Making object-oriented databases more knowledgeable (From ADAM to ABEL)

A thesis presented for the degree of
Doctor of Philosophy
at the University of Aberdeen

Oscar Díaz García
Licenciado en Informática (Basque Country University)

1992

Declaration

This thesis has been composed by myself, it has not been accepted in any previous application for a degree, the work of which it is a record has been done by myself and all quotations have been distinguished by quotations marks and the sources of information specially acknowledged.

Oscar Díaz García
28 October 1991
Department of Computing Science
University of Aberdeen
King's College
Aberdeen, Scotland

Acknowledgements

I would like to thank my supervisors Prof. Peter Gray at the University of Aberdeen and Dr. Arantza Illarramendi at the Basque Country University for their advice and guidance during the past three years.

The work described in this thesis has benefited from many discussions with colleagues in Aberdeen and San Sebastián. This is the part of the thesis where I would have most liked to use my mother tongue to thank in a more friendly and warm way all of the people which help me during these years. But, do not worry, reader, you will not have to put up with a sentimental and trying piece of Spanish prose! First of all, I am greatly indebted with Norman Paton, the author of ADAM, whose encouragement, patience and generosity has been invaluable throughout. I would like also to thank José Miguel Blanco, for his friendship and lively discussions, hardly about Computer Science but about much more interesting subjects. Particular thanks have also to be given to Suzanne Embury for her patience in correcting my clumsy English and for her help in implementing some parts of ABEL. I have also had useful discussions with Graham Kemp and Zhuoan Jiao with whom I kept up a friendly and enlightening rivalry between P/FDM and ADAM. My gratitude to Javier Torrealdea whose encouragement and help in coping with administrative tasks made it possible for me to stay in Aberdeen. I would like also to thank Ian Kirby for his help with LaTeX, and Jorg Forster and Inés Arana for arranging things for me when I was away. I would like to thank the computer officers Chris Cotton, Steve Trythall, Andrew Mellanby and Nick Murray in Aberdeen and Blanca Martínez and Liborio Revilla in San Sebastián.

Finally, I would like to thank Ana for not letting me work most of the week-ends, and my parents for their support throughout.

This work was supported by a F.P.I grant from the Spanish Government.

Summary

The salient points of this thesis are as follows:

- Object-Oriented Databases can help in solving the impedance mismatch problem by introducing methods. However, methods have sometimes been overused in the sense that the code encapsulated refers not only to how the operation is implemented but also to other kinds of knowledge that are implicit in the code. The disadvantages of this approach for modelling integrity constraints, user-defined relationships and active behaviour are pointed out.
- The ADAM Object-Oriented Database has been extended to allow the designer to specify integrity constraints declaratively. A constraint equation approach is implemented that supports the inheritance of constraints.
- A need for semantic-rich user-defined relationships has been identified. In this thesis, relationships are represented as objects. An approach to enhance the semantics of relationships in both its structural and behavioural aspects is presented. The most novel idea of the approach presented is the support of the inferred properties and the operational semantics of relationships.
- Active Databases have recently become an important area of research. This thesis shows how to extend an Object-Oriented Database with active capabilities. The principal contribution lies in representing as 'first-class' objects not only the active rules but also the rule manager itself. Hence, besides handling active rules as any other object in the system, future requirements can be supported just by specialising the current rule manager.
- Active rules have been proposed for several purposes. Several examples are given of the direct use of rules. However, higher level tools can be provided of which rules

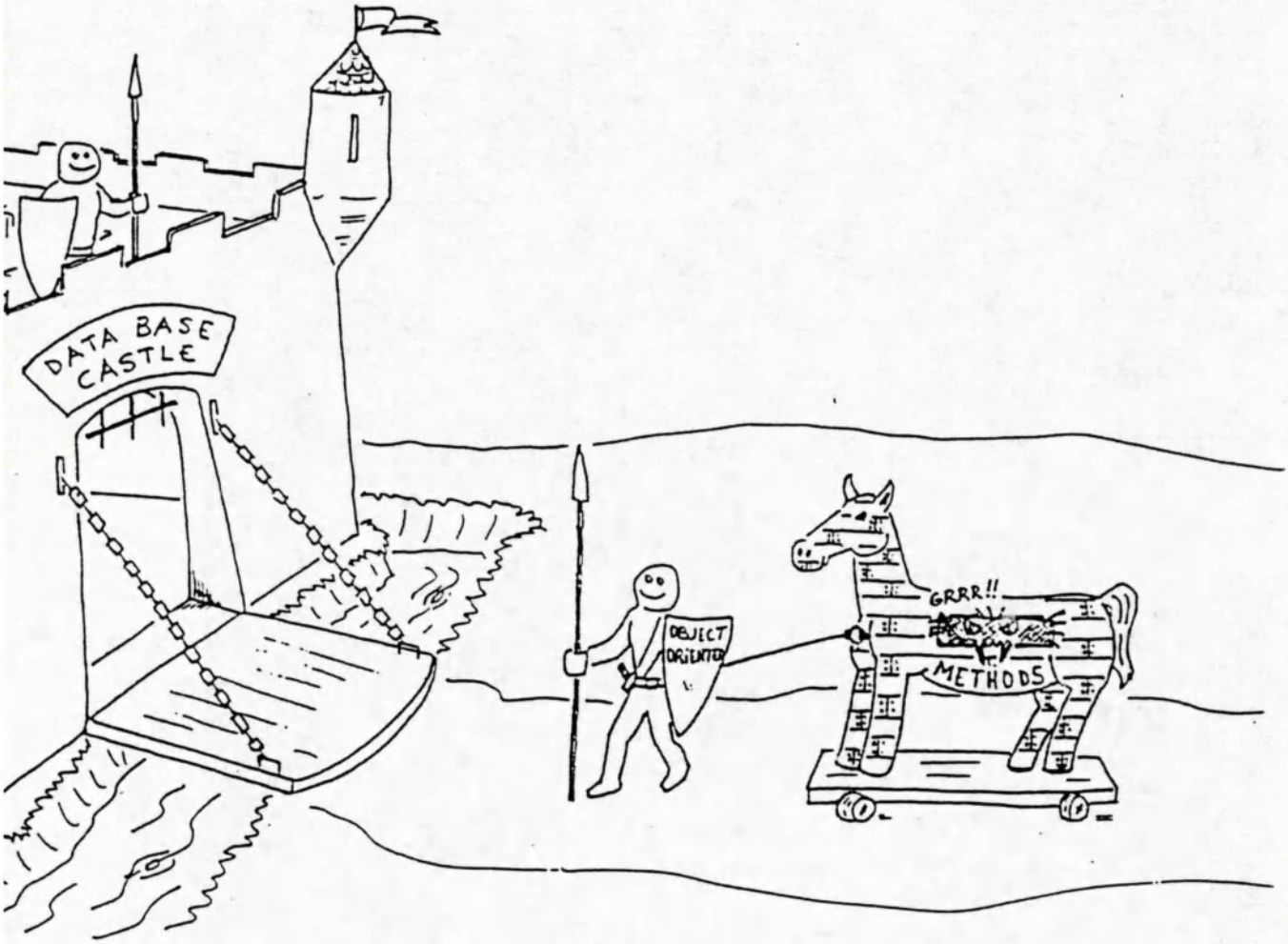
are simple a vehicle for implementation. This is shown by generating rules from the declarative specification of integrity constraints and it has also been applied to generate rules for enforcing the operational semantics of relationships.

- Metaclasses are a valuable mechanism for enhancing the uniformity, accessibility and extensibility of the system. This is demonstrated by showing how ADAM has been extended with integrity constraints, semantic-rich user-defined relationships and active behaviour.

Contents

List of Figures

Once upon a time ...



Chapter 1

Introduction

This thesis is about extending an Object Oriented Data Base with a set of constructs that allow one to capture explicitly those features of the Universe of Discourse (UoD) that are implicit in method definitions. The focus is on integrity constraints, user-defined relationships and active behaviour. To identify, represent and extend the data base with these primitives constitutes the main contributions of this work.

1.1 Purpose of the thesis: Motivations and Contributions

Object Oriented Data Bases (OODBs) have been proposed both for coping with the *impedance mismatch* problem and for increasing the semantic details kept in the Data Base (DB) by collecting not only the structural features of the application but the behavioural ones as well. This is achieved by including procedures (called *methods*) within schema definition.

However, methods have sometimes been overused in the sense that the code encapsulated refers not only to how the operation is implemented but also to other kinds of knowledge that is implicit in the code. In this way, methods can be seen as a kind of *Trojan horse* which introduce surreptitiously embedded knowledge that is not open to direct manipulation. As a result, not only is it more difficult to update the system, but its ease of use, legibility and inferential power are also diminished.

The aim of this work is to provide a set of primitives to represent knowledge explicitly in an OODB. Until now, much of this knowledge was implicit in method code, and so

could not be used for other purposes. The focus is on four different kinds of knowledge, namely

- metaknowledge (metaclasses)
- integrity constraints (attribute facets)
- active behaviour (event-condition-action rules)
- user-defined relationships (relationship objects)

In parentheses we give the approach that has been followed, which is more carefully described in the following sections.

By providing primitives that allow these concepts to be captured explicitly, the OODBs become more *knowledgeable.* In addition, since most of the primitives are applicable to objects and they have been implemented as objects themselves, the concept represented by the primitive can be applied to other primitives or even to itself. This is shown by involving primitives in defining other primitives. *Hence, providing this set of constructs, not only allows knowledge to be made explicit but also a new range of possibilities stemming from the combination of primitives.*

This has been born out by an implementation in ADAM, an OODB in Prolog. It has to be emphasised that this extension has been made in an object oriented fashion (i.e. specialising and reusing previous concepts already existing in ADAM), and that only once was the core of the system modified. The use of metaclasses to achieve extensibility is also illustrated throughout the thesis and constitutes a contribution of this work on the line established by [Paton 89a]. The result is called ABEL (ABerdeen Euskadi data modeL) which we hope will become the *good successor* of ADAM.

1.2 Context: From Data Bases to Knowledge Bases

One of the rationales behind data bases (DBs) is to achieve *data independence* by separating data from applications. This has been seen as a major breakthrough, since it allows the data to be used by many applications and to be managed centrally. However, traditional data models usually capture the plain structure but very little of the meaning of the data. Hence, information on the meaning of the data and how this data can be used

is spread throughout the applications. As with file-based systems, redundancy, inconsistency and maintainability problems arise, but now at the knowledge level. More recently knowledge base management systems (KBMS) have emerged which pursue what can be called *knowledge independence*, i.e. representing knowledge declaratively regardless of how this knowledge is used. This can be seen as an evolution of the data independence concept. Whereas in a DBMS most data is just ground data, a KBMS attempts to increase the amount of generic data (or knowledge) kept in the DB. Knowledge is extracted from applications and moved to the DB. But this places new requirements on the DB. One must enhance DB expressiveness by providing a richer set of primitives that allow this knowledge to be captured explicitly. In the following, a brief outline of the gaining of expressiveness in DBs is presented from record-oriented models to KBMSs.

Traditional data models are basically record-oriented. For instance, in the relational model, tuples in a table correspond to records of a file, and functional dependencies are enforced through the concept of key. This leads to severe limitations when the UoD does not fit directly into tables. To overcome these limitations, *Semantic Data Models* (SDMs) have been developed which attempt to increase the semantic content of the DB. SDMs have objects, relationships, dynamic properties and integrity constraints. Traditional data models attempt to overcome the shortage of powerful constructs by using integrity constraints. Some integrity constraints are part of the model itself, the so-called *structural constraints*. However, these constraints are not sufficient to model all the complexity of the UoD. In this case, the semantics has to be embedded into user programs or reflected by means of a constraint language expressing the so-called *behavioural constraints*.

SDMs enlarge the number of structural constraints supported by the model, by providing constructs to represent explicitly abstract relationships that were already used both in Artificial Intelligence (AI) and philosophy, such as *generalisation, aggregation, classification* and *association*. A clear semantics must be defined, specifying how insertion, deletion and modification operations made at a higher abstraction level (e.g. *person*) can affect the object abstracted (*student, lecturer* or other subclasses) and vice versa.

It is worth noticing that in relational DBs, tables can also materialise hierarchies. However, the difference is that here the semantics of the generalisation and classification relationships is in the user programs which have to draw the corresponding inferences,

whereas in a SDM these relationships are provided as *primitives* and the user has only to *specify* the hierarchy, leaving the semantic maintenance to the system itself.

However, it is not only a question of legibility or ease of use. A more fundamental distinction underlies both approaches: whereas with relational systems, programs *know how* to cope with the hierarchy, in a SDM approach programs not only know how to treat a hierarchy but also *know that* there is a hierarchy. This distinction corresponds to the philosophical difference between knowing *how to* and knowing *that.* Integrity constraints are a case in point. In relational systems, code can be added to a table to validate a given constraint (the *VALIDPROC* procedure in DB2). In this way, the system knows how to enforce this constraint but it is unaware of the constraint itself. Being wired into the code, the constraint cannot be used for other purposes such as semantic query optimisation [Chakravarthy 90, Demolombe 90].

KBMSs aim to make knowledge explicit. As pointed out in [Freundlich 90] "explicit means open to direct manipulation. Within the programming context, this means removing the knowledge from the procedural setting in which it is usually embedded in conventional programming and representing it in a declarative form." Understandability, modularity, maintainability and extensibility are greatly enhanced with this new approach.

However, what makes a knowledge representation formalism different from a simple data structure is the possibility of it being interpreted, i.e. the ability to draw inferences, allowing information to be obtained which is implicit in the knowledge base. Thus, unlike relational DBs, in KBMSs the data available is not only the data explicitly stored but also data that can be *inferred* from this knowledge. As an example, if *Peter* has been defined as a *student*, the system can automatically infer that *Peter* is a *person* from the semantics of the generalisation abstraction, without it being explictly declared. From this point of view, SDMs can be seen as a rudimentary KBMS where primitives are provided to represent explicitly a set of abstract relationships.

SDMs focus on the structural features of the UoD. Thus a *student* can be seen as a classification (where for instance, the common characteristics of *graham, zhuoan* and *suzanne* are abstracted) and as a specialisation of a higher abstraction called *person.* More recently, OODBs have emerged, where all information concerned with an object is gathered

together. This refers not only to the structural but to the behavioural features as well. Now a *student* is not only a *person* that has a *registration number* at the University (i.e. a specialisation of *person*) but also one that behaves in a certain way (e.g. that is able to *study*). So, an object is characterised by the set of actions that it can undertake (the so-called *object interface*). How an action is implemented (i.e. *the method*) is *transparent* to the user who only accesses the object that through *messages*.

Besides being a step ahead in modelling the UoD, OODBs can also help in solving the *impedance mismatch* problem. This problem refers to the type clash that arises when a relational languages such as SQL is embedded within a conventional programming language (e.g. COBOL). Then the set-at-a-time relational processing has to be converted to the record-at-a-time processing of conventional languages.

Apart from such remarkable advantages, methods have sometimes been overused since the code encapsulated not only refers to the implementation of the operation but to other kinds of knowledge as well. Integrity constraint maintenance is a case in point. As a result, the DB-KB philosophy whereby data (ground or generic) is separated from application, is jeopardised.

In the following sections, different kinds of knowledge are presented. The issue of how to represent such knowledge explictly and how to extend the system with the appropriate constructs is the main concern of this work.

1.3 Meta-knowledge

Unlike more traditional databases, which are mainly concerned with *the extension* of the UoD, knowledge-based systems are characterised by an increase of the knowledge kept in the database, often referred to as *the intensional* side of the UoD. A new range of applications has arisen which need to treat the objects in the data dictionary as regular data, as *objects of discourse* [Freundlich 90]. This can be achieved by making explicit the metaknowledge, i.e. knowledge about knowledge.

In an object oriented (OO) paradigm knowledge is represented through *classes* that gather together common features of a set of objects called the class *instances*. Thus, classes collect generic data or knowledge. In this context, to make knowledge explicit

means to treat classes as any other object in the system, i.e. the ability to query a class about its information in the same way that for example, the *registration number* of *graham* can be retrieved. Furthermore, the OO paradigm encourages reuse and modularity through the subclass mechanism. When a new class has to be introduced in the system, the designer thinks about the differences between, and similarities within existing classes, pointing out what is really new and what can be reused. Although this philosophy has been broadly used in user-applications, few systems apply it to the definition of the system itself.

The ability of query a class as well as the possibility of classifying, generalising and relating classes can be accomplished by introducing *metaclasses*. A metaclass is used to define the structure and behaviour of class objects in the same way that a class is used to define the structure and behaviour of instance objects. Metaclasses not only permit classes to be stored and accessed using the facilities of the data model, but make it possible to refine the default behaviour for class creation using specialisation and inheritance.

