data model should thus offer the possibility of handling multiple temporal granularities. In addition, in this thesis we have revisited on a formal basis issues related to inheritance, type refinement, and substitutability in a temporal object model supporting multiple granularities. We have considered how multi-granularity temporal objects can be queried and we have described how the proposed model has been implemented.

Finally, in this thesis we have proposed an extension of the ODMG data model with reactive capabilities. In particular, we have proposed an extension of the SQL:1999 trigger language suitable for the ODMG model. Our proposal, thus, is backward compatible with the SQL:1999 proposal, which can be expressed as a subset of our language. The definition of a trigger model for ODMG has involved the investigation of several related issues: we have revisited data manipulation in ODMG, considering both set-oriented and instance-oriented computations, and we have addressed the problems of trigger inheritance and overriding, which is not considered at all in SQL:1999. The formal semantics of our trigger language has also been developed in the thesis.

The work presented in the thesis can be extended along several directions.

For what concerns the temporal extension of the ODMG object model, we are extending the model to support object migration, that is, the possibility for an object to dynamically change class. Though the ODMG object model does not support migration, we believe that this possibility should be offered in a temporal context. Moreover, we are extending the ideas on navigational access to objects, developing a full query language, as an extension of ODMG OQL, for the proposed data model. Finally, we are extending our prototype implementation in several respects. The extensions under development include an improvement of the graphical user interface and the development of ad-hoc index structures well-suited to our data and our navigational access modalities.

For what concerns the future work on the active extension of the ODMG object model, first of all, the trigger language can be enriched in many respects. For instance, more than one event per trigger could be specified, and composite events could be considered, though there is an obvious trade-off between the richness of the event language and the efficiency of event detection. Moreover, different trigger activation times could be considered. In the current proposal after triggers are all activated (if their monitored event occurred) immediately after the execution of the triggering operation; it could be possible to have after triggers whose activation is *deferred* at transaction commit. This is useful for rules that enforce integrity constraints, since a transaction may execute several operations that violate a constraint, but the transaction may restore the constraint before it reaches its commit point. The design of an architecture for the reactive component and the development of an implementation of our extension will also be investigated.

Finally, we believe that the integration of the temporal and active extensions we have proposed in this thesis is a very interesting issue, though very complex, to be investigated.

The areas of temporal and active object database systems have been 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 for providing the kind of powerful database processing environment that will be required by database applications of the future. Thus, we believe that the extension of the ODMG object model with active and temporal features will be an important step towards a new generation of powerful data management systems. Nowadays, many applications need both temporal and active features. For example, we can think of a booking flight system. In a system like that, the history of flights purchased by a customer should be kept in the database for marketing proposals, such as for sending advertisements according to his/her travel choices. At the same time the system will be very complex and much of the effort will be done to monitor the system status, activity which could be performed by an active component of the system itself.

Bibliography

[ABD+89] 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.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[ACPvdB95] 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*, pages 208–218, 1995.

[AG98] K. Arnold and J. Gosling. *The Java Programming Language Second Edition*. Addison-Wesley, 1998.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Ala97] S. Alagić. The ODMG Object Model: Does it Make Sense? In *Proc. Twelfth International Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 253–270, 1997.

[Ala99a] S. Alagic. A Family of the ODMG Object Models. In Johann Eder, Ivan Rozman, and Tatjana Welzer, editors, *Advances in Databases and Information Systems, Third East European Conference, ADBIS'99*, volume 1691 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 1999.

[Ala99b] S. Alagić. O2 and The ODMG Standard: Do They Match? *Theory and Practice of Object Systems*, 5(4):239–247, 1999.

[Ala99c] S. Alagić. Type Checking OQL Queries in the ODMG Type System. *ACM Transactions on Database Systems*, 24(3):319–360, 1999.

[AM95] M. Atkinsons and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[AMC93] 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.

[And82] T. L. Anderson. Modeling Time at the Conceptual Level. In *Proc. of the International Conference on Databases: Improving Usability and Responsiveness*, Israel: Academic Press, pages 273–297, 1982.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1):68–88, 1996.

[BBKZ93] 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.

[BCC+95] 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.

[BDES98] C. Bettini, C.E. Dyreson, W.S. Evans, and R.T. Snodgrass. A Glossary of Time Granularity Concepts. In *Temporal Databases: Research and Practice*, number 1399 in Lecture Notes in Computer Science, pages 406–413. Springer-Verlag, 1998.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan-Kaufmann, 1992.