ADAM provides metaclasses that account for an OO approach to system extensibility. From this point of view, the system can be seen as a core of useful facilities which the user can tailor to suit specific applications, a philosophy similar to that advocated by the proponents of extensible DB systems [Carey 88]. In this way, *uniformity, extensibility* and *accessibility* are greatly enhanced. As a case in point, this thesis is about extending ADAM with integrity constraints, active behaviour and user-defined relationships.

1.4 Integrity Constraints

Integrity constraints can be seen as restrictions which must hold between different pieces of information to keep the database *consistent*. Besides *structural constraints*, i.e. those provided by the data model, there are other kinds of constraints that cannot be reflected in a structural way. These constraints range from simple domain restrictions (e.g. the age of a teenager must be between 10 and 19) to more complex relationships between different pieces of information (e.g. the projects a lecturer has responsibility for are to be the same as the set of projects which his/her research-assistants work on).

Usually the specification and checking of these constraints is left to the user, thereby being hard coded in application programs. Hence, constraints cannot be *uniformly* enforced

for all users. Through integration of data and programs, the OO paradigm allows a uniform enforcement by coding integrity constraints as part of methods *within the database.* However this is not a satisfactory solution. Several disadvantages can be enumerated:

- Overriding is a common practice in OO systems, a subclass redefining a method may no longer maintain constraints defined at a higher level in the hierarchy. Hence, constraints and methods should be two separate mechanisms since the underlying philosophies are quite different, namely that constraints represent invariant states of the database but method definitions can sometimes be overridden.
- A designer may not correctly consider the effects of the constraint from the point of view of all objects involved.
- Being implicit, constraints cannot be used to enhance other system capabilities such as semantic query optimisation.

The proposed solution is to represent integrity constraints explicitly as an additional facet attached to the attributes. A *constraint equation* approach [Morgenstern 84] has been chosen to represent integrity constraints. For instance, constraining the set of projects which a lecturer has a responsibility for, to be the same as the set of projects his/her research-assistants works on, can be expressed as:

```
projects OF lecturer :: projects OF research-assistants OF lecturer
```

Constraint equations (CEs) provide a formalism that closely mimics the structure of the OO model. As the user has to navigate through the OODB following different links, CEs are expressed by means of chains of relationships called *paths.* This formalism is already familiar to users who do not have to express constraints in a different paradigm such as first order logic. Another topic that has to be addressed is constraint inheritance. Constraints applicable to a class should also be enforced for its subclasses where further constraints can be specified. An approach based on a rule mechanism has been chosen to provide the right behaviour.

Once constraints have been explicitly specified by the user as constraint equations, they can be used for several purposes. The constraint maintenance mechanism can generate code to enforce a given constraint, whereas the query optimiser can use the same

declaratively-stated constraint to improve system response. In this way reusability has been improved: "Knowledge removed from a particular procedural context is free to be used in many contexts, within the existing application and in others, often in ways not foreseen when developing the original application" [Freundlich 90].

1.5 User-defined relationships

Whereas the abstract relationships, provided by SDMs, have a form of semantic definition attached to them, no mechanism is provided to describe the semantics of user-defined relationships. As a result, this semantics is still wired into the user programs instead of being in the schema definition of the DB. Hence, in the same way that SDMs provide a system maintained semantics for abstract relationships, allowing all users to have a clear notion of the consequences of DB manipulation, here a system maintained mechanism to specify the semantics of user-defined relationship is proposed. The DB designer should be able to specify the semantics of a relationship, e.g. what facts can be inferred as a result of relationship establishment. For instance, let *working-in* be a relationship between a *company* and a *person*. When a relation is established, can the *person* obtain his *working-address* attribute from the *address* attribute of the *company*? If the system does not specify anything, different users can have different understandings of what *working-in* a *company* means and different conclusions can be drawn. Hence, the semantics of insertion, deletion and modification operations have to be specified not only for abstract relationships but also for user-defined relationships. Enhancing the semantics of user-defined relationships helps to increase the UoD semantics kept in the DB as well as preserving the "behavioural" integrity of the system.

User-defined relationships have been represented either as pointers (i.e. attribute-values) or as aggregations (i.e. list of pairs). The former has several disadvantages such as:

- The inverse relationship is not expressed declaratively.
- The semantics of the relationship are split over methods attached to different objects.
- It is not clear where to put the attributes of the relationship.

For our research, an aggregation approach has been chosen that allows relationships to be seen as objects. Then, the entire semantics of the relationship can be kept in just one place, getting round of some of the problems of the pointer-based approach and is more in keeping with the object-oriented philosophy.

Traditionally, relationships have been characterised by the degree, the cardinality constraint, the participant objects and the attributes of the relationship itself. A new set of primitives, inspired by AI concepts, is proposed to enhance relationship semantics both in its static and in its dynamic aspects so that the DB designer can specify the intended meaning of the relationship instead of leaving the user to guess it. As a result, an increase in the semantics of the UoD preserved by the DB is achieved. Moreover, treating relationships as *first-class* objects allows the establishment of hierarchies, rules or even relationships among relationships.

1.6 Active behaviour

Active behaviour has been defined as behaviour exhibited *automatically* by the system in response to events generated internally or externally without user intervention [Bauzer 90]. Quite often this behaviour is associated with the event itself, i.e. encapsulated with the corresponding method (e.g. integrity constraint maintenance). Event-Condition-Action rules (ECA rules) have been proposed in [Dayal 89] for enhancing DBs with active behaviour in an explicit manner. An ECA rule is mainly described by *the event* that triggers the rule, *the condition* to be checked and *the action* to be performed if the condition is satisfied. The condition is a set of queries to check that the state of the DB is appropriate for action execution. The action is a set of operations that can have different aims, e.g. enforcing of integrity constraints, user intervention, propagation of methods, etc.

The research presented here is an attempt to represent rules in an OODB. The focus is on providing a uniform approach. What is meant by a uniform approach is that rules have to be defined and treated in the same way as other objects in the system, without defining any additional mechanisms or auxiliary structures. Rules are seen as 'first-class' objects, and are described using attributes and methods. In this way, rule management operations are conceived and implemented as methods. This brings all the advantages

of the OO paradigm into rule management: encapsulation, modularity, reusability. In a uniform approach the system should not distinguish rules from other kinds of object. As a result, rules can be related to other objects, and also arranged in hierarchies. Since methods attached to objects can trigger rules, and rules are themselves objects, rules can be defined which are triggered by methods attached to rules. As with any other entity, the meaning of a rule lies in the attributes attached to the rule, and their interpretation by the associated methods. From the point of view of the system however, no distinction should be made.

Nevertheless, rules can be difficult to define for non-experienced users. Hence, higher more-declarative primitives can be provided from where rules preserving the required semantics can be generated by the system. This approach is illustrated in the support of constraint equations and the operational semantics of relationships.

1.7 Overview of the thesis

- Chapter 2: The Object Oriented Paradigm.

  This provides a comparison of different approaches within the OO paradigm. Since this work has been influenced by ideas coming from Artificial Intelligence, Programming Languages and Data Bases, the aim of this second chapter is to provide a common base for readers with different backgrounds.

- Chapter 3: Object Oriented Databases.

  This introduces the principles of Object Oriented Data Bases. Several systems are described, ADAM among them. ADAM is used as an implementation vehicle to test the ideas presented in the rest of the chapters.

- Chapter 4: Making metadata explicit.

  This introduces the concept of metadata and how it has been accomplished in OO systems by introducing metaclasses. The approach followed by ADAM is carefully described and examples are shown of the resulting advantages, namely uniformity, accessibility and extensibility.

- Chapter 5: Making integrity constraints explicit.

In this chapter, a proposal is made to represent integrity constraint explicitly as attribute facets. Some of the problems attached to implicit constraint definition are outlined. An implementation in ADAM is presented.

- Chapter 6: Making user-defined relationships explicit.

  This introduces an approach for enhancing user-defined relationships in OO systems. Several examples of how relationships have been supported are presented and some of the new features worth considering are illustrated. How ADAM has been extended with a construct to represent semantic-rich user-defined relationships is described.

- Chapter 7: Making active behaviour explicit.

  This presents a rule manager where the idiosyncrasies of the OO approach have been considered. Several uses of active rules are given. An implementation of such mechanism in ADAM is discussed.

- Chapter 8: Conclusions.

  This presents the conclusions of the research and some ideas for the future.

Chapters 2 and 3 provide an overview of the area. Chapter 4 presents a description of metaclasses and an example of how metaclasses have been defined in ADAM by Norman Paton. Finally, in chapters 5, 6 and 7 the main contribution of this thesis is presented by illustrating the case for integrity constraints, user-defined relationships and active behaviour. In each chapter, a review of related work, the disadvantages of embedding knowledge within method code, a proposed solution and an implementation extending ADAM with convenient primitives is given.

Chapter 2

The Object Oriented Paradigm

2.1 Introduction

The object-oriented (OO) paradigm is characterised in a general sense by a grouping of information with the concept or entity to which it relates. One of the reasons for the recent interest in this paradigm is that it permits many concepts from the real world to be modelled in a direct and natural way. Indeed, in [Mylopoulos 90] a notation is said to be object oriented:

> "when it encourages a direct and natural correspondance between components of notation instances and objects of application."

In this sense, the relational data model can not be considered OO since an entity in the process of normalisation can be split between different tables. In addition, from an engineering point of view, the OO paradigm brings such benefits as reusability and extensibility through the inheritance mechanism.

Despite being a widely used paradigm, there is not yet a common understanding of what an object is. This stems principally from the different motivations underlying the distinct fields where OO-like systems have emerged. Research has been going on in the areas of DBs, AI and programming languages (PLs). As a result, a myriad of systems have appeared where a diverse terminology is used such as classes, instances, frames, terms, actors, entities... But, are these synonymous, similar or completely distinct? How is an object different from these terms? To provide some guidelines, instead of describing a set of specific systems, an analytical introduction is taken in this chapter.

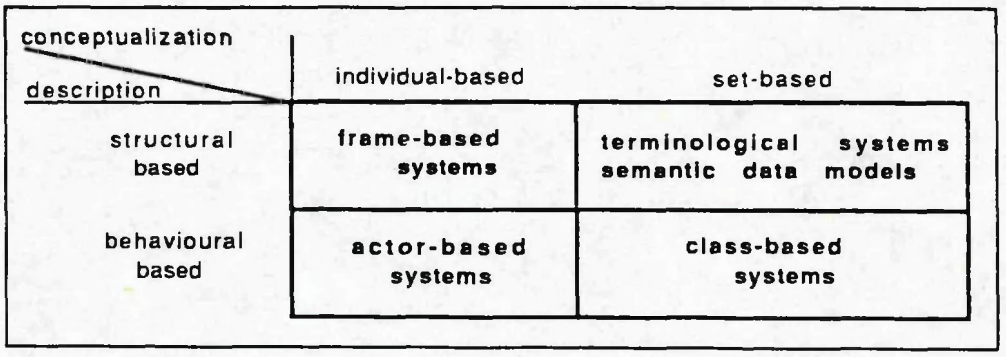


Figure 2.1: A classification of object-oriented systems.

Classification is always a risky undertaking since not everyone will agree on the criteria followed, and some hybrid systems are always difficult to pigeon-hole. However, a classification allows us to identify a common ground up which to compare different systems that, although sharing a similar terminology, are based on fundamentally distinct notions of objects, inheritance, etc. This is, after all, the aim of this chapter: to provide a common framework for readers coming from different backgrounds.

The classification is based on the following criteria:

- *what is the conceptualisation underlying an object*
- *how is such conceptualisation described*

These criteria distinguish four broad families, shown in figure 2.1.

The first group consists of those systems where the common description shared by a *set* of objects is placed on an abstract object. If the characteristics abstracted refer only to structural, relational and attributive features the term **entity type** is normally used for this abstract object. This approach is mainly taken in the area of *Semantic Data Models (SDMs)*, and hence these are sometimes referred to as *structurally object oriented models* [Dittrich 86]. SDM [Hammer 81], TAXIS [Mylopoulos 90] and IFO [Abiteboul 87] are well-know examples of this family. In AI, *terminological languages* also follow a structural definition of concepts. Following the nomenclature of KRYPTON [Brachman 83], a distinction is made between the terminological (TBox) and the assertional (ABox) part of the knowledge base. **Terms**, discribing the terminology of the UoD, are specified by stating superconcepts and restrictions on other concepts called *roles*. Thanks to their enhanced structural definition capabilities, terminological languages profit from a richer

ability to reason about structure than is commonly found in SDMs. Systems such as KL-ONE [Brachman 85] and BACK [Peltason 89] belong to this area.

If the abstraction encompasses not only the structural features but also the behavioural ones, the term **class** is used. A new range of topics appears concerning the use and reuse of behaviour in class hierarchies. This approach has mainly been followed by PLs and, more recently, DB practitioners. SMALLTALK [Goldberg 83], C++ [Stroustrup 86], Eiffel [Meyer 88] in PLs, and O_2 [Deux 90] and ADAM [Paton 89b] in DBs are examples of class-based systems.

The second main group is formed from those systems in which objects have a more individual existence and they do *not* have to conform to some abstract definition. In AI, the term **frame** is used to name the structural, attributive and relational properties of a *prototype*. A frame no longer represents the necessary and sufficient conditions of other objects in the system, just default properties. Besides its own properties, a frame can inherit, specialize or override properties from another frame. Well-known frame systems are FRL [Goldstein 77] and CRL [Carnegie 85].

Within this group, other systems are more interested in reflecting behaviour. In this case, the term **actor** is used. An actor is characterised by a set of messages to which it can reply, where a *script* describes how a given message should be answered. The main aim of these systems is the development of open systems and concurrent programming. ACTORS [Agha 86] is an example of an actor-based languages.

In the rest of the chapter these categories are described. For each category, a definition of the concept is first presented, followed by a description of its characteristics (structural or/and behavioural) together with some examples of the intended use. Lastly, a conclusion is presented where the different approaches are compared.

2.2 Class-based systems

A set can be defined explicitly by enumeration of all its members or implicitly by specifying the properties held by any object of the set. In class-based systems, the common description of a *set* of similar objects is held in a **class**. Its members, called **instances**, *must* conform to the description given in the class. Thus, a class can be defined as an

implicit definition of a set that incorporates not only the structural features of its members but the behavioural ones as well. So two components are distinguished in a class definition, namely:

- the static component, described by *instance variables*
- the dynamic component, described by *methods*

Both method and instance variable descriptions are kept at the class level, but each instance retains its own values for the instance variables, representing the state of the instance. As an example, a class and instance definition in ADAM could be as follows:

```
:- new([student,[
    is_a([person]),
    slot(slot_tuple(cname,global,single,optional,string)),
    slot(slot_tuple(age,global,single,optional,integer)),
    method((studying(global,[],[integer],string,[Hours,Result])
       :-  % here is the implementation
           ......................))
  ]]) => class.

:- new([OID,[
    cname([john]),
    age([25])
  ]]) => student.
```

The *student* class holds the definition of both the *studying* method and the *cname* and *age* instance variables, whereas a specific student keeps the specific values of its instance variables (e.g. *john* and *25* for the *cname* and *age* variables respectively).

Instances can be created by sending the message *new* to a given class. At this time, *the system* assigns a *unique object identifier* to the instance just created. This is a core concept in class-based systems whereby instances can be uniquely referred to within the object space [Khoshifian 86]. An object keeps its object identifier throughout its life regardless of the changes it undergos. In the previous example, the variable OID is assigned with the identifier of the object just created at run time.

A method definition encompasses:

- the name of the method, also called *selector* (e.g. *studying*)

- the arguments of the method (e.g. *Hours* and *Result* are integer input and string output parameters respectively)
- the implementation of the method, encapsulated within the class

Methods are always referred to by their selectors. It is important to distinguish between the selector and the method itself since there may be different method definitions with the same selector, a phenomenon known as *overloading.*

When a method is to be executed for a given instance, most of the systems send a *message* to this instance. A message is formed by the selector, the set of actual arguments and the instance itself. As an example, consider a class *person* with a method *increase_salary* that has as an argument the amount of the increase. Let T be a variable holding an instance of *person.* An increase of 10 in the salary of T can be achieved in the following way:

- T increase_salary: 10 (SMALLTALK)
- T.increase_salary(10) (C++)
- increase_salary([10]) => T (ADAM)

The selector is bound to the corresponding method at run time, unlike conventional languages where such binding occurs at compile time. This process is known as *dynamic binding* or *late binding.* Although dynamic binding slows down the system because checking has to be done at run time, it supports overloading and make it possible to ignore the instance class till execution time. This occurs in untyped systems such as SMALLTALK.

Encapsulation is a major feature of these systems, whereby the structure of an object is transparent to the user who accesses the object only through methods. So an object is characterised by the set of operations which can be performed on it, the so-called object *interface* or *protocol.* In this way, classes inherit the ideas of *abstract data types* where a clear distinction is drawn between the interface and the implementation of the data type. Thus, the structure of an object is always hidden from outside. In ADAM, a set of methods to delete, update, put and get the value of an attribute is automatically generated by the system when the attribute definition is entered. As a result, the user ignores the internal implementation.

An interesting issue arises when comparing classes and *types*. Both have a similar specification: they describe the attributes and operations shared by a set of objects. However, as pointed out by [Bancilhon 88], they embody different notions. A type is more a *compile time* concept used to check the correctness of programs. Strongly typed languages can detect wrong assignments or invalid actual argments when the program is compiled. In contrast, a class is a more *run time* notion where two aspects can be distinguished: a template whereby replicated objects can be created (*the object factory*) and a repository of objects belonging to the class, i.e. the class extension (*the object warehouse*) [Bancilhon 88]. The latter is specially important in DBs where set-based operations are performed on collections of objects.

Untill now most of the concepts introduced for classes are also applicable to abstract data types. The real step ahead comes with the introduction of *class hierarchies* and *inheritance*. The idea is that instead of having an isolated class collecting all the information, a class hierarchy can be defined where *subclasses* can be obtained by specialising a more abstract class called the *superclass*. Subclasses hold only information that is specific to themselves, inheriting more general descriptions from their superclasses. Besides organising information in a familiar way (after all, animals have been classified in taxonomies for decades) inheritance also accounts for software reusability and extensibility. A subclass can simply inherit or specialize the definition kept in its superclass without starting from the beginning.

Classes can be specialised either by adding new properties to the class (either instance variables or methods) or by constraining previous specifications. In [Khoshifian 90] a categorisation of the different specialisation alternatives allowed by PLs can be found. The two extreme policies are to forbid any sort of redefinition and to allow arbitrary redefinition. Whereas the former guarantees strong typing, the latter provides more flexibility at the expense of an overhead at execution time.

Inheritance is an area of active research where a broad set of issues arises such as subtyping, encapsulation, multiple inheritance, etc. For a friendly introduction to these topics see [Khoshifian 90]. A broader discussion including knowledge representation formalism can be found in [Lenzerini 91]

2.3 Semantic Data Models

The trend followed in PLs, whereby implementation details are hidden from the user providing higher abstraction tools, has a counterpart in DBs. PLs have evolved from assembler languages to ALGOL-like languages and currently to OO languages implementing abstract data types. Paralleling this development, file systems evolved to DBs where data models are provided to capture the structure of the UoD, ignoring the physical details. However, traditional data models, i.e. the hierarchical, network and relational data models, are still very much record-oriented. As a result, the objects of the UoD can not be directly mapped into data model primitives, and users have to reconstruct objects by navigating throughout the DB, assembling (foreign key joining) the different pieces of the object spread in several records. In contrast, SDMs attempt to achieve a more one-to-one mapping between concepts of the real world and how they are represented in the computer, by providing a set of structural abstractions. In this way SDMs fulfill the requirement given by Mylopoulos [Mylopoulos 90] to be object-oriented notations. In the following, a set of abstractions generally considered in SDMs are described along with some behavioural features.

The structural abstractions that can be found in SDMs are the following where definitions are taken from [Peckham 88]

- *"Generalisation is the means by which differences among similar objects are ignored to form a higher order type in which similarities can be emphasised."*
- *"Aggregation is the means by which relationships between low-level types can be considered a higher level type."*
- *"Classification is a form of abstraction in which a collection of objects is considered a higher level object class."*
- *"Association is a form of abstraction in which a relationship between member objects is considered a higher level set object."*

These abstractions isolate the user from physical details and also allow the semantics explicitly represented in the DB to be increased. For instance, now the system knows what a generalisation means and can preserve its semantics. Hence, the burden of maintaining these structural abstractions is moved from the user to the DB. Of course, a clear

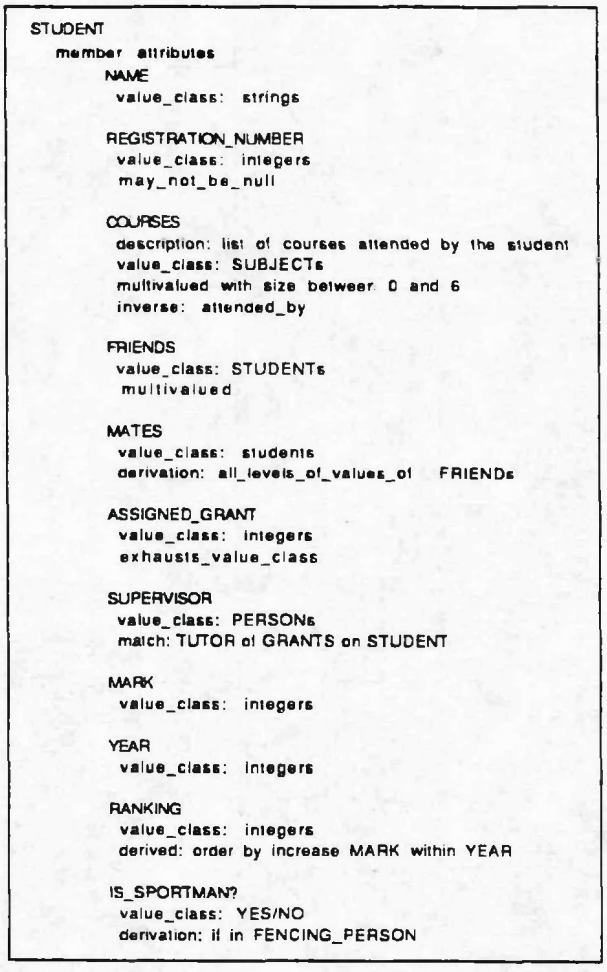


Figure 2.2: A student type definition in the SDM data model.

semantics must be defined for each abstraction by specifying how insertion, deletion and modification operations made at a higher level can affect the object abstracted and vice versa [Hull 87]. For the SDM data model, these semantics in a rule-based format can be found in [Amy 89].

SDMs are mainly concerned with the structural description of the UoD. However, besides the operational semantics mentioned above, some sort of behaviour is considered in the so-called *derived attributes* and *derived classes*. A derived attribute can be defined as a property not stored but obtained from other attributes in the DB. In the SDM data model [Hammer 81], a broad range of possibilities is available to specify the derivation criteria for such attributes. As an example, consider the definition of STUDENT in the SDM data model given in figure 2.2. The following criteria are illustrated:

- *Ordering criteria.* For example the attribute *ranking* is defined as the sequential position of the student among the students within the same *year* considering the *mark* obtained.
- *Boolean criteria.* The attribute takes the value 'yes' if the object is a member of a given class, and 'no' otherwise. For instance, *is_sportman* has a derivation criteria to check whether the student is a member of the *fencing_person* class.
- *Tracing criteria.* The value of the attribute is the transitive closure of a given attribute. As an example, the attribute *mates* reflects the fact that "all mates of my mates are also my mates".
- *Arithmetic criteria.* The attribute is obtained as a result of some arithmetic operation on other attributes. Aggregation functions can also be applied.
- *Set criteria.* The attribute is derived as a result of some set operations on multi-valued attributes.

Figure 2.2 also shows the expressiveness of the SDM data model in attribute definition. Besides the type of the value, other properties that can be specified are,

- cardinality, e.g. the *courses* attribute,
- inverse constraints, e.g. the *courses* attribute,
- matching constraints i.e. restricting the value of the attribute to match the value of another attribute in the system, e.g. the *supervisor* attribute, where *grants* is another class having *tutor* and *student* as attribute,
- the not-null constraint, e.g. the *registration_number* attribute,
- the exhaustive constraint where "every member of the value class of the attribute (call it A) must be the A value of some entity" [Hammer 81], e.g. the *assigned_grant* attribute where it is required that every grant is assigned to some student.

The SDM data model also offers some further constructs to define derived classes, namely:

- *Attribute-derived* e.g. the *fresher* subclass can be defined as *students* having "1" as the valued of the *year* attribute.

- *Set-operator-derived* where subclasses can be defined as the intersection, difference or union of other subclasses.
- *Existence-derived* where the members of the class are the set of values of a given attribute of another subclass. For instance, the *grant* class can be defined as the values of the *assigned_grant* attribute.

Some SDMs have also addressed the description of the dynamic aspects of the UoD (e.g. TAXIS [Mylopoulos 86], SHM+ [Brodie 84b] and *the event model* [King 84]). In TAXIS *transactions, exceptions* and *exception handlers* are manipulated as detached entities. As a result, they can have attributes and be arranged in hierarchies like any other entity. In TAXIS a transaction can be described in terms of the entities involved (i.e. *the parameters*), the type constraints on the participant entities, the initial, final and invariant states defining the context, and the set of subactions that comprise the definition of the transaction. Furthermore, since transactions are parameterised by the entities involved, transactions can be specialised along with the entities, to considerer special requirements for the specialised subclasses. In this way, TAXIS provides a "stepwise refinement by specialisation", i.e. a methodology to describe the dynamic aspects of an application, in contrast with the "stepwise refinement by decomposition" used in traditional programming [Borgida 84]. Here, the refinement is driven by the entity hierarchy instead of any functional refinement.

Exceptions and exception handlers are also considered as TAXIS entities. This allows one to disregard exceptional circumstances right at the beginning, postponing anomalous situations till the design is further developed.

As in TAXIS, in SHM+ [Brodie 84b] the structure of the data *drives* the description of the activities. Nevertheless, the structuring mechanisms in SHM+ include not only generalisation, but also aggregation and association. As pointed out by [Borgida 84] both data and behaviour can be structured in terms of the same mechanisms, providing a well-integrated methodology for designing DB applications. Hence, these models can be considerered as a step ahead in data modelling, allowing a wholistic view of the application by not only designing but also integrating the static and dynamic aspects of the UoD.

Initially used as design tools, SDMs have become available as Semantic DBMS (e.g. SIM

from Unisys Corp. based on the SDM data model). They provide a richer set of primitives to capture the UoD. Increasing data independence, enhancing user productivity and understanding, and providing modelling flexibility can be seen as the main advantages brought to DBs from this research area.

Diverse overviews of SDMs can be found in the literature. In [Hull 87] several models are compared based on a common example, making it a good pedagogical introduction. In [Urban 86] an analysis of the structural, dynamic and temporal aspects of SDMs can be found. In [Peckham 88], SDMs are compared based on "the support of relationships, the abstraction they represent, the manner in which the semantics are specified and the approach (if any) to dynamic modelling" [Peckham 88]. Finally a comparison of advanced SDM, namely TAXIS, ADAPLEX and GALILEO can be found in [Albano 89].

2.4 Frame-based systems

The concept of *frame* was first proposed by Marvin Minsky, influenced by psychological theories, as a knowledge representation paradigm to understand the efficiency and effectiveness of human reasoning [Minsky 75]. He defines a frame as:

> "... a data structure for representing a stereotyped situation... Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed."

The idea is that when human beings have to cope with new situations, they do not begin from scratch, but attempt to used their past experience to face the new context. So when an unknown object is confronted, human beings recognize that it is a person thanks to a set of features characterising the stereotype of a *person*. Such concept is know as a **prototype**.

A frame is described by *slots* which reflect *attributive, structural* and *relational* information about the frame. For instance in CRL (Carnegie Representation Language), a frame-based language within the Expert System development environment Knowledge Craft (a good introduction can be found in [Kingston 87]), the frame *person* can be defined simply as:

```
(defschema PERSON
```

```
(CNAME)
(AGE))
```

The slots themselves are described by a set of *facets* which are grouped in turn into frames (called *slot_control_schematas* in CRL). An example of such a frame for the slot *age* can be the following,

```
(defschema age
   (IS-A slot)
   (DOMAIN person)
   (RANGE integer)
   (CARDINALITY (0 1))
   (DEMON age_constraint))
```

In general, the following items can normally be specified to describe a facet:

- *domain*, specifying in which frame this slot can appear,
- *range*, restricting the values that can fill the slot,
- *cardinality*, restricting the minimum and maximum number of values that the slot can have,
- *value*, holding the current value,
- *default* indicates the value used when *no* other information is available. Notice that it is not an initialisation value as found in class-based systems, where an attribute can be initialised at the moment an instance is created and thus this value is owned by the instance. Default values on the other hand, stay with the generic prototype and are *dynamically inherited* when required,
- *attached procedures* (sometimes referred to as *demons*). Unlike methods that are explicitly invoked, attached procedures have a kind of event-driven behaviour, i.e. they are fired as a result of event detection rather than by explicit calls. Different kind of procedures can be found depending on the associated event, namely:
  - *if-added*: the procedure is executed when a new value is inserted. It implements a kind of *forward-chaining* process,

- *if-needed*: the procedure is executed when a request to retrieve the value of the slot has been issued but its value is not available. It implements a kind of *backward-chaining* process.,
- *if-removed*: the procedure is executed when an attempt to delete the slot value is detected,

In CRL an attached procedure can be defined as follows:

```
(defschema AGE_CONSTRAINT
    (INSTANCE demon)
    (ACCESS add-value)
    (WHEN before)
    (EFFECT block)
    (ACTION checking_age_constraint))
```

This demon is fired *before* any value is *added* to the slot *age*. The demon executes the function appearing as the filler of the slot *action*, in this case the LISP function *checking_age_constraint* and avoids the insertion of the *age* if required [1] .

It is worth noticing that attached procedures are the only way of representing behaviour in frame-based languages. Thus, unlike class-based languages, a frame does not have its own behaviour described in the form of methods but all behaviour is associated with the slots.

In addition, prototypes can be specialised. For example, we can have the *student* and *lecturer* prototypes as specialisations of the *person* prototype, where additional properties are defined. In this context CRL is more powerful than similar systems since it allows one to customize inheritance for special requirements. So the user can tailor his own particular sort of inheritance by explicitly stating:

- which slots and values from the generic frame are inherited unchanged by the specialised frame,
- which slots and values are not inherited,
- which slots and values are introduced when the relationship is created,

[1] In fact, the LISP function is quite complicated since the block effect is done regardless of the result. So if the constraint is satisfied, it is the function's task to guarantee that the value is inserted.

- a one-to-one mapping of slots and values from the generic to the specialised frame

The motivation for this explicit declaration of inheritance, is that a small set of inheritance types is not enough. As pointed out in [Fox 79] "in some cases the inheritance relation will have to be specialised to the particular concepts they relate. Hence, the inheritance link is context sensitive". This can sound odd to DB people but it has proved useful in Expert Systems [Fox 86] and it has influenced our work on user-defined relationships [Diaz 90].

However, it has to be underlined that in frames sharing is based on the prototype theory. An object can inherit properties from its prototype if *no information of its own is available.* Otherwise the properties of the prototype are overridden. Hence, unlike the well-structured applications of class-based systems, frames are used to capture all the complexity of the real world where exceptions to the inheritance between the generic prototype and the specific exemplifications can arise. Penguins as birds which cannot fly, and whales as mammals which live in water are well-known examples of the complex problem of exceptions. For instance, consider the prototype of *person* as having a permanent address. However this property can be overridden by *John* who is nomadic and likes to know how many kilometres he has travelled. In CRL this can be reflected as:

```
(defschema person1
    (INSTANCE person)
    (CNAME john)
    (KM_COUNTER 1268)
```

Although *person1* is an *instance* of *person*, he adds a new attribute *km_counter* just for himself. Therefore, prototypes do not play the role of templates to which all the 'instances' have to conform. Moreover, it has to be pointed out that the frame identifier (e.g. *person1*) is provided by the user, unlike class-based systems where such an identifier is provided by the system.

Furthermore, unlike class-based systems, in frame-based systems any object can become a prototype. Whereas in the former there is a clear separation between objects that can generate other objects -class descriptors- and objects that cannot -instances-, in frame systems such a difference does not exist. Any object can become a prototype and other objects can inherit from it. Therefore, the *is_a* and *instance* relationships found in CRL can be seen as a specialisation and exemplification of prototypes, respectively, rather

than the subset and membership relations found in class-based languages. For a more detailed comparison between ADAM and CRL refer to [Paton 91].

The different understanding of what an object represents also leads to distinct ways of working in each system. For instance, the process of categorising an object in a class-based system is completely expressed by sending the message *new* to a given class, where this class is known in advance. By contrast, an equivalent process in a frame system can be much more complicated. Expert systems based on classification spend more of their time ascertaining which of the prototypes already known by the system is the one that best matches the properties of an unknown object. This process, known as *frame matching*, has been extensively used in diagnostic systems. An abnormal setting (e.g. a disease or any damage in a system) is represented as a frame whose slots are expected malfunctions of the system (e.g. the steam temperature or the count of red blood cells of a patient). The expert system then has to find the frame that best matches the current situation. In this process, arranging frames representing prototypical abnormal situations in a taxonomy, not only improves reuse but helps in the order of searching for information. Due to the quantity and different qualities (e.g. cost, danger or importance) of the data to be considerered in the process of solving the problem, one of the main difficulties in diagnostic systems is to decide which are the symptoms to consider next. Building a taxonomy from specifically diagnosable diseases (e.g. cirrhosis) up to more abstract diseases (e.g. liver illness) allows factoring out common symptoms, and then, to proceed in a top-down fashion, imposing an order on the consideration of the symptoms. In addition, similarity networks can be used to support differential diagnosis "which is the art of selecting those questions to ask (or to test to perform) that best differentiate among competing hypothesis" [Szolovits 86]. A well-known medical diagnostic system that works in this way is MDX [Chandrasekaran 83].