[Ber88] M. Bertrand. *Objected-oriented software construction.* International Series in Computer Science. Prentice Hall, 1988.

[BFG96] 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. Springer-Verlag, 1996.

[BFG97] 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.

[BFG98] E. Bertino, E. Ferrari, and G. Guerrini. Navigational Access in a Temporal Object Model. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):656–665, 1998.

[BFGM98] E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extending the ODMG Object Model with Time. In E. Jul, editor, *Proc. Twelfth European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 41–66. Springer-Verlag, 1998. Belgium.

[BFP+95] M.L. Barja, A.A.A. Fernandes, N.W. Paton, M.H. Williams, A. Dinn, and A.I. Abdelmoty. Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System. *Information Systems*, 20(3):185–211, 1995.

[BG98] E. Bertino and G. Guerrini. Extending the ODMG Object Model with Composite Objects. In *Proc. Thirteenth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 259–270, 1998.

[BG00] E. Bertino and G. Guerrini. OODBMS History and Concepts. In A.B. Chaudhri and R. Zicari, editors, *Succeeding with Object Databases: A Practical Look at Today's Implementations with Java and XML*, pages 3–26. John Wiley & Sons, 2000.

[BGM97] 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*, number 1341 in Lecture Notes in Computer Science, pages 193–210. Springer-Verlag, 1997.

[BGM00] E. Bertino, G. Guerrini, and I. Merlo. Trigger Inheritance and Overriding in Active Object Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):588–608, 2000.

[BJW00] C. Bettini, S. Jajodia, and X.S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning.* Springer-Verlag, 2000.

[BK89] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[BM91] 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.

[BM92] E. Bertino and L.D. Martino. *Sistemi di Basi di Dati Orientate agli Oggetti.* Addison-Wesley, 1992.

[BNPS92] 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(3):223–237, 1992.

[BOSW99] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java^{TM} Programming Language. In *Proc. Fourteenth International Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 183–200, 1999.

[BWJ98] C. Bettini, X. S. Wang, and S. Jajodia. Temporal Semantic Assumptions and Their Use in Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):277–296, 1998.

[Car88] 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*, number 303 in Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 1988.

[Cas97] G. Castagna. *Object-Oriented Programming: A Unified Foudation.* Birkhauser, Boston, 1997.

[Cat93] R. Cattel. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, 1993.

[Cat94] R. Cattel. *The Object Database Standard: ODMG-93 (Release 1.1).* Morgan-Kaufmann, 1994.

[Cat96] R. Cattel. *The Object Database Standard: ODMG-93 (Release 1.2).* Morgan-Kaufmann, 1996.

[CBB+97] R. Cattel, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan-Kaufmann, 1997.

[CBB+99] R. Cattel, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stanienda, and F. Velez. *The Object Database Standard: ODMG 3.0.* Morgan-Kaufmann, 1999.

[CC95] T. Coupaye and C. Collet. Denotational Semantics for an Active Rule Execution Model. In T. Sellis, editor, *Proc. Second International Workshop on Rules in Database Systems*, number 985 in Lecture Notes in Computer Science, pages 36–50. Springer-Verlag, 1995.

[CCN+99] M. Carey, D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, and N. Mattos. O-O, What's Happening to DB2? In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 511–512, 1999.

[CCP97] C. Combi, G. Cucchi, and F. Pinciroli. Applying Object-Oriented Technologies in Modeling and Querying Temporally Oriented Clinical Databases Dealing with Temporal Granularity and Indeterminacy. *IEEE Transactions on Information Technology in Biomedicine*, 1(2):100–127, 1997.

[CCS94] 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.

[CDI+97] J. Clifford, C.E. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass. On the Semantics of "Now" in Databases. *ACM Transactions on Database Systems*, 22(2):171–214, 1997.

[CFPT96] 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.

[CG93] T. Cheng and S. Gadia. An Object-Oriented Model for Temporal Databases. In *Proc. of the Int'l Workshop on an Infrastructure for Temporal Databases*, 1993.

[Cha92] 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, 1992.

[Cha96] D. Chamberlin. *Using the New DB2 - IBM's Object-Relational Database System*. Morgan-Kaufmann, 1996.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[Che76] P. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CJ99] R. Cartwright and G.L. Steele Jr. Compatible Genericity with Run-time Types for the Java^{TM} Programming Language. In *Proc. Fourteenth International Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 201–215, 1999.