2.5 Terminological systems

Frame-based systems were critised for their lack of formal semantics. There was no clear understanding of the meaning of a frame, and a formal interpretation of frame manipulation was missing [Brachman 83, Brachman 85]. In [Woods 75] the distinction between *structural links* used to describe and *assertional links* used to make statements, is pre-

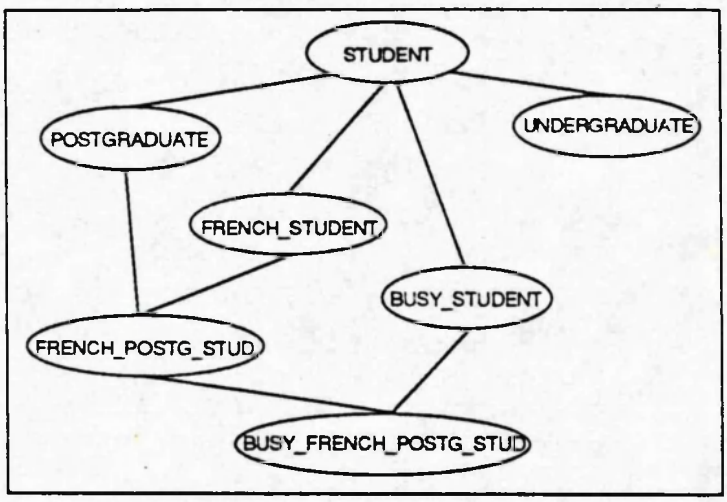


Figure 2.3: A messy hierarchy.

sented. In Wood's example a representation for a telephone with a slot containing the colour 'black' is given. The author points out that it is not clear whether 'black colour' is a description of the concept *black telephone* or an assertion that telephones are black. In [Lambert 88] an example is given of an 'eclectic' hierarchy similar to the one shown in figure 2.3, where to ask the kinds of *students* represented become meaningless. As Lambert points out "the fact is that frames, especially with the structural interpretation, are crying out to be manipulated as data structures". The problem is that the user has a direct access to the data structure and so changes can be made, disregarding or ignoring the consequences for the knowledge that the data structure is supposed to represent.

This leads to the *functional approach* in knowledge representation, where schemas are always manipulated through functions provided by the system, and hence no direct access is allowed to the underlying data structure implementing the schema. This resembles the aims of abstract data types, where an object is characterised by the set of operations it can undertake, preventing the user from directly accessing to its hidden implementation. Now, "in knowledge representation, it is a whole knowledge base with associated operations that may be regarded as an abstract data type" [Lambert 88].

These ideas are crystallised in *terminological languages.* These systems are characterised by the set of constructs and querying functions provided. Two components are distinguished:

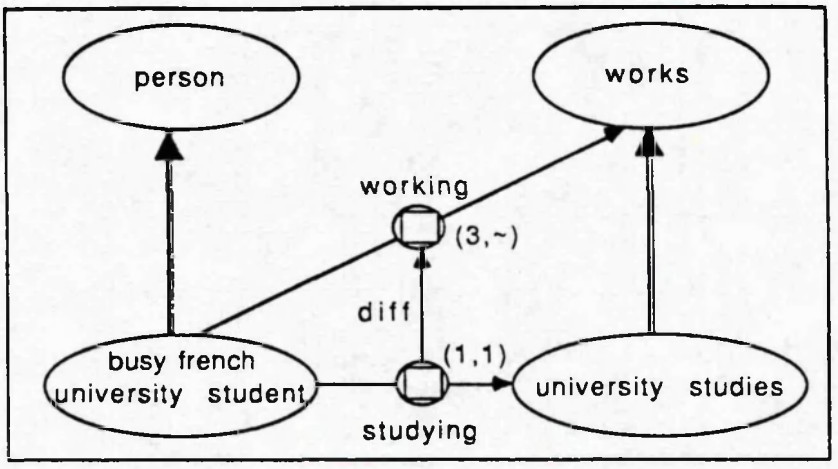


Figure 2.4: A KL-ONE notation for the busy French university student concept.

- the *TBox* collecting the concepts or **terms**. This defines the terminology of the UoD.
- the *ABox* making **assertions** about the terms in the TBox.

As is suggested by the word 'box', neither the terminological component nor the assertional one are directly accessible. A set of functions are provided for introducing new elements (the TELL side of the interface) or for asking questions about these elements (the ASK side of the interface). In this way, a knowledge base is characterised by the kind of questions it can answer, referring not only to facts or terms explicitly stated, but also to any kind of inference drawn by the system. In the following, the TBox and ABox components are presented for the system BACK [Peltason 89].

Term definition (i.e. the TBox TELL interface) is achieved by stating superconcepts and restrictions, of value or cardinality, on relationships to other concepts called *roles*. So *terms are fundamentally structurally defined*. As an example, consider the definition of a *busy French university student*. In figure 2.4 a diagram representing this term is given. Assuming that *person*, *works* and *university_studies* are concepts already introduced, and that *studying* is a subrole of *working*, the definition can be as follows in a terminological language such as BACK:

```
:- tboxtell(
      busy_french_university_student :=
         person
         and atleast(3,working)
         and atleast(1,studying)
```

```
and atmost(1,studying)
and all(nationality,french)).
```

In the above definition, *and, all, atmost* and *atleast* are all BACK constructs with a clear system-defined semantics. Term definition then, is always done through these constructs. No direct manipulation is allowed.

As a result, the system can reason about terms based on these well-established semantics, where *the meaning comes from the structure.* The system can then establish relationships based on such structure. As an example, these are some predicates provided by the TBox ASK interface of BACK [Peltason 89]:

- *equivalent(T1,T2)* is a predicate that is true if T1 subsumes T2 and T2 subsumes T1.
- *defined_as(Term-name,Term)* obtains in Term the definition of Term-name,
- *describe_fully(T1,T2)*, the second argument retrieves the description of the first one, using all primitive superconcepts plus all role restrictions which apply to the concept to be described,
- *difference(T1,T2,T3,T4)*[2] takes two terms, the first two arguments, and computes their conceptual difference, expressed by another two terms, the third and fourth arguments.

But the most important relationship among terms is called *subsumption* which is established between the so-called *defined concepts.* A distinction is made between *primitives* and *defined* concepts made -firstly made by KRYPTON [Brachman 83]. *Primitive concepts* are described using only *necessary conditions*, whereas *defined concepts* are specified by *necessary* and *sufficient* conditions. For instance if the *busy French university student* term were a *primitive concept*, then for anything to become a member of this concept it would have to be explicitly stated. Since instead it is *defined*, the system itself can infer the membership. As an example, if a *French person* is introduced *working* in four places where only one is an *university_studies* then, the system can *infer* that this assertion is an instance of the *busy french university student* without being explicitly declared.

[2]Not yet implemented.

Among *defined concepts* the *supsumption* relationship can be established. A concept *C1* subsumes a concept *C2*, if the extension of *C1* (i.e. the set of assertions of C1) is a super set of the extension of *C2*. However, such relationship is inferred without looking at the extension but based on the definition of the concepts. For example, the concept *person* subsumes the concept *busy French university student* since all busy French university students must, of necessity, also be persons. This allows *the system* to keep a term hierarchy from more general concepts to more specific ones. The process of inserting a new concept in the hierarchy is known as *classification*. The crucial point here is that such a hierarchy is maintained by the system. It is up to the system to decide where to place a given concept in the hierarchy, unlike in frame systems where the user takes such decision with dangerous consequences (remember figure 2.3). Classification is not restricted to the TBox. As previously shown, given an assertion it is up to the system to decide which is the most specific concept of which this assertion is an instance. *The user cannot decide where an assertion is placed.* However, classication also constitutes the weakness of terminological systems due to the time consumed in its realization. Notice that classification takes place everytime a new assertion is inserted!

It is worth noticing that the idea of defined concepts is missing in most of the semantic data models (the SDM data model is an exception). A class represents a primitive concept where the user has to specify explicitly the class members. This means that is not possible to use the class as a *recognizer* such as by creating the class *busy_student*, and later retrieving all the *students* with at least three *works* just by querying for the extension of this class. So, although in some semantic data models attribute cardinality can be specified, it is used only as an integrity constraint of the data stored, rather than a filter of the class instances [Borgida 90].

The assertional component is represented by the ABox where statements using TBox terms are made. Several approaches have been followed: from full first order logic as in KRYPTON to object-centered schemas as in KANDOR [Patel-Schneider 84]. In BACK a middle approach is preferred, where assertions have an identifier but where role fillers can be incomplete. Supposing a role *attending* is defined between the *busy French university student* term and the *courses* concepts, then here are some examples of the ABox TELL functions provided by BACK:

- Asserting the values of a role,

```
:- aboxtell( X = busy_french_university_student
     with name = 'claudine'
     andwith studying = 'computing_science'
     ..............
     andwith attending = 'programming' and 'databases'
  ).
```

The role *name* is applicable since any of these students is also a person, and the term *person* can have *name* as a role. The variable X will be instantiated with the identifier of this assertion.

- Incomplete information: stating the cardinality of a role filler,

```
:- aboxtell( X = busy_french_university_student
     with name = 'yves'
     andwith studying = 'computing_science'
     ..............
     andwith attending = card(2,0)
  ).
```

Here the name of the courses is ignored but the fact that *yves* is attending atleast two courses is introduced.

- Introducing the exhaustiveness of the provided information

```
:- aboxtell( X = busy_french_university_student
     with name = 'nicolas'
     andwith studying = 'computing_science'
     ..............
     andwith attending = close('databases')
  ).
```

Here the assertion refers to a student named *nicolas* that attends a course in *databases* and this is the only course that follows.

A set of functions is also provided for the ABox ASK interface. Here are some examples from BACK:

- Retrieving all the assertions of a given concept,

```
:- aboxask(Students = getall busy_french_university_student).
```

The variable *Students* is instantiated with the identifier of the different assertions verifying the query.

- Obtaining all the students attending *databases* or *expert-systems*,

```
:- aboxask(Students = getall busy_french_university_student
              with attending: 'databases' or 'expert-systems').
```

A big issue with terminological languages is the trade-off between expressive power and computational tractability. Although enhancing expressiveness is important to capture the UoD more accurately, knowledge-based systems have also to respond to queries and draw inferences in an efficient way. Unfortunately, the complexity of the algorithms supporting these inferences, increase with the expressiveness of the language [Levesque 85]. Hence, research is going on to find a balance between expressiveness and computational tractability [Willians 88].

2.6 Actor-based systems

These are based on ideas first proposed by Carl Hewitt [Hewitt 77]. An **actor** can be defined as an isolated agent which knows how to perform a given task. The *know-how* is distributed among the actors which cooperate to solve a given problem and where different actors can work in parallel. Actors are independent and active objects that communicate freely with their neighbours to accomplish tasks. Open, distributed systems and concurrent programming can be seen as the main experimental interests of these systems.

An actor collects some knowledge and details of how and when to use it, namely,

- the other actors known directly, called *acquaintances*. They play the role of instance variables.
- how and when a given behaviour is to be undertaken. Unlike classes, behaviour is not described by methods but by a *script* which filters the messages and activates the corresponding behaviour.

An actor is characterised by the messages filtered by its script, known as the *intention* of the actor. Implementation details are hidden from other actors, and thereby the

encapsulation principle is adhered to.

All these features have been summarised by Henry Lieberman in the following famous words,

- *Liberté:* actors are autonomous entities.
- *Egalité:* all actors work at the same level. No privileges.
- *Fraternité:* actors cooperate to solve a given problem.

Although they can be seen as a set of experts working in parallel, actor systems differ from *blackboard architectures* found in AI [Nii 86] in that they do not have a common pool of information reflecting the state of the problem (i.e. the blackboard). All processes are driven by message passing. Also, unlike class-based systems, actors are not abstracted in higher concepts such as classes. Rather they represent the idea of prototypes.

Sharing is achieved by *delegation*, i.e. when an actor receives a message it does not know how to answer, it forwards the message to another actor. Delegation as inheritance, can be seen as a means to share behaviour: a specific actor can have a specialised behaviour and delegate more general messages to another actor representing a more abstract concept.

There has been a lot of discussion about inheritance versus delegation [Stein 89]. Initially, delegation was seen as a more powerful mechanism since it can model inheritance and it allows one to have not only behaviour but also states (i.e. the acquaintances). However, in [Stein 87], it was shown that, in a system providing metaclasses, classes can be used to model delegation, where class variables are used to represent properties of the 'actors'. Since, unlike instances, classes can be arranged in hierarchies, class variables and class methods can be inherited among classes, thereby providing similar features to actor systems.

In [Stein 89] *the Treaty of Orlando* is presented where the different sharing mechanism are analysed and characterised along the following dimensions:

- *Static versus Dynamic:* is sharing determined when an object is created (static) or when the object receives the message (dynamic)?

- *Explicit versus Implicit:* are there primitives that allow the sharing to be explicitly fixed (explicit) or do the system primitives enforce a uniform mechanism (implicit)?
- *Per group or Per object:* is sharing specified for a set of objects (classes) or for individual objects (actors).

Inheritance can be seen as a kind of static, implicit and per group sharing mechanism, whereas delegation is dynamic, explicit and per object. Which is better? As stated in the treaty [Stein 89]:

> "... no definitive answers as to what set of these choices is best can be reached. Rather, that different programming situations call for different combinations of these features: for more exploratory, experimental programming environments, it may be desirable to allow the flexibility of dynamic, explicit, per object sharing; while for large, relatively routine software production, restricting to the complementary set of choices -strictly static, implicit, and group-oriented -may be more appropriate."

These ideas of flexible sharing are considered in chapter 6 when the operational semantics of user-defined relationships are introduced. Such semantics can specify when a participant object in a relationship can delegate a given message to the other participant object through the relationship. This is quite a novel feature in OODBs: to allow sharing other than through *is_a* links.

2.7 Conclusions

Different structuring concepts have been presented that share the notion of object as the main building block in attempting to represent the concepts of the UoD directly. Nevertheless, important differences have been identified stemming from the distinct research interests driving each of these areas. As a result, it is not always easy to identify common ground since quite often the same terminology hides subtle differences. In figure 2.5 a somewhat forced comparison is shown regarding the concepts abstracted, the structural and behavioural features, the support for encapsulation and the sharing mechanism used.

Object-oriented programming languages are an area where the class-based approach has mainly been followed: they pay special attention to reusability and extensibility of code. The use of *mixins* in these systems is a case in point.

	CLASSES	FRAMES	TERMS	SDMs	ACTORS
CONCEPT ABSTRACTED	SET	PROTOTYPE	CONCEPT	SET	PROTOTYPE
STRUCTURAL ABSTRACTIONS	classification generalization	'specialization' exemplification (overriding allowed)	specialization	classification generalization aggregation association	all objects at the same level (egalite)
BEHAVIOURAL FEATURES	METHODS (at the class level)	PROCEDURAL ATTACHMENT (at the slot level)	———	derived attr. derived class transaction mod.	SCRIPT
ENCAPSULATION SUPPORT	YES	NO	YES	———	YES
SHARING MECHANISM	STATIC INHERITANCE	DYNAMIC INHERITANCE (prototype theory)	SUBSUMPTION	STATIC INHERITANCE	DELEGATION

Figure 2.5: A comparison among different object-oriented systems.

DBs have been more interested in the structural features of the UoD. However, new developments also consider the behavioural side of the application [Gray 92]. SDMs such as TAXIS and OODBs are examples that confirm such a tendency.

In contrast, AI systems, where the prototype approach can be found, are more concerned with providing flexibility rather than rigidly-defined structure, in order to cope with the richness and heterogeneity of the real world. Recent systems in this area have evolved towards the so-called terminological languages. Although a richer and safer structural mechanism is provided in these systems, they lose some flexibility, especially in representing defaults. Terminological languages have also influenced research in DBs where systems based on these ideas have been built such as CLASSIC [Borgida 90]. Nevertheless, the trade-off between expressiveness and computational tractability is the Damocles' sword of such developments.

Finally, actor-based systems are mainly concerned with distributed and concurrent issues.

To sum up, no approach can claim to be better than other in all respects. As the real

world has many faces, so different requirements suit distinct modelling tools.

Chapter 3

Object Oriented Databases

3.1 Introduction

DBs have been mainly used for general data management such as insurance and banking management. The simple structure and large quantities of data to be processed characterise these applications. Such features have lead to DBs which are concerned principally with physical issues to improve system performance rather than with the enhancement of the data model. Recently however, a new range of applications have arisen where complexity rather than large quantities of data is the outstanding feature. Common examples include CAD/CAM, hypermedia systems, office information systems and Artificial Intelligence. Among the new features required are:

- higher-level abstract constructs that allow an easier modelling of the UoD,
- versioning support. For CAD/CAM applications different versions of the design of an object can be required,
- support for temporal modelling to trace the evolution of objects along time,
- modelling not only of the structural features of the UoD but also the behavioural ones,

Further, due to an increase in the complexity of objects, traditional DB mechanism for transactions, recovery, security and concurrency previously applied to simpler units such as tuples, have to be reviewed.

Object-oriented database management systems (OODBMSs) are a promising direction in DB research to face some of the challenges of the new applications. Being object-oriented, they encourage a direct mapping between concepts in the real world and how they are represented in the computer, embodying both the structural and behavioural features of the UoD.

Compared with relational database management systems (RDBMSs), OODBMSs

- provide richer constructs so that objects do not need to be flattened to fit the data model,
- associate data items with the development of complete applications by enclosing structure and operations together. In this way, OODBMSs alleviate the *impedance mismatch* problem. This problem refers to the type clash that arises when a relational language is embedded within a conventional programming language. Then the set-at-a-time relational processing has to be converted to the record-at-a-time processing of conventional languages,
- have an independent system-maintained identifier, unlike RDBMSs where a value-based mechanism (i.e. through primitive attributes or keys) is used to identify any object (i.e. tuple) within the system,
- have a more navigational tendency in data manipulation unlike RDBMSs where manipulation is more declarative. This is clearly an advantage of the relational data model,
- do not have a formal definition yet. In contrast, the relational data model is based on a clear and formal theory provided by [Codd 79]

The lack of a formal definition for OODBMSs makes difficult to find a set of required characteristics to be held for any DBMS to claim to be object-oriented. This difficulty stems largely from the different areas that lead to OODBMS development. Indeed, DBs systems that claim to have object-oriented features have been built by enhancing OOPLs (e.g. *GemStone* [Copeland 84]), relational DBs (e.g. *POSTGRES* [Stonebraker 90]), from semantic data models (e.g. *SIM* [Jagannathan 88]), built from scratch (e.g. *ADAM* [Paton 89b]). As a result of such diverse origins, not all OODBMSs found in the literature share exactly the same features. Broadly speaking, an OODBMS can be defined as a

database management system (and thus, having persistency, transactions, concurrency control, recovery and so on) based on an object-oriented model, i.e. providing object identity, data abstraction and hierarchies.

In the following section these object-oriented concepts are addressed for DBs. In section 3, the OODBMS ADAM is introduced, illustrating previously introduced ideas. For an overview of database management see [Brodie 86c].

3.2 Object orientation in databases

In this section, some of the features outlined for class-based systems in chapter 2 are revisited, and put in a DB context. The concepts of object identity, data abstraction and hierarchies are further explained within a DB framework.

Object identity (OID) has been defined as "the property of an object which distinguishes each object from all others" [Khoshifian 86]. OID has been independently addressed by PLs and DBs.

In PLs, object identity is based on user-defined labels or memory references. This mixes addressibility (i.e. how to access an object in a given environment) and identity that is internal to the object and independent of how it is accessed [Khoshifian 86].

On the other hand, in relational DBs a value-based approach to object identification is chosen where a set of attributes (called *primary attributes*) uniquely identify a given object within a set. Some problems arise with this approach, namely,

- primary attributes are not allowed to change despite being descriptive properties of the object,
- attributes represent a partial view of the *real* object, the view of interest for the applications. Hence, it could not always be possible to find a set of attributes whose values are unique for each object in the UoD. For instance, if the *name* and the *age* are the only attributes of interest for a *person*, we should not be forced to introduced the *social security number* only for the sake of having a unique identifier,
- the set of primary attributes can change

Thus, neither of these proposals are satisfactory since the concept of identity is mixed either with the concept of 'addressibility' or data value. In contrast, object-oriented systems provide an OID *internal to the object* and *maintained by the system.* Such an identifier is kept by the object throughout its lifetime regardless of the changes to the properties of the object.

Different approaches have been followed for building OIDs in DBs [Bertino 91]:

- The OID is formed from the class identifier and the instance identifier. For example in ADAM, *23@person* is a correct OID, where the '@' symbol separates the name of the class (e.g. *person*) from the instance number within the class (e.g. *23*). When a message is sent to an instance, the system obtains the class identifier from the object identifier to access the method directory of this class. As pointed out by [Bertino 91], this approach jeopardizes object migration from one class to another. Since migration involves a change in the object identifier, previous references to this object also need to be changed.

- The OID does not refer to the class which is kept as control information inside the object. When a message is sent, the object is retrieved and the class obtained from it. However, this approach implies an overhead even for nonvalid messages, and type checking becomes quite expensive.

Nevertheless, both approaches assure physical and value independence. As a result, several advantages are achieved, namely

- the referential integrity constraint whereby a referenced object must always exist, is directly supported

- operations are simpler since joins are not needed to access properties of a referenced object. Instead path expressions can be used,

- object sharing, cyclic objects and object versioning can be modelled more easily

Data abstraction, whereby an object is known by its external operations (i.e. the interface) rather than by its structure, allows object implementation to be changed as long as the interface is kept the same. Implementation details are *encapsulated*, transparent from the user or other objects. Such details are kept in a *class*. A class holds

the structural (i.e. instance variables) and behavioural (i.e. methods) features of a set of similar objects, called the *extension* of the class.

A distinction has to be made here between classes such as *integer* (i.e. *scalar types*) or *date* (i.e. *composite-valued types*) and other classes such as *person*. Whereas the extension of the former are considered irrelevant, the set of instances belonging at a given time to certain classes such as *person* is specially important in DBs where iteration, quantification and value-based retrieval on such sets is the hallmark of DB manipulation. Moreover, scalar and composite-valued types do not have OID, and equality is value based. In contrast, objects such as persons, have uniqueness based on the OID and it is possible for two *persons* to have the same attributes but being different objects.

In DBs the ability to model behavioural characteristics of the UoD, represents a remarkable gain with respect to previous approaches. In traditional DBs a clear separation is drawn between data and programs. Only the data is managed centrally. Ordinary manipulation has to be repeated in each program. Including behaviour allows these manipulations to be enclosed within the DB.

In OODBMSs, **hierarchies** can be formed by specialsing classes into subclasses. Subclasses only hold specific behaviour, *inheriting* the rest from their superclasses.

Besides the external user of the object, subclasses introduce a new *client* for the class: a subclass can have direct access to the instance variables not only it defines but also those instance variables defined by any of its superclasses. As pointed out by [Snyder 86], this severely compromise encapsulation since instance variables cannot longer be changed without affecting possible subclasses. To avoid jeopardising encapsulation, subclasses as any other object, should refer to classes only throught the interface. In this way, instance variables can be changed without compromising subclass implementation.

Snyder proposes three categories of *class clients*, leading to three different kinds of interfaces or method visibilities [1]. The question is: who can use a given method?

- any object within the system. Then, the visibility will be *global*,
- only subclasses of the class defining the method. Then, the visibility will be *family*,
- only the class where the method is defined. Then, the visibility will be *local*

[1] ADAM terminology has been used to name the visibilities.

Besides inheriting methods directly, subclasses can **override methods** defined in their superclasses. However, to ensure that instances of the subclass can be used wherever an instance of its superclass can appear, such overriding has to conform to certain rules. This leads to the view of classes as types. A type T1 is said to be a subtype of T2 if every instance of T1 is also an instance of T2 [Cardelli 85]. This implies the so-called *principle of substitutability* whereby an instance of T1 can be used whenever an operation expects an instance of T2. Hence, overriding can be done as long as this principle of substitutability is not violated.

The *signature* of a method (or function) is the types of the input and output arguments, normally represented as:

$$method : I_1 \times I_2 \times ... \times I_n \longrightarrow O_1 \times O_2 \times ... \times O_n$$

where I_1 normally represents the type of the object receiver, and I_i represent the ith input argument and O_i stands for the i-ary output argument. The question is when a method with signature

$$T1 \longrightarrow T2$$

can be overridden by a method f' with signature

$$T3 \longrightarrow T4$$

without violating the principle of substitutability.

Some systems use the *covariant rule* that requires the input and output arguments of the subclass method to be subtypes of the input and output arguments of the method to be overridden in the superclass [Khoshifian 90], i.e.

T3 should be a subtype of T1
T4 should be a subtype of T2

In figure 3.1 the covariant rule is shown using Venn diagrams. But this rule has some pitfalls. For example, suppose a *person* can have *borrow* as a method. *Borrow* has a *book* as an input argument. The subclass *historian* specialises *person* by constraining the argument of *borrow* to be a *rare_book*, a subclass of *book*. Such overriding conforms to the covariant rule. The borrowing of a *book* by a *person* is achieved by sending the message:

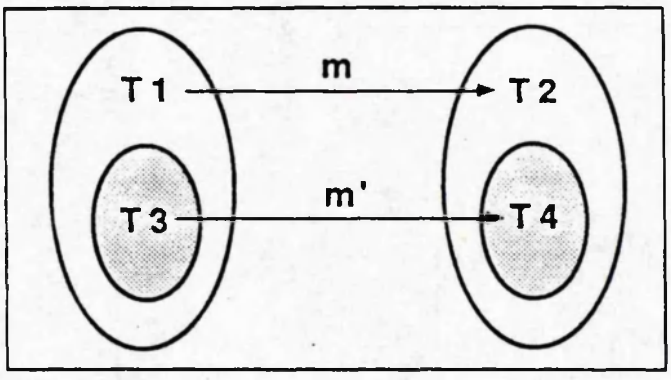


Figure 3.1: The covariant rule.

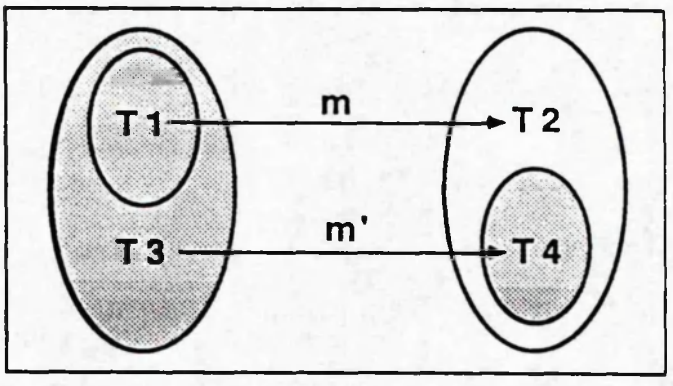


Figure 3.2: The contravariance rule.

borrow(b) => p.

where *b* and *p* are members of the *book* and *person* class respectively. No type error occurs at compile time. However, at execution time *p* can be bound to a *historian* and then, the specialised method would be executed. If it happens that *b* is not bound to a *rare_book* an incompatibility error will be caused at run time.

To ensure strong type checking and avoid run time errors, the *contravariance rule* is used instead. This rule states that the *input* arguments of the more specialised method should be *supertypes* of the input arguments of the least specialised method [Khoshifian 90], i.e:

T3 should be a supertype of T1

The contravariance rule is illustrated by Venn diagrams in figure 3.2. This rule guarantees strong type checking and works correctly for complete overriding (i.e. when the method in the subclass does not call the overridden method of the superclass). However, if a specialisation rather than a complete overriding is done some problems can occur.

That is, if the method of the subclass *m'* calls the overridden method of the superclass *m*, type errors can emerge at run time as a result of the input arguments of *m'* being more general that those of *m*. For instance, in the latter example, suppose that the *historian* subclass specialises the method *borrow* as having an *available_material* (a superclass of *book*) as input argument. Such overriding conforms with the contravariance rule. However, a type error arise when the *borrow* method of *person* is called if happens that the *available_material* is not a *book*.

The problem is that complete overriding and specialised overriding have different requirements. Complete overriding has proved very useful in OOPLs where reusing and flexible sharing are major concerns. In our opinion however, specilised overriding should be preferred in DBs where more strict overriding should be enforced.

This also leads to the distinction between inheritance and subtyping. Inheritance can be seen as a technique for sharing behaviour. In OOPLs a class can be defined by reusing methods of an existing class instead of beginning the definition from scratch. This process can be repeated leading to an **inheritance hierarchy**. In addition, since in a strict inheritance hierarchy all the methods of a class are available to its subclasses, it seems that whenever an instance of the class can be used, so can an instance of the subclass, and then, the inheritance hierarchy coincides with the **type hierarchy**. However, this is not always true, since subtyping is based not on sharing but on the externally observable behaviour, i.e. the interface [America 91]. Hence, if class B holds the interface of class A, B can behave as a subtype of A, without reusing A methods (e.g. it could reimplement then in a completely different way). As a result some systems, such as CommonObjects, distinguish between the type hierarchy and the inheritance hierarchy, making it possible for example "to specify that the class is not a subtype of a parent or that the class is a subtype of an unrelated class (not its parent). The first case arises when the behaviour of the objects is incompatible with the interface of parent objects. The second case arises when the class is supporting the external interface of another class without its implementation" [Snyder 86].

If a class can have more than one superclass the system is said to support **multiple inheritance**. In this context, ambiguity can arise if the same name is used in different superclasses to refer to the same method or instance variable. An example of such a problem is shown in figure 3.3. The class *person* defines the method *title* that writes

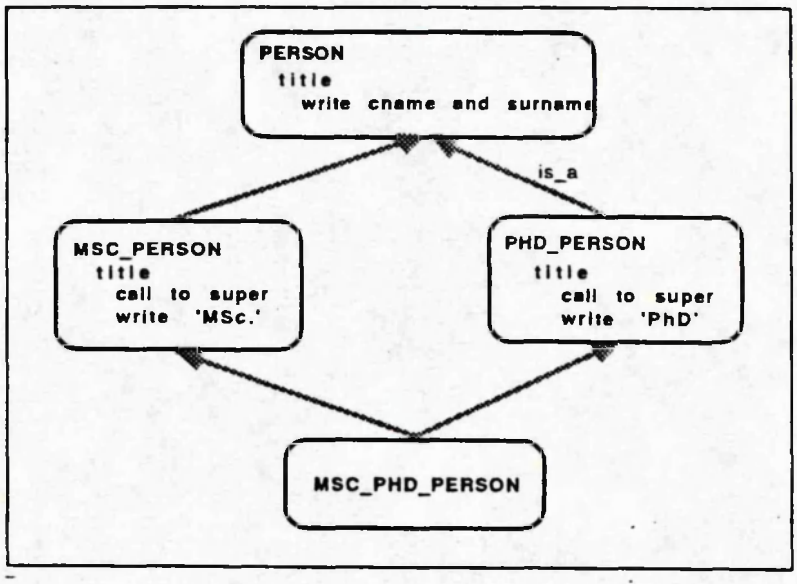


Figure 3.3: A multiple inheritance hierarchy.

the *name* and *surname* of a given *person*. *MSc_person* and *PhD_person* are subclasses of *person* where the method *title* is specialised by adding *MSc* and *PhD* at the end of the *surname*. The obvious way of doing this is by referring to the *title* method at the level of *person* and subsequently adding either *MSc* or *PhD*. However, an ambiguity occurs at the level of the class *MSc_PhD_person* subclass since it inherits two methods with the same name. Two approaches have been followed to cope with such situation [Snyder 87]:

- *the graph-oriented approach.* Methods are inherited directly and an error is signalled if any conflict occurs. In the above example, such error would arise if the message *title* is sent to an *MSc_PhD_person*. To resolve the ambiguity, a method *title* can be defined in the *MSc_PhD_person* class by calling both the *title* method of *MSc_person* and the *title* method of *PhD_person*. Nevertheless, this still causes some problems since the method *title* defined at the level of *person* is executed twice.

- *the linear approach* which flattens the acyclic graph. For example, the sequence: *MSc_PhD_person*, *MSc_person*, *PhD_person*, *person*, is the linear representation of the above example. When a method is sent, the *first* method found following the sequence is executed. Although now the method *title* of *person* is executed only once, it prevents the *title* method of *PhD_person* from being used.

Thus none of these approaches is completely satisfactory.

3.3 ADAM: an object-oriented database in Prolog

ADAM has been mainly influenced by research in OOPLs rather than by semantic data modelling, and it has been supported both by a form of grid file known as the BANG file and by a triple store based upon NDB [Paton 89a]. ADAM has been used as the implementation vehicle to support the ideas proposed in this work. In the following, the features outlined in the previous section for OODBMS are illustrated for ADAM.

• **Object identity.** In ADAM instance identifiers are provided by the systems which adds the class identifier to the instance number within the class to form the identifier. For example, *23@person* is a valid object identifier, where the *@* operator separates the name of the class (i.e. *person*) from the instance number (i.e. *23*). Although object migration is compromised, this approach allows an efficient message sending mechanism. As far as class identifiers are concerned, they are provided by the user.

• **Class and instance definition.** A new object is created in ADAM by sending the message *new* to the required class. For example, the creation of an instance of *person* can be achieved in the following way:

```
new([OID,
     <list of attribute values>
]) => person.
```

'=>' denotes message sending. In this case, the message is *new* and it is sent to the object *person*. The first argument of *new* is a Prolog variable which is instantiated with the system-provided object identifier of the instance just created.