[Clu98] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5):279–305, 1998.

[CM92] R. Connor and R. Morrison. Subtyping without Tears. In *Proc. Fifteenth Australian Computer Science Conference*, pages 209–225, 1992.

[CPW97] J. Campin, N. Paton, and H. Williams. Specifying Active Databases Systems in an Object-Oriented Framework. *Software Engineering and Knowledge Engineering*, 7(1):101–123, 1997.

[CR87] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In *Proc. of the Conference on Temporal Aspects in Information Systems*, pages 23–30, 1987.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[CW96] S. Ceri and J. Widom. *Active Database Systems*. Morgan-Kaufmann, 1996.

[CW97] M. Chavda and P. T. Wood. Towards an ODMG-Compliant Visual Object Query Language. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proc. Twentythird Int'l Conf. on Very Large Data Bases*, pages 456–465. Morgan-Kaufmann, 1997.

[CZ00] A.B. Chaudhri and R. Zicari, editors. *Succeeding with Object Databases: A Practical Look at Today's Implementations with Java and XML*. John Wiley & Sons, 2000.

[DBC96] U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC project. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.

[DELS00] C. E. Dyreson, W.S. Evans, H. Lin, and R. Snodgrass. Efficiently Supporting Temporal Granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.

[Des98] Object Design. ObjectStore Java API User Guide (ObjectStore 6.0). Available at `http://www.odi.com`, 1998.

[Des99] Object Design. An XML Data Server for Building Enterprise Web Applications. White paper, 1999.

[DJ97] O. Diaz and A. Jaime. EXACT: An Extensible Approach to Active Object-Oriented Databases. *VLDB Journal*, 6(4):282–295, 1997.

[DPG91] 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.

[EdOP93] N. Edelweiss, J.P.M. de Oliveira, and B. Pernici. An Object-Oriented Temporal Model. In *Proc. Advanced Information Systems Engineering, CAiSE'93*, number 685 in Lecture Notes in Computer Science, pages 397–415. Springer-Verlag, 1993.

[EdOP94] N. Edelweiss, J.P.M. de Oliveira, and B. Pernici. An Object-Oriented Approach to a Temporal Query Language. In *Proc. Database and Expert Systems Applications, 5th International Conference, DEXA '94*, number 856 in Lecture Notes in Computer Science, pages 225–235. Springer-Verlag, 1994.

[EM00] A. Eisenberg and J. Melton. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, 2000.

[FE98] L. Fegaras and R. Elmasri. A Temporal Object Query Language. In *IEEE Proc. Fifth International Workshop on Temporal Representation and Reasoning*, IEEE Computer Society Press, pages 51–59, 1998.

[Feg99] L. Fegaras. VOODOO: A Visual Object-Oriented Database Language for ODMG OQL. In *ECOOP Workshop on Object-Oriented Databases*, pages 61–72, 1999. Electronic edition available at http://www.disi.unige.it/conferences/oodbws99.

[FMP97] C. Fahrner, T. Marx, and S. Philippi. DICE: Declarative Integrity Constraint Embedding Into the Object Database Standard ODMG-93. *Data and Knowledge Engineering*, 23(2):119–145, 1997.

[FSRM00] L. Fegaras, C. Srinivasan, A. Rajendran, and D. Maier. lambda-DB: An ODMG-Based Object-Oriented DBMS. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 2000.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems*, 20(4):414–471, 1995.

[FWP97] A. Fernandes, H. Williams, and N. Paton. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205–244, 1997.

[Gad88] S.K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems*, 13(4):418–448, 1988.

[GBB98] 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.

[GD92] 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.

[GFV96] S. Gatziu, H. Fritschi, and A. Vaduva. SAMOS an Active Object-Oriented Database System: Manual. Technical Report 96.02, Department of Computer Science, University of Zurich, 1996.

[GJ91] 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.

[GJ96] N. Gehani and H. Jagadish. Active Database Facilities in Ode. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.

[GJS92] 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.

[GLO+98] I.A. Goralwalla, Y. Leontiev, T. Ozsu, D. Szafron, and C. Combi. Temporal Granularity for Unanchored Temporal Data. In *ACM Proc. of the 1998 ACM CIKM International Conference on Information and Knowledge Management*, ACM Press, pages 414–423, 1998.

[GM99] F. Grandi and F. Mandreoli. ODMG Language Extensions for Generalised Schema Versioning Support. In P. P. Chen, D. W. Embley, J. Kouloumdjian, S. W. Liddle, and J. F. Roddick, editors, *Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, volume 1727 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 1999.

[GMF+99] G. Guerrini, I. Merlo, E. Ferrari, G. Kappel, and A. de Miguel. The First ECOOP Workshop on Object-Oriented Databases. In A. Moreira and S. Demeyers, editors, *ECOOP99 Workshop Reader*, number 1743 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[GN93] S.K. Gadia and S.S. Nair. Temporal Databases: A Prelude to Parametric Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 28–66. Benjamin/Cummings, 1993.

[GP99] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* Miller Freeman, 1999.

[Gro94] ODMG Group. Response to the ODMG-93 Commentary Written by Dr. Won Kim of UniSQL, Inc. *ACM SIGMOD Record*, 23(3):4–9, 1994.

[HK97] T. Hruska and P. Kiloncik. Comparison of Categorical Foundations of Object-Oriented Database Model. In F. Bry et al., editors, *Proc. Fifth Int'l Conf. on Deductive and Object-Oriented Databases*, number 1341 in Lecture Notes in Computer Science, pages 302–319. Springer-Verlag, 1997.

[JD98] C.S. Jensen and C.E. Dyreson. The Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice*, number 1399 in Lecture Notes in Computer Science, pages 366–405. Springer-Verlag, 1998.

[Kim94] W. Kim. Observations on the ODMG-93 Proposal for an Object-Oriented Language. *ACM SIGMOD Record*, 23(1):3–7, 1994.

[Kow92] R. Kowalsky. Database Updates in Event Calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[KR98] G. Kappel and W. Retschitzegger. The TriGS Active Object-Oriented Database System - An Overview. *SIGMOD Record*, 27(3):36–41, 1998.

[KRSR94] 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. Springer-Verlag, 1994.

[KS92] W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 266–275. ACM Press, 1992.

[LK99] S. Lee and H. Kim. Object Versioning in an ODMG-compliant Object Database System. *Software Practice and Experience*, 29(5), 1999.

[LRV90] C. Lecluse, P. Richard, and F. Velez. o_2, an Object-Oriented Data Model. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 257–276. Addison-Wesley, 1990.

[LS93] C. Laasch and M. H. Scholl. Determinstic Semantics of Set-Oriented Update Sequences. In *Proc. Ninth IEEE Int'l Conf. on Data Engineering*, pages 4–13, 1993.

[MBF+00] I. Merlo, E. Bertino, E. Ferrari, S. Gadia, and G. Guerrini. Querying Multiple Temporal Granularity Data. In S. Goodwin and A. Trudel, editors, *IEEE Proc. Seventh International Workshop on Temporal Representation and Reasoning*, IEEE Computer Society Press, pages 103–114, 2000.

[MBFG99] I. Merlo, E. Bertino, E. Ferrari, and G. Guerrini. A Temporal Object-Oriented Data Model with Multiple Granularities. In C. Dixon and M. Fischer, editors, *IEEE Proc. Sixth International Workshop on Temporal Representation and Reasoning*, IEEE Computer Society Press, pages 73–81, 1999.

[MD89] 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.

[Mei92] B. Meier. *Eiffel: The Language.* Prenctice Hall - Object-Oriented Series, Second Edition, 1992.

[Mel99] J. Melton. Ansi/iso working draft - sql foundation: Ansi x3h2-99-079/wg3:ygj-011. Available at `ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-foundation-wd-1999-03.pdf`, 1999.

[MGP98] N. Murray, C. A. Goble, and N. W. Paton. Kaleidoscape: A 3D Environment for Querying ODMG Compliant Databases. In Y. E. Ioannidis and W. Klas, editors, *Visual Database Systems 4 (VDB4), IFIP TC2/WG 2.6 Fourth Working Conf. on Visual Database Systems*, pages 85–101. Chapman & Hall, 1998.

[MMCR92] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in the Event Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems 1992*, pages 702–712, 1992.

[MP91] 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. Springer-Verlag, 1991.

[MPC96] R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, number 1057 in Lecture Notes in Computer Science, pages 56–76. Springer-Verlag, 1996.

[OPR96] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA*. Prentice Hall, 1996.