An outstanding feature of ADAM is that classes are themselves instances of more abstract objects callled *metaclasses*. A class then, is created by sending the message *new* to a given *metaclass*. For instance, to create the class *person*, the following message is sent:

```
new([person,
     <list of attribute values>,
     <list of property definitions of person>,
     <list of method definitions of person>
]) => class.
```

The message *new* is sent to the object *class*, a metaclass supplied by the system [2]. Notice that class identifiers are provided by the user and that besides a list of attributes

[2] For further details about metaclasses, see chapter 4.

(e.g. the person who created the class), class description also includes the definition of the behavioural (i.e. methods) and attributive (i.e. properties) features of the class instances.

• **Method definition.** The following can be an example of a method definition in ADAM:

```
method((kind_of_salary(global,single,[integer],string,[Salary,Kind])
  :-  message_recipient(ThePerson),
      get_status(Status) => ThePerson,
      member(Status-Low-Medium,[clerical-50-70,managerial-68-100]),
      (Salary < Low
           -> Kind = low
      ; Salary < Medium
           -> Kind = medium
      ; Kind = high)
))
```

Method definition is done by specifying:

- the name. It is the method selector (e.g. *kind_of_salary*)
- the visibility. It describes from where this method can be called. The range of values are:
  - *global*, no restrictions on who can invoke this method
  - *family*, the method can be invoked only within the own class or subclasses of this class
  - *local*, the method can be invoked only within the class where it is defined
- the cardinality. It can be *single* or *set* depending on whether a method is likely to resucceed by Prolog backtracking, and then returning several values
- list of input arguments (e.g. *[integer]*)
- the output argument (e.g. *string*). Only one output argument is allowed
- list of variables representing the input and output arguments -in this order- (e.g. *[Salary,Kind]*)
- the body of the method implemented in Prolog

The method *kind_of_salary* shown above can be invoked from outwith the class (since it is global), returns only one value (since it is single), and has two arguments: *Salary* an integer input argument and *Kind* a string output argument.

Methods can be invoked by specifying the corresponding parameters and the object receiver. As an example, consider the *person* with object identifier *3@person*, having a *salary* of *86*. The previous method can be called to obtain the kind of salary this person has:

```
| ?- kind_of_salary([86],Type) => 3@person.
Type = high
```

- **Property definition.** An example of an attribute definition in ADAM can be as follows:

```
slot(slot_tuple(age,single,optional,integer))
```

where the following facets or property features have to be specified:

- the name of the property (e.g. *age*)
- the cardinality of the property. It can be either *single* or *set*
- the status. It can be either *total,mandatory* or *optional.Total* properties cannot be empty. They must be given a value when the object is created and any attempt to delete its value results in the deletion of the whole object. The motivation for total properties is that "any object with a total slot which contains a reference, is dependent upon the object referred to by the total slot" [Paton 89a]. The dependency is established at the class level since *all* objects in the class *must* specify their total properties. *Mandatory* properties "need not be given a value when the object is created, but once a value has been assigned to the slot, any attempt to delete the value results in deletion of the object to which the slot is attached" [Paton 89a]. Here the dependency is at the instance level since it is when a given instance has the slot filled that it becomes dependent. Finally, with *optional* properties no dependency with any other object is established. The user is free to fill or delete the value of these slots at any time. Total and mandatory slots can be seen as a way of achieving aggregation hierarchies by extending the semantics of

the slot values instead of using an explicit *part_of* hierarchy. Only single slots can be total or mandatory

- the type of the slot. It can be any scalar type, class name or composite value type

In addition, properties have also an internal facet known as *updateability* to guarantee the integrity of system-maintained properties such as *is_a* and *instance_of*. Such system properties have an updateability of *system* whereas user-provided properties have updateability of *local*.

• **Encapsulation.** When a property is created, methods to retrieve (*get_*), to add (*put_*), to delete (*delete_*) and to update (*update_*) the property value are generated automatically so that attributes are *always* handled by these methods. For example, the methods *get_age, put_age, delete_age* and *update_age* are created from the above definition of *age*. Properties that play the role of instance variables are not inherited by subclasses. Only methods are passed downwards so that the structure of properties can be altered without compromising subclasses. In this way, encapsulation is greatly increased [Snyder 86].

In addition, ADAM provides value-based retrieval through the methods with prefix *get_by_* which are also generated at property creation time. For example, the following call

```
get_by_age([29],Person) => person.
```

retrieves all persons whose *age* is *29*. Such methods "support inverses on all relationships between classes, and secondary indexes on every scalar slot" (i.e. property) [Paton 89a].

Nevertheless, a set of property operators are provided to allow the user to access directly the property so that the system-provided methods can be overriden by the user. This set is shown in figure 3.4. This can be useful to enforce integrity constraints. As an example, consider the *age* property to be restricted to take a value between 0 and 120. The system-provided method can be overridden to include this constraint in the following way:

```
replace_method([
     (put_age(global,single,[integer],[],[Age]) :-
               Age > 0,
               Age < 120,
               age <-- Age)
```

SYNTAX	ACTION
slotname <-- []	Remove all values from the named slot
slotname <-- old/new	Replace value old with new in named slot
slotname <-- value	Add the given value to the named slot
value <= slotname	Unify value with the values in the named slot

Figure 3.4: Property operators in ADAM.

```
]) => person.
```

When the *age* is inserted for a given *person*, this method is executed: the input argument *Age* is checked to conform with the constraint and if so, it is inserted as the valued of the property *age*. Otherwise, the method fails [3].

• **Hierarchies.** In ADAM classes can be specialised in several subclasses where the inheritance hierarchy and the subtype hierarchy coincide. Objects can be an instance of only one class (held in the *instance_of* property). However, classes can have more than one superclass (held in the *is_a* property). Thus, ADAM supports *multiple inheritance.* Ambiguity is prevented by avoiding *is_a* lattices in which it is unclear which is the method to be invoked in response to a message. That is, when a subclass or new method is defined, the system checks whether any ambiguity arises and if so, it causes an error. As pointed out by [Paton 89a], this approach is felt to be safer than imposing an order on the superclasses of a class or using heuristics such as 'choose the nearest'.

In many cases, *method specialisation* rather than a complete overriding is desired, i.e. we are interested in invoking the more general method from the more specialised one. Two mechanism are available in ADAM to achieve this result:

- *Sending to super:* "A message can be sent to the atom *super*. If a method in a class C sends a message to *super*, then the search for a method to respond to the message starts with the superclass(es) of C. If C has more than one superclass, they are searched in the order specified in the definition, depth-first up to joins"

[3] However, embedding constraints in method definitions severely compromise method overriding as discussed in chapter 5.

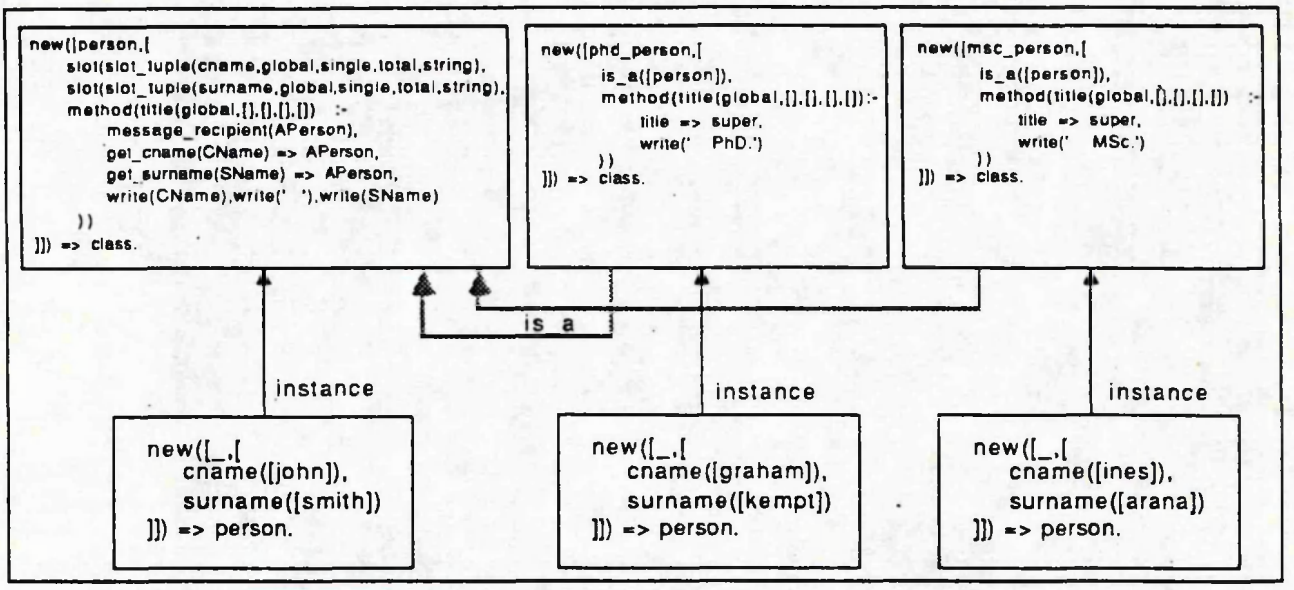


Figure 3.5: Method specialisation: an example.

[Paton 89a]. To illustrate this approach, let us suppose the message *title* is sent to instances of each of the classes shown in figure 3.5. The results are the following:

```
| ?- title => 0@person.
john smith

| ?- title => 1@phd_person.
graham kempt PhD.

| ?- title => 2@msc_person.
ines arana MSc.
```

- Selecting a super: "A message can be sent to a construct of the form *self@<class>* where *<class>* is a superclass of the class of object which received the message. The method at the given class is selected for execution. If there is no such method, an error is signalled and the message-send fails" [Paton 89a].

Method overriding is allowed in ADAM as long as the arity, cardinality and visibility of the new method are the same as in the method to be overridden and, input and output argument types conform to the contravariance rule. In figure 3.6, the type hierarchy is shown. The type *plog* is the most general type and any Prolog data structure is considered as a *plog*. The motivation for such type is that it provides a means of avoiding the type system when defining methods such as *new* which must take arbitrary Prolog clauses as

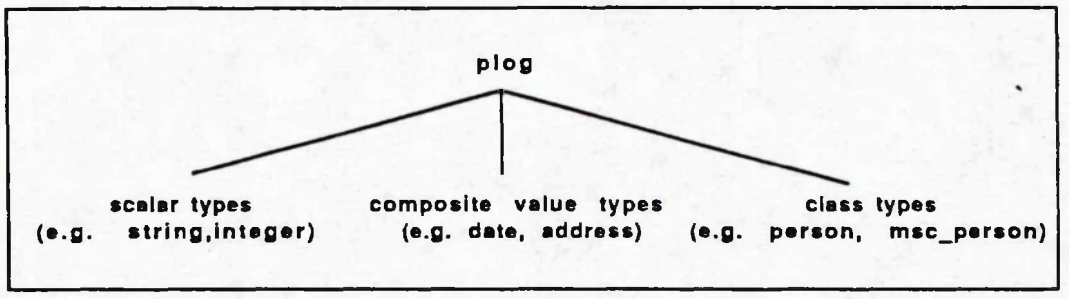


Figure 3.6: Type hierarchy in ADAM.

arguments for use as methods [Paton 89a].

The idea of *composite value types* is represented in ADAM by tuples. A tuple has a name and several named attributes of any type. The only restriction is that tuple attributes cannot be empty. As an example, consider the following definition of the *slot_description_tuple* and *method_desc_tuple* in ADAM:

```
new_tuple(meta_class,
   slot_desc_tuple(
      name:string,
      class:string,
      visibility:string,
      cardinality:string,
      status:string,
      type:string
))
```

```
new_tuple(meta_class,
   method_desc_tuple(
      object:string,
      name:string,
      visibility:string,
      cardinality:string,
      argtypes:plog,
      restypes:string
))
```

The first argument of *new_tuple* indicates where the definition of the tuple is to be stored.

Mechanisms for temporal modelling and versioning support are missing in ADAM, although a naive implementation of a versioning mechanism has been built based on rules [Diaz 91a].

3.4 Conclusions

In this chapter, the concepts of object identity, data abstraction and hierarchy mechanisms have been reviewed within a database framework. Such enhancements lead to the so-called object-oriented databases. Despite the fact that several prototypes and even commercial systems have been developed, a formal definition of object-oriented

databases is still missing. Hence, the ADAM system has been presented to illustrate some of these concepts. Moreover, this database has been used as the implementation vehicle to materialise some of the ideas proposed in this work.

Chapter 4

Making metadata explicit

4.1 Introduction

Unlike more traditional databases which are mainly concerned with *the extension* of the UoD, knowledge base systems are characterised by an increase of the knowledge kept in the database, i.e. *the intensional* side of the UoD. To access this knowledge, in the same way as any other data in the system, the objects in the data dictionary should be treated as regular objects. As pointed out in [Freundlich 90] "in user-oriented environments - and they are proliferating- users need modelling capabilities traditionally mediated by systems analysts and database managers. They need more than this -more than just tools for doing what was traditionally database management, and more than the ability to specify a data dictionary. They need to treat the objects in the data dictionary as regular data, as *objects of discourse*". In fact, one of the main differences between DBMSs and KBMSs is that whereas the former have a clear distinction between *ground* knowledge and *generic* knowledge, for KBMS such a distinction disappears [Mylopoulos 86].

In an object oriented paradigm, knowledge is represented through *classes* that gather together the common features of a set of objects called the class *instances*. In this context, making knowledge accessible means the ability to query, classify, generalise and relate classes just like any other object in the system. A means to accomplish this aim is through *metaclasses*. A metaclass is used to define the structure and behaviour of class objects in the same way that a class is used to define the structure and behaviour of instance objects. Metaclasses not only permit classes to be stored and accessed using

the facilities of the data model, but also make it possible to refine the default behaviour for class creation using specialisation and inheritance. Three main advantages can be drawn from this approach:

- *Accessibility.* Metadata is available like any other data in the system
- *Extensibility.* The object oriented paradigm encourages reuse and modularity through the subclass mechanism. When a new class (e.g. *student*) has to be introduced in the system, the designer thinks about the differences between, and similarities with existing classes (e.g. *person*), pointing out what is really new and what can be reused. The same mechanism used for building the DB can now be used to extend the system. New constructs can be introduced, specialising the ones already provided by the system. From this point of view, the system can be seen as a core of useful facilities which the user can tailor to suit specific applications [Paton 90].
- *Uniformity.* The same tools are used for querying, relating and specialising data and metadata. No new mechanism is required. In this way, the distinction between data and metadata is removed and it is just a question of the level of abstraction at which one is working.

Object oriented systems differ in the extent to which they support metaclasses as first-class objects. In many systems such as C++ [Stroustrup 86], O_2 [Deux 90] and EIFFEL [Meyer 88], classes are not objects and thus, there are no metaclasses. SMALLTALK-76 the experimental version previous to SMALLTALK-80 [Goldberg 83], was the first to introduce metaclasses. Later systems such as LOOPS [Bobrow 81], ObjVlisp [Cointe 87] and ADAM [Paton 89b] have supported this concept, but with major differences. This chapter provides the background material needed to understand how the constructs presented in the following chapters have been introduced in ADAM.

The remainder of this chapter is structured as follows. In section 2, how metaclasses have been implemented in object oriented programming languages is addressed. Section 3 introduces how metaclasses are handled in ADAM. Achieving extensibility and accessibility through metaclasses is illustrated in sections 4 and 5 respectively. Some drawbacks are briefly outlined in section 6. Finally, conclusions are presented.

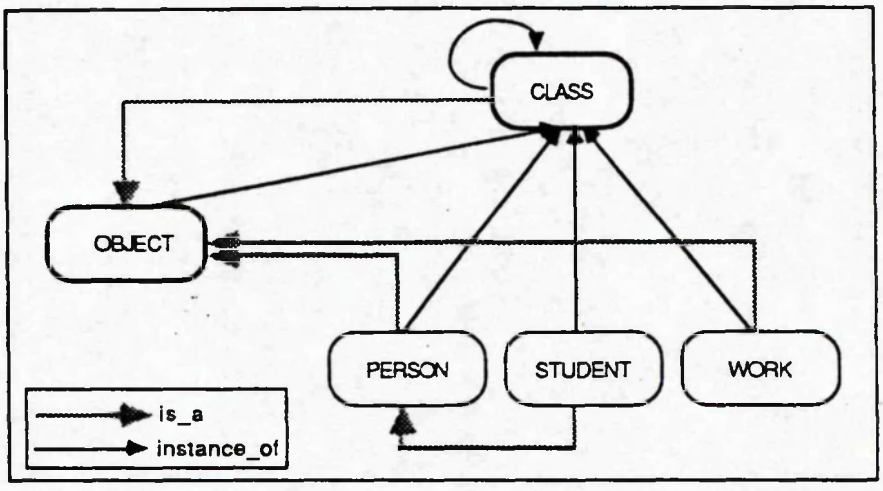


Figure 4.1: Inheritance and instantiation in SMALLTALK-76.

4.2 Metaclasses in Object Oriented Programming Languages

Achieving uniformity can be seen as the main motivation for introducing metaclasses into OOPLs. Systems which do not support metaclasses have to cope with two different kinds of objects, namely:

- Classes. They are instance generators but are not objects as such, since they cannot receive any messages (except *new*, to create instances) and they are not instances of any class.
- Instances. They can receive messages except *new*. Instances cannot create other instances.

To enhance uniformity, metaclasses have been introduced. A metaclass is a class whose instances are themselves classes. Now a class is a proper object in the sense that it is an instance of another object (the metaclass) and thus, it can receive message just like any other object in the system.

The following subsections present an overview of metaclasses in different OOPLs.

4.2.1 SMALLTALK

SMALLTALK was the first language to support metaclasses. In the following, the approach taken by SMALLTALK-76 is first presented, and once some drawbacks have been pointed out, the definitive mechanism provided by SMALLTALK-80 is shown.

In SMALLTALK-76, two orthogonal hierarchies are distinguished concerning the two aspects of an object: its behaviour as an instance and its behaviour as a class. In figure 4.1 these hierarchies are shown for SMALLTALK-76.

- The *instantiation hierarchy* based on how an object is created and which links each object with its object generator. The root is the metaclass *CLASS*. *CLASS* collects all the common information about classes such as how to add a method to the method directory or how to instantiate (i.e. the method *new*). For uniformity's sake, and to avoid an infinite regression, *CLASS* is an instance of itself. Now, the *STUDENT* class is created by sending the message *new* to *CLASS* whereas the object *JOHN* is created by sending the message *new* to *STUDENT*. Hence, *CLASS*, *STUDENT* and *JOHN* are all proper objects, i.e. are all instances of some object.

- The *inheritance hierarchy* where objects are seen as instances. The root of this hierarchy is the *OBJECT* class which collects the behaviour common to all instances within the system. This includes not only terminal instances but classes as well, since they are instances of *CLASS*. Among this common behaviour is the ability to cope with a message. So, now *STUDENT* can receive messages other than *new* just like any other object in the system. Uniformity is achieved.

Although several advantages can be drawn from this approach, its main drawback is that all classes behave in the same way, since they are all instances of the same class *CLASS*. Therefore, it is not possible to specialise *new* to suit special generation requirements, as this would jeopardise the extensibility of the system.

To overcome some of these disadvantages, SMALLTALK-80 introduces several metaclasses. However, metaclasses are hidden from the user. When a class is created, the system automatically defines its counterpart metaclass, where the metaclass name is obtained by adding 'class' to the class name (e.g. the class *person* has as a metaclass *person_class*). As shown in figure 4.2, metaclasses are arranged so that they mirror the class structure and each class is the only instance of its metaclass. At class creation time, the user can specify *class instances* and *class methods*, which play the role of instance variables and instance methods of the metaclass, respectively. As a result, it is possible to specialise the behaviour of classes by adding class methods. Despite this achievement, SMALLTALK-80 still has some problems. Its weakness comes principally from the sys-

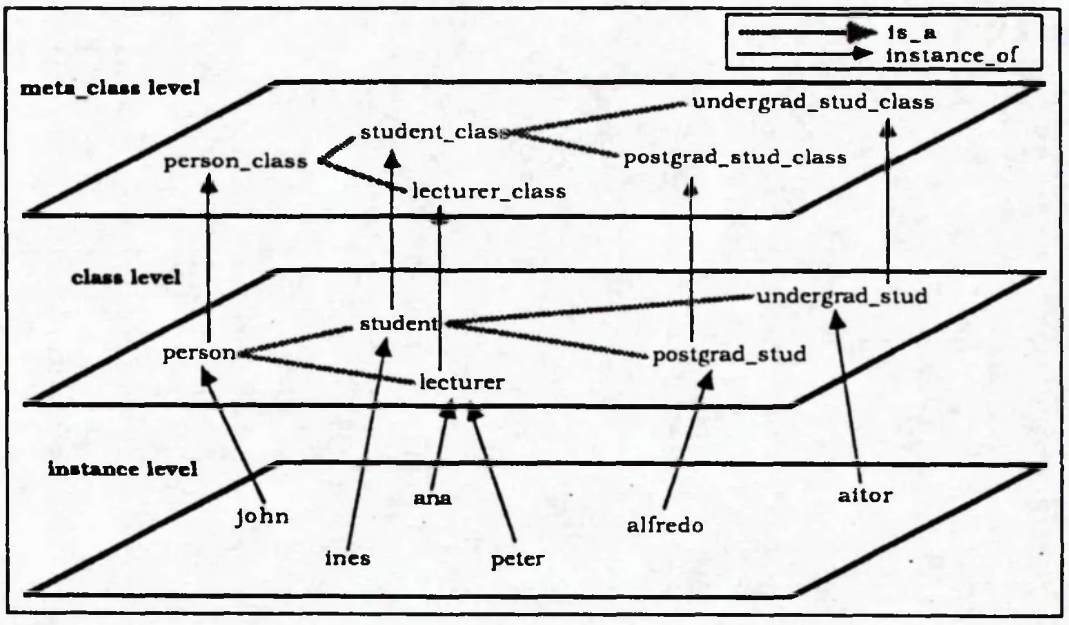


Figure 4.2: An example of the three levels in SMALLTALK-80.

tem maintained metaclass structure. Being too rigid, the structure does not offer the flexibility found at the class level where classes can have several instances and distinct subclasses.

4.2.2 LOOPS

Unlike SMALLTALK-76 where objects are categorised as instances and classes, LOOPS introduces a third category: metaclasses. Notice that SMALLTALK-76 does not have metaclasses as such. The concept of metaclass is in the users mind, but from the system point of view *STUDENT* and *CLASS* are handled in the same way.

In LOOPS, three objects stand for each category [Masini 89]:

- *META_CLASS* holds the default behaviour of metaclasses, particularly the method *new to create classes.* It is the metaclass of other metaclasses and as a result, its own metaclass.
- *CLASS* holds the default behaviour of classes, particularly the method *new to create terminal instances.* It is the default metaclass.
- *OBJECT* holds the instance behaviour. Since classes are themselves instances, *CLASS* is a subclass of *OBJECT.*

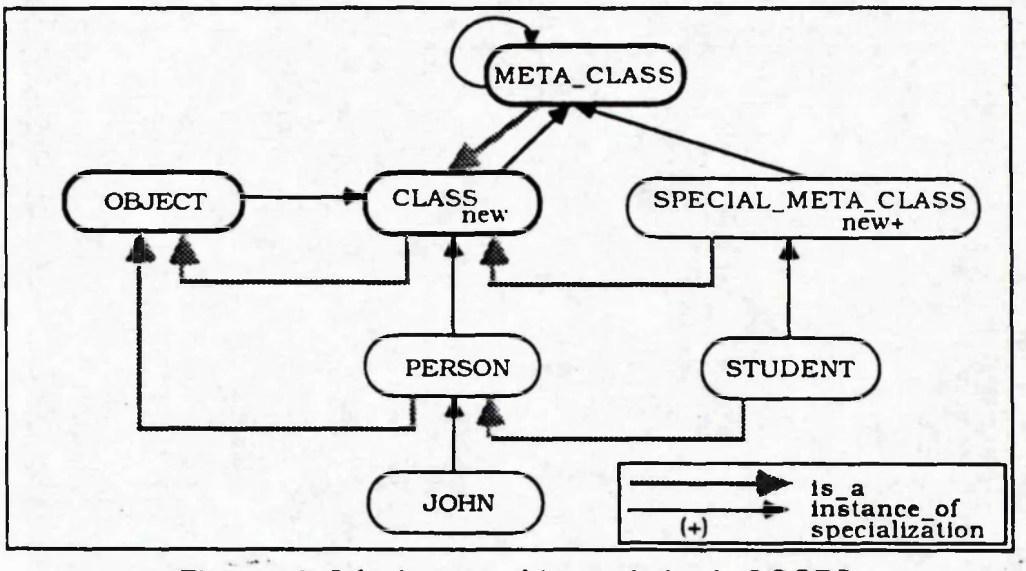


Figure 4.3: Inheritance and instantiation in LOOPS.

Figure 4.3 shows this hierarchy. Being a class, *PERSON* is created by sending the message *new* to *CLASS*. Likewise, since *JOHN* is a *PERSON*, it is created by sending the message *new* to *PERSON*. Moreover, now it is possible to specialise *new* (i.e. how an instance is created) by introducing a different metaclass. As an example, if some special behaviour must to occur when a *STUDENT* is created, a metaclass *SPECIAL_META_CLASS* can be created as a specialisation of *CLASS* and *STUDENT* can be made an instance of *SPECIAL_META_CLASS*. Hence, when an instance of *STUDENT* is created, besides the default behaviour held in *CLASS*, the additional behaviour of *SPECIAL_META_CLASS* can take places. In this way, special requirements can be supported. Therefore, the LOOPS mechanism allows specialisation of instance creation (unlike SMALLTALK-76) and explicit definition of metaclasses (unlike SMALLTALK-80).

However, the uniformity achieved in SMALLTALK-76 is lost since classes and instances no longer share the same structure as they have been created by different methods *new*: one stored at the level of *META_CLASS* and the other held by *CLASS*. Moreover the depth of the instantiation tree is limited to three levels.

4.2.3 ObjVlisp

ObjVlisp goes back to the ideas of SMALLTALK-76 but where CLASS can be specialised like any other class in the system. Unlike LOOPS, all objects are created by the same method *new*, and thus all have the same structure. Hence, the only difference among

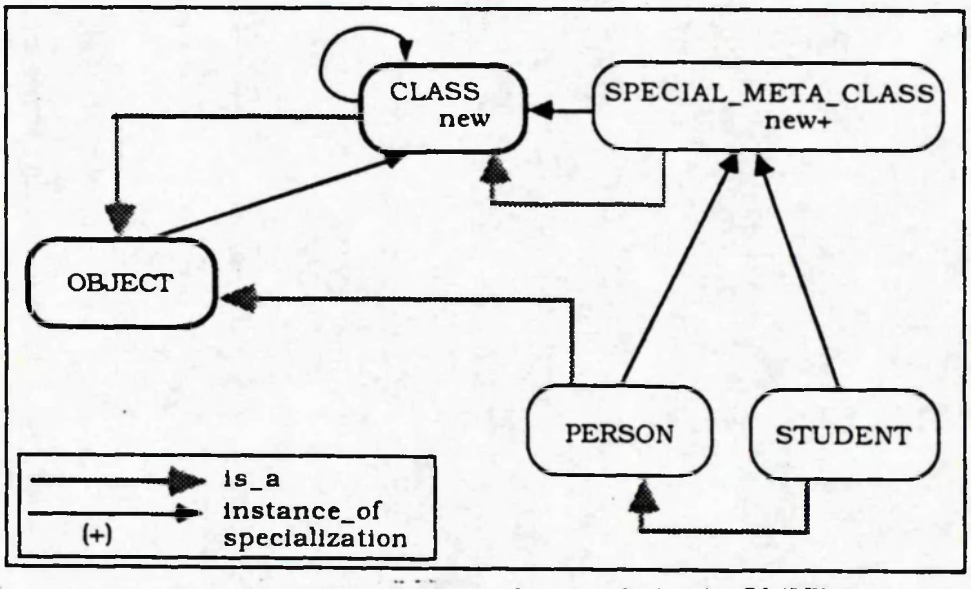


Figure 4.4: Inheritance and instantiation in ObjVlisp.

objects is their ability to create other objects, i.e. to answer the message *new*.

As in SMALLTALK-76, two hierarchies are defined:

- The instantiation hierarchy having CLASS as its root. It holds the message *new*. As any other class in the system, CLASS can be refined to account for special requirements. In figure 4.4 the user-defined metaclass SPECIAL_META_CLASS specialises the method *new* kept in CLASS to provide some special behaviour when PERSON instances are created.

- The inheritance hierarchy having OBJECT as its root. OBJECT holds the common behaviour of all objects in the system. Thus, any class in the system is a subclass of OBJECT.

In this way, classes and metaclasses have been integrated, both are handled in the same way. Since the distinction is removed, it is up to the user to be aware of the level at which he is working. As far as the system is concerned, ground objects and generic objects have an equal treatment. As a result, the flexibility and extensibility is preserved while keeping the system's simplicity.

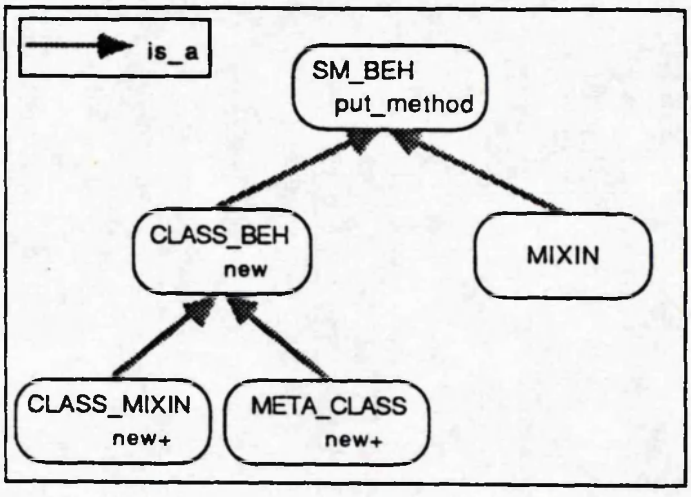


Figure 4.5: Instantiation hierarchy in ADAM.

4.3 Metaclasses in ADAM

In this section, a detailed description of the metaclass mechanism in ADAM is given. The rationales and the different objects building the core ADAM system are presented. A careful reading is recommended for those interested in how the extension of ADAM is accomplished in the following chapters. As far as metaclasses are concerned, ADAM follows a middle way, nearer to ObjVlisp than to LOOPS. As in previous systems two hierarchies are distinguished:

- The inheritance hierarchy where objects are seen as instances. The root of this hierarchy is the object OBJECT and it collects the common behaviour of all instances within the system. In the current implementation (version 2.1) this is restricted to the method *display* that shows the structural and behavioural features of a given object.

- The instantiation hierarchy based on object creation (i.e. the method *new*). In figure 4.5 such a hierarchy can be seen. Three kinds of objects are distinguished: *terminal instances, classes* and *mixins.*

  Terminal instances. They cannot create other instances. The message *new* cannot be sent to this kind of object.

  Classes. Classes can be characterised as being object generators, i.e. that can understand the message *new.* A distinction has to be made between classes of terminal instances -instance classes- and classes of other classes -metaclasses. Al-

though both share a common behaviour to generate an object, they specialise this behaviour to consider special features such as:

- terminal instances have an object identifier generated by the system whereas classes have an object identifier provided by the user,
- terminal instances do not have slot or method descriptions, unlike classes where such descriptions can be part of the class definition,
- unlike terminal instances, classes can be arranged in hierarchies. Hence, the superclass can be part of the definition of a class.

These distinctions can be illustrated by creating the class STUDENT and an instance of this class (e.g. JOHN), according to the ADAM syntax:

```
:- new([student,[
    is_a([person]),
    created_by([josemi]),
    slot(slot_tuple(cname,global,single,optional,string)),
    slot(slot_tuple(surname,global,single,optional,string)),
    slot(slot_tuple(age,global,single,optional,integer))
  ]]) => class.

:- new([OID,[
    cname([john]),
    age([25])
  ]]) => student.
```

The definition of both STUDENT and JOHN involves giving a value to the attributes defined in their respective classes (e.g. *cname* and *age* for JOHN and *is_a* and *created_by* for STUDENT). However, unlike JOHN, the definition of the class STUDENT includes the description of the slots *cname, surname* and *age*, and could have also involved some method definitions. Moreover, the object identifier of JOHN is automatically generated by the system and unified with the Prolog variable OID, whereas the object identifier of the class STUDENT is given by the user through the literal *student*.

The behaviour shared by the creation of terminal instances and classes is then factored out in the object CLASS_BEH, whereas the objects CLASS_MIXIN and META_CLASS specialise such behaviour for instance classes and metaclasses respectively.