[OPS+95] M. T. Ozsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *VLDB Journal*, 4(3):445–492, 1995.

[Ora97] Oracle Corporation. *Oracle8 Server SQL Reference, Release 8.0*, 1997.

[Pat99] N. W. Paton. *Active Rules in Database Systems*. Springer New York, 1999.

[PCFW95] 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. Springer-Verlag, 1995.

[RKJ98] M. Roantree, J. B. Kennedy, and P. J.Barclay. A Multidatabase Layer for the ODMG Object Model. In C. Rolland and G. Grosz, editors, *Proc. 1998 International Conference on Object-Oriented Information Systems*. Springer-Verlag, 1998.

[Ros92] G. A. Rose. Object-Z. In S. Stepheney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, pages 59–77. Springer-Verlag, 1992.

[RPS95] S. Reddi, A. Poulovassilis, and C. Small. Extending a Functional DBPL With ECA-Rules. 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 101–115. Springer-Verlag, 1995.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. Tenth Int'l Conf. on the Entity-Relationship Approach*, pages 205–229, 1991.

[RS97] H. Riedel and M. H. Scholl. A Formalization of ODMG Queries. In *Proc. Seventh International Conference on Data Semantics (DS-7)*, 1997.

[SA99] J.H. Solorzano and S. Alagić. Parametric Polymorphism for Java^{TM}: A Reflective Solution. In *Proc. Fourteenth International Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 216–225, 1999.

[Sak96] M. Sakkinen. More about the ODMG-93 standard, and a bit about manifestos. Technical Report 4/96, Department of Information Systems, University of Linz, 1996.

[Sho87] Y. Shoham. Temporal Logics in AI: Semantical and Ontological Considerations. *Artificial Intelligence*, 33(1):89–104, 1987.

[SN97] A. Steiner and M.C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Proc. of the Fifth International Conference on Database Systems for Advanced Applications*, pages 381–390, 1997.

[Sno95a] R. T. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 386–408. Addison-Wesley/ACM Press, 1995.

[Sno95b] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.

[Sno99] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan-Kaufmann, 1999.

[Sof99] POET Software. POET Java^{TM} Programmer's Guide (POET 6.0), 1999.

[SP98] P.R.F. Sampaio and N. W. Paton. Deductive Queries in ODMG Databases: the DOQL Approach. In C. Rolland and G. Grosz, editors, *Proc. 1998 International Conference on Object-Oriented Information Systems*, pages 57–74. Springer-Verlag, 1998.

[SYB] SYBASE Inc., Berkley, California. *Transact-SQL user's guide for Sybase*. Release 10.0.

[Ter97] P. Terenziani. Integrating Calendar Dates and Qualitative Temporal Constraints in the Treatment of Periodic Events. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):763–783, 1997.

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 1997.

[vB99] M.F. van Bommel. Interactive Design of Object-Oriented Schema. In *Proc. of the 1999 Int'l Database Engineering and Applications Symposium*, pages 16–23, 1999.

[WD93] G. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 230–247. Benjamin/Cummings, 1993.

[Wid92] J. Widom. A Denotational Semantics for the Starbust Production Rule Language. *ACM SIGMOD Record*, 21(3):4–9, 1992.

[WJL91] G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with Granularity of Time in Temporal Dabases. In *Proc. of the Third Nordic Conference on Advanced Information Systems Engineering*, pages 124–140, 1991.

[WJW98] Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. In *Temporal Databases: Research and Practice*, number 1399 in Lecture Notes in Computer Science, pages 338–366. Springer-Verlag, 1998.

[Wor98] Word Wide Web Consortium. Extensible Markup Language (XML) 1.0, February 1998. W3C Recommendation available at `http://www.w3.org/TR/REC-xml`.

[Zan94] 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, 1994.

Appendix A

Notations used in the Thesis

Table A.1 illustrates the symbols most frequently used in the thesis. For each symbol the table reports a brief explanation of its meaning. We assume all sets introduced in Table A.1 be mutually disjoint. This assumption is only useful to simplify definitions and results, but it is used only at theoretical level, while in ODMG it does not hold. One of the most remarkable characteristics of object-oriented data models is indeed the independence of each interface/class definition from the others.

Symbol	Meaning
$\mathcal{T}$	set of ODMG types
$\mathcal{T}+$	set $\mathcal{T} \cup$ {`void`}
$\mathcal{OT}$,$\mathcal{LT}$,$\mathcal{NT}$	set of ODMG object types/literal types/named types
$\mathcal{OT}_{int}$, $\mathcal{OT}_{class}$	set of object types defined through an interface/class definition
$\mathcal{UOT}$,$\mathcal{COT}$,$\mathcal{SOT}$	set of ODMG user-defined/collection/structured object types
$\mathcal{ALT}$,$\mathcal{CLT}$,$\mathcal{SLT}$	set of ODMG atomic/collection/structured literal types
$\mathcal{OI}$,$\mathcal{II}$,$\mathcal{CI}$	set of object/interface/class identifiers
$\mathcal{STRUCT}_{field}$	set of struct field names
$type(\tau)$	type associated with a named type τ
$dom(\tau)$	set of values of a literal type τ
$\pi(\tau)$	extent (set of instances) of an object type τ
$\pi^+(\tau)$	set of *proper* instances of an object type τ
$[\![\tau]\!]$	set of values of a type τ
$\mathcal{V}$	set of ODMG values
$\mathcal{AN}$, $\mathcal{RN}$, $\mathcal{MN}$	set of attribute/relationship/method names
$\mathcal{NAME}$	set of object names
$ISA(\tau)$	set of direct superinterfaces of an object type τ
$ISA^*(\tau)$	set of superinterfaces of an object type τ
$EXTENDS(\tau)$	direct superclass of an object type τ
$EXTENDS^*(\tau)$	set of superclasses of an object type τ
$\mathcal{A}(\tau)$	types and names of the attributes of an object type τ
$\mathcal{A}^+(\tau)$	types and names of the proper attributes of an object type τ
$\mathcal{R}(\tau)$	types, names and inverse of the relationships of an object type τ
$\mathcal{R}^+(\tau)$	types, names and inverse of the proper relationships of an object type τ
$\mathcal{A}_{names}(\tau), \mathcal{R}_{names}(\tau), \mathcal{M}_{names}(\tau)$	names of the attributes/relationships/methods of an object type τ
$\mathcal{A}^+_{names}(\tau), \mathcal{R}^+_{names}(\tau), \mathcal{M}^+_{names}(\tau)$	names of the proper attributes/relationships/methods of an object type τ
$source(n, \tau)$	most specific object type from which τ inherits property n
$<_T$,$\leq_T$	subtype relationship and its reflexive closure
$Id(S)$	set of identifiers of the object type external specifications in S

Table A.1: Notations and terminology used in the thesis

Symbol	**Meaning**
$\mathcal{IS}$	index set
$(\mathbb{N},\leq)$	time domain
0	relative beginning
now,∞	time instants now and $forever$
$\mathcal{G}$	set of granularities
l_i^G	label of the ith granule of granularity G
$\mathcal{LABEL}$, $\mathcal{LABEL}_G$	set of granule labels, set of granule labels corresponding to granularity G
$\prec$,$\preceq$	finer than relationship and its reflexive closure
$G_\emptyset$, G_I	empty and chronon granularity
$[i,j]^G$	interval with respect to granularity G
$\mathcal{T}_R$	set of ODMG types considered for extensions $\mathcal{T}_R = \mathcal{OT}_{class} \cup \mathcal{LT}$
$temporal_G(\tau)$	generic temporal type ($G \in \mathcal{G}$ and $\tau \in \mathcal{T}_R$)
$\mathcal{TT}$	set of temporal types
$\mathcal{T}_{temp}$	set of T_ODMG types
$\mathcal{OT}_{temp}$,$\mathcal{LT}_{temp}$	set of T_ODMG object/literal types
$\mathcal{ST}$	$\mathcal{ST} = \mathcal{T}_{temp} \setminus \mathcal{TT}$ set of T_ODMG static types
$\pi_{temp}(\tau,t)$	set of instances of an object type τ at time t
$\pi^+_{temp}(\tau,t)$	set of *proper* instances of an object type τ at time t
$[\![\tau]\!]_t$	set of values of the static type τ at time t
$[\![\tau]\!]_i^G$	set of values of the static type τ in granule $G(i)$
$[\![\tau]\!]_{l_i^G}$	alternative notation to $[\![\tau]\!]_i^G$
$\mathcal{V}_{temp}$	set of T_ODMG values
v^H	conversion of value $v \in [\![temporal_G(\tau)]\!]$ to H, $H \preceq G$
Υ^G	temporal element with respect to granularity G
$\mathcal{TELEM}$, $\mathcal{TELEM}_G$	set of temporal elements, set of temporal elements corresponding to granularity G
$\mid S \mid$	cardinality of set S
$min([i,j]^G)$	lower bound i of the interval $[i,j]^G$
$max([i,j]^G)$	upper bound j of the interval $[i,j]^G$

Table A.1: Notations and terminology used in the thesis (continued)

Symbol	Meaning
$\Pi(\Upsilon^G, n)$	n-th interval in the temporal element Υ^G
Exp_{simple}	set of well-formed simple expressions
$\mathcal{E}_{simple}$	semantic function for simple expressions
Exp_{temp}	set of well-formed temporal expressions
$\mathcal{E}_{temp}$	semantic function for temporal expressions
Exp_{path}	set of well-formed path expressions
$\mathcal{E}_{path}$	semantic function for path expressions
ϑ	oid-assignment
Θ	set of all oid-assignments
T_{BS},T_{BI}	sequence of activated before set/instance-oriented triggers
T_{AS},T_{AI}	sequence of activated after set/instance-oriented triggers
TN	set of trigger names
2^S	powerset of set S
L^S,$\circ$	set of all possible sequences with elements in S, sequence concatenation
$Rule$	set of triggers defined for a database
(T, c, e, C, A, t, g, v)	tuple modelling trigger T
$\mathcal{E}$	semantic function modeling expression evaluation
$\mathcal{C}$	semantic function modeling condition evaluation
$\mathcal{U}$	semantic function modeling update evaluation, if no trigger is activated
$\mathcal{A}$	semantic function modeling update evaluation, if triggers are activated
$\mathcal{T}$	semantic function modeling trigger evaluation
$\mathcal{R}_s, \mathcal{R}_i, \mathcal{I}$	semantic functions used in trigger evaluation
$\langle e, c, O\rangle$	even instance of event e on class c and objects O
$trig(u)$	set of rules triggered by an update
$conv(u, o, S)$	conversion of update u in instance-oriented update w.r.t. object o and state S
$d(o, T)$	class whose implementation for T is selected for execution on object o
$P(O, T)$	partitioning of set of objects O w.r.t. dispatching of trigger T

Table A.1: Notations and terminology used in the thesis (continued)

Appendix B

Triggers in SQL:1999

The processing granularity is an orthogonal dimension with respect to the activation time, thus both before and after triggers can be either instance-level or statement-level. Table B.1 summarizes the meaning of the resulting combinations, whereas Table B.2 summarizes which kinds of transition variables and tables can be employed in different kinds of triggers. In particular, it shows, for each kind of trigger and each kind of transition variable or table the events that can be monitored by the trigger which is allowed to refer to that transition structure.

	statement-level trigger (`FOR EACH STATEMENT`)	instance-level trigger (`FOR EACH ROW`)
`BEFORE` option	*before statement trigger*: the trigger is executed only once, before executing the triggering operation	*before row trigger*: the trigger is executed before manipulating each tuple affected by the triggering operation
`AFTER` option	*after statement trigger*: the trigger is executed only once, after executing the triggering operation	*after row trigger*: the trigger is executed after having modified each tuple affected by the triggering operation

Table B.1: Options for SQL:1999 trigger execution

The following example illustrates triggers in SQL:1999.

Example B.1 Referring to the following tables:

`MovieExec(cert#, name, address, netWorth)`

`Studio(name, address, presCert#)`

	OLD ROW	NEW ROW	OLD TABLE	NEW TABLE
before statement trigger	-	-	-	-
before row trigger	delete, update	insert, update	-	-
after statement trigger	-	-	delete, update	insert, update
after row trigger	delete, update	insert, update	delete, update	insert, update

Table B.2: Transition tables and variables allowed in different kinds of SQL:1999 triggers

the following trigger (adapted from [UW97]) undoes any modification attempting to reduce the net worth of a studio president.

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING OLD AS OldTuple, NEW AS NewTuple
FOR EACH ROW
       WHEN (OldTuple.netWorth > NewTuple.netWorth AND
             NewTuple.cert# IN SELECT presCert# FROM Studio)
       UPDATE MovieExec
          SET netWorth = OldTuple.netWorth
          WHERE cert# = OldTuple.cert#
```

◊