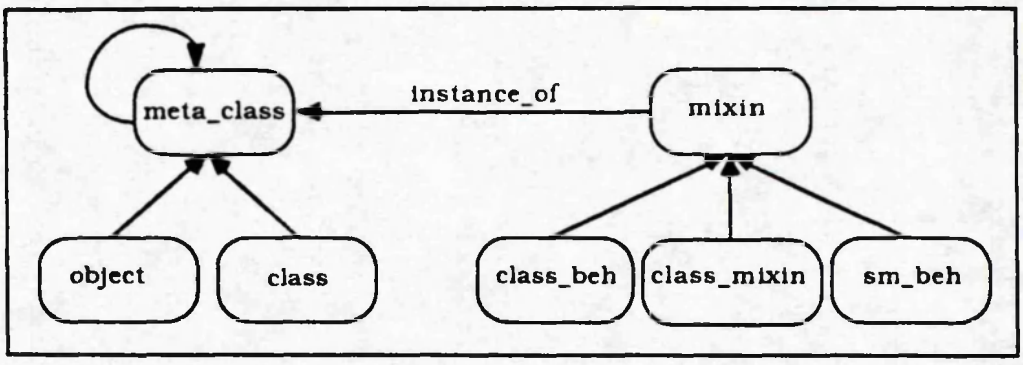


Figure 4.6: The 'who's who' hierarchy in ADAM.

Mixins. A mixin has been defined as "a class designed to augment the description of its subclasses in a multiple inheritance lattice" [Stefik 86]. They can be seen as a kind of library from which classes can inherit behaviour. Their main motivation is thus, the enhancement of sharing and they come mainly from the programming arena where reusability is a major concern. However they cannot have instances of their own.

Mixins have been found useful in programming languages and they are introduced in ADAM through the object MIXIN. Although they cannot have instances, mixins share with classes the possibility of defining behaviour (i.e. the method *put_method*). To factor out those features common to MIXIN and CLASS_BEH, the object SM_BEH has been created.

In the previous paragraphs, the inheritance and instantiation hierarchy have been introduced. Since the system itself has been constructed in an OO fashion, the next question to be addressed is which kind of objects are the objects which describe the system? For example, is CLASS_BEH a terminal instance, a mixin or a metaclass?

Certain of the objects in the instantiation hierarchy have as a *raison d'être* to factor out common behaviour of their subclasses and it is meaningless to have instances of most of these objects. So, they can be considered as a kind of library with methods to be used or specialised. The objects CLASS_BEH, CLASS_MIXIN and SM_BEH are then instances of MIXIN. Although the object CLASS_MIXIN could have been considered as a metaclass, it was preferred to hold the special behaviour of classes in a mixin so that it can be reused more easely. The metaclass CLASS was instead introduced which inherits from CLASS_MIXIN. In [Paton 90] an implementation is shown where ADAM

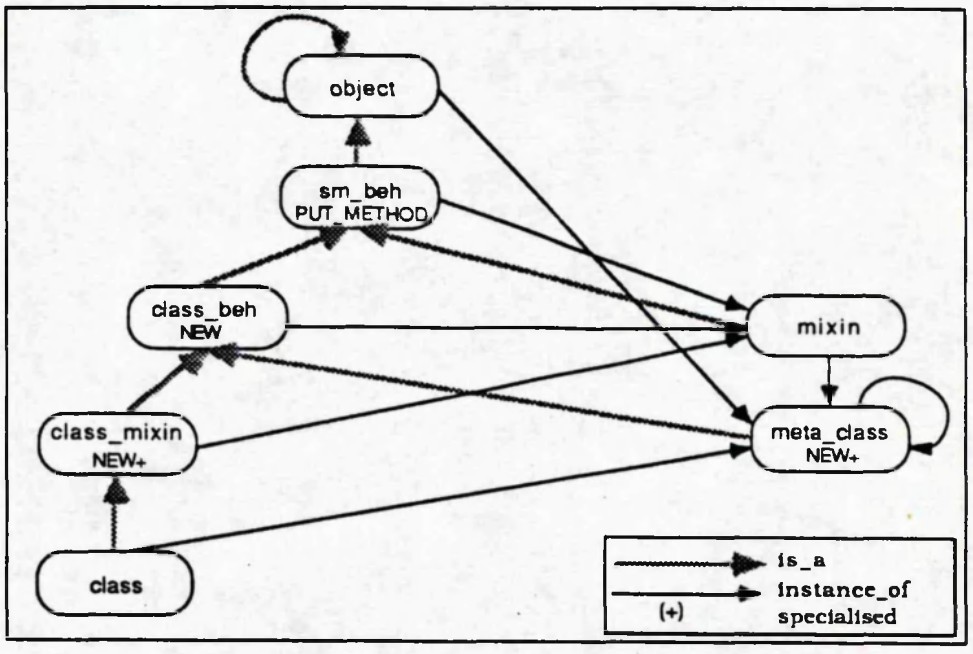


Figure 4.7: The ADAM system.

is extended with keyed classes illustrating the advantage of this approach.

The objects MIXIN, OBJECT and CLASS can have instances which in turn are classes. So, they are metaclasses and therefore, instances of META_CLASS. Finally, the object META_CLASS is supported as an instance of itself to avoid infinite regression. Figure 4.6 shows the *'who's who'* hierarchy in ADAM.

To conclude this section, in figure 4.7 all the objects defining the ADAM system are shown as well as their links. On top of this system, a set of new constructs have been added. How this has been accomplished using an OO approach is one of the main contributions of this work and it is presented in the following chapters.

4.4 Achieving extensibility using metaclasses

Extensibility can be defined as the ability of a system to be enlarged with new functionalities without changing the core system. This property becomes fundamental in the next generation of DB systems due to the difficulty of foreseeing all the requirements of a given application [Oxborrow 91]. The hypothesis is that it is better for a data model to provide a modest number of basic constructs which are easily adapted and extended than to support a large number of constructs which attempt to anticipate the needs of

individual users.

In OO systems, extensibility can be achieved if the system itself is described using an OO approach. The OO paradigm encourages reuse and modularity through the subclass mechanism. When a new class is introduced into the system, the designer thinks about the differences between, and similarities with existing classes, pointing out what is really new and what can be reused. If the system is described in an OO way, the same mechanisms used for building the DB can now be used to extend the system itself. So if a new kind of object such as a relationship object, is introduced, instead of beginning from scratch, we can reuse what relationship objects have in common with the already provided *normal* objects. In this way, the advantages of the OO paradigm are brought to the realm of the system itself.

In ADAM extensibility can be obtained by:

- specialising how terminal instances are created by creating subclasses of the object CLASS.
- specialising how classes are created. This can be accomplished by specialising the object META_CLASS,

As an example, consider the extension of ADAM with a new category of classes: *optimisation classes.* Besides the normal behaviour already provided by ADAM, optimisation classes add as a further requirement a knowledge of how many instances of each optimisation class there are in the system. Such counters can be used by some optimisation strategies within the DBMS [1]. In figure 4.8 how this can be achieved in ADAM is shown. This extension requires:

- to specialise the instantiation hierarchy (i.e. the method *new*) used to create classes so that when an optimisation class is created, the optimisation counter is initialised with zero. This is achieved by defining the OPTIMIZATION_META_CLASS metaclass as a subclass of META_CLASS. This metaclass can be defined in ADAM as follows:

```
:- new([optimisation_meta_class,[
```

[1] Other counters such as number of retrievals, updates or deletions can be added in a similar way.

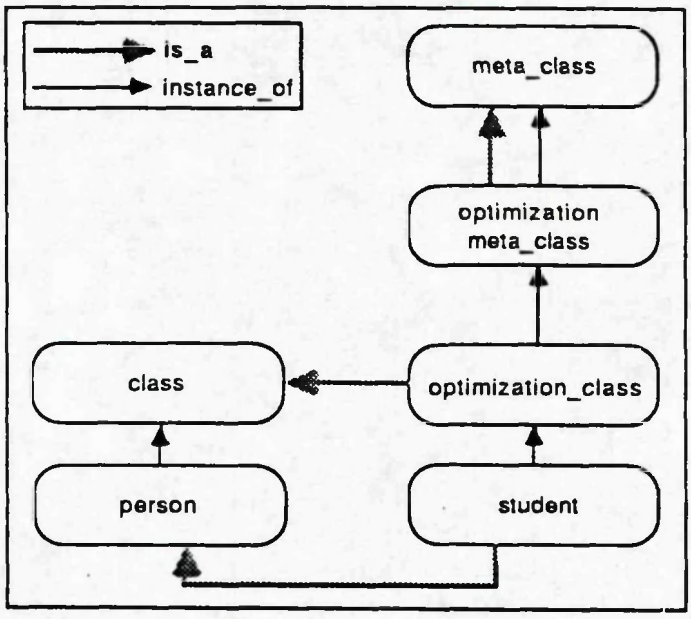


Figure 4.8: Extending the system with optimisation classes.

```
    is_a([meta_class]),
    method((new(global,[],[string,plog],[],[Class,Atts]) :-
       new([Class,Atts]) => super,
       put_optimisation_counter([0]) => Class
    ))
]]) => meta_class.
```

When a new *optimisation class* is created, the method *new* defined in OPTIMIZATION META CLASS will be used. This method initialises the *optimisation_counter* to zero once the class has been created using the standard behaviour. Although this initialisation could have been done explicitly by the user when the class is created, making such initialisation from the method *new* makes the *optimisation_counter* completely transparent to the user who is unaware of such a counter. Besides, new counters can be added or deleted as required for the DBMS, without changing user programs.

- to specialise how classes are created. Besides the standard creation procedure, *optimisation_classes* have an attribute, *optimisation counter*, where the number of instances is recorded so that when a new instance is created the appropriate counter is increased. This is achieved by defining the OPTIMIZATION_CLASS metaclass as a subclass of CLASS. OPTIMIZATION_CLASS can be defined in ADAM as follows:

```
:- new([optimisation_class,[
    is_a([class]),
    slot(slot_tuple(optimisation_counter,global,single,optional,integer)),
    method((new(global,[],[plog,plog],[],[Oid,Atts]) :-
       new([Oid,Atts]) => super,
       message_recipient(Class),
       get_optimisation_counter(Counter) => Class,
       NewCounter is Counter + 1,
       update_optimisation_counter([Counter,NewCounter]) => Class
    ))
]]) => optimisation_meta_class.
```

As with standard classes, a new optimisation instance can be created by sending the message *new* to the *appropriate optimisation class* that takes the method *new* from OPTIMIZATION_CLASS. It is worth noticing that optimisation instances are objects, i.e. they do not override the behaviour of *normal* objects but rather specialise this behaviour, adding the new features characteristic of *optimisation* instances. As shown above, the message new is first sent to *super*, i.e. to the class CLASS that holds the standard behaviour, and after, the special requirements are taken into account, in this case, to increase the corresponding counter.

In this way, the system has been extended with *optimisation classes* in an OO fashion. In [Paton 90], an extension to support objects with keys and persistent objects can be found.

4.5 Achieving accessibility using metaclasses

By introducing the concept of metaclasses, classes become *normal* objects in the sense that they can receive messages other than *new* and they can have their own attributes and methods [2]. Such attributes can be used to represent aggregate information (e.g. the average height of persons) or some system attributes (e.g. the *optimisation counter* shown before). These attributes and methods describe the structure and behaviour of the system itself and thus they play a similar role to *a data dictionary*. To retrieve such information, a set of methods are already provided by ADAM. Here are some examples:

- To retrieve all instances of a given class,

[2] In Smalltalk they are known as class variables and class methods.

```
| ?- get(Inst) => student.
Inst = 0@student;
Inst = 23@student;
......
```

- To obtain method and slot description of a given class,

```
| ?- get_slot_desc(Slot) => student.

Slot = slot_desc_tuple(course,student,local,single,total,integer);
Slot = slot_desc_tuple(home,student,local,single,optional,string);
......

| ?- get_method_desc(Method) => student.

Method = method_desc_tuple(student,get_course,global,single,[],integer);
Method = method_desc_tuple(student,put_course,global,single,[integer],[]);
......
```

- To browse the application hierarchy,

```
| ?- get_is_a(SuperClass) => student.

SuperClass = person

| ?- get_instance_of(MetaClass) => student.

MetaClass = optimisation_class
```

However, accessibility is not only limited to the metadata of the user applications but any structural and behavioural feature of the DBMS itself can be handled in a similar way. This stems from the fact that much of the system consists of metaclass objects itself. Hence, in the same way that previous examples have shown how to retrieve metainformation about *the application domain*, a similar set of queries can be used to retrieve information about *the system domain*. Some examples follows:

- To retrieve the instances of a metaclass,

```
| ?- get(Inst) => class.

Inst = person;
Inst = student;
```

```
| ?- get(Inst) => optimisation_class.

Inst = student;

| ?- get(Inst) => object.

Inst = mixin;
Inst = meta_class;
Inst = object;
Inst = class;
Inst = sm_beh;
Inst = class_beh;
Inst = class_mixin;
Inst = optimisation_meta_class;
Inst = optimisation_class;
Inst = person;
Inst = student;
Inst = 0@student;
......
```

- To browse the structure and behaviour of the system,

```
| ?- get_slot_desc(Slot) => optimisation_class.

Slot = slot_desc_tuple(optimisation_counter,optimisation_class,
                       global,single,optional,integer);
......

| ?- get_method_desc(Method) => optimisation_class.

Method = method_desc_tuple(optimisation_class,get_optimisation_counter,
                           global,single,[],integer);
Method = method_desc_tuple(sh_beh,put_method,global,single,
                           [plog],[]);
......
```

- To query the value of a meta-attribute, e.g. *optimisation counter*,

```
| ?- get_optimisation_counter(Counter) => student.

Counter = 59;
```

- To browse the hierarchy of the system,

```
| ?- get_instance_of(Class) => optimisation_class.
```

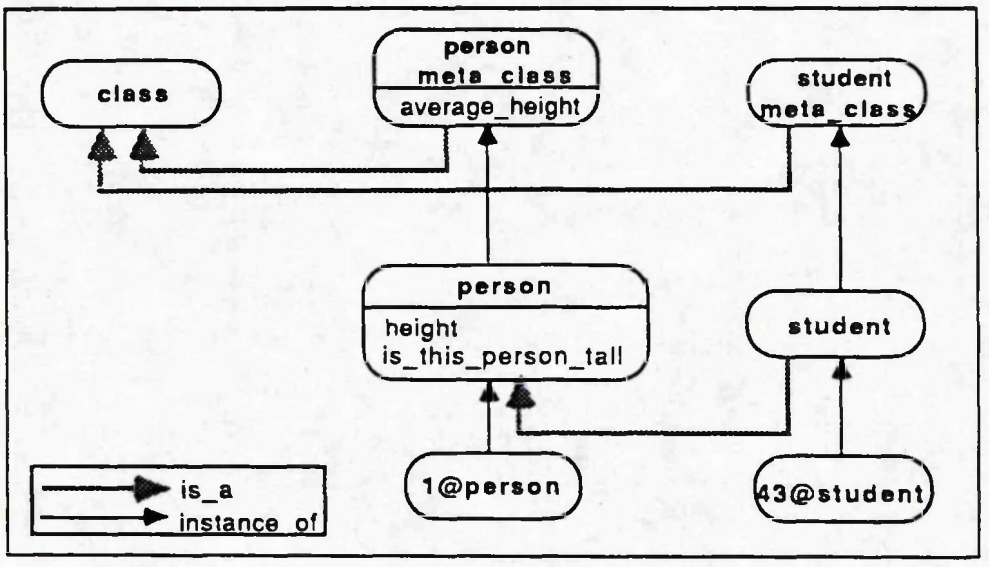


Figure 4.9: Metaclass compatibility problems: an example.

```
Class = optimisation_meta_class;

| ?- get_is_a(Class) => optimisation_class.

Class = class;
```

4.6 Some drawbacks with metaclasses

Although a favourable view of metaclasses has been presented, some problems arise when explicit specification of metaclasses is introduced. Some of these drawbacks have been pointed out in [Khoshifian 90]:

- Metaclasses make the system more difficult to understand.

- Metaclasses place a further burden on the user who has to cope with three levels of objects: instances, classes and metaclasses. The optimisation example has shown how two considerations have to be made: specialisation of the behaviour to create instances (specialisation of the class CLASS) and specialisation of the behaviour to create classes (specialisation of the class META_CLASS). The user has to keep in mind both hierarchies when the system is extended and this could be found difficult to grasp at the beginning.

- Metaclass compatibilities. Since instance methods can invoke class methods, some problems can arise when the class hierarchy does not parallel the corresponding metaclass hierarchy. As an example, consider the schema shown in figure 4.9. The class hierarchy does not mirror the metaclass hierarchy: STUDENT is a PERSON whereas STUDENT_META_CLASS is not a PERSON_META_CLASS. Now, let us suppose that *persons* have an attribute *height* and a method to check whether a given *person* is taller than the average *person*. This method can be defined in ADAM as follows:

```
method((is_this_person_tall(global,[],[],[],[]) :-
  message_recipient(APerson),
  get_height(Length) => APerson,
  get_instance_of(TheClass) => APerson,
  get_average_height(AverageHeigth) => TheClass,
  (Length > AverageHeigth
      -> write('This person is tall')
      ;  write('This person is not so tall')
  )
))
```

  This method is inherited by STUDENT. Since the definition of the *is_this_person_tall* method implies a metaclass request (the *average_height* attribute) and the metaclass of PERSON and STUDENT do not coincide then this method will fail for students [3]. So some set of metaclass compatibility rules is needed.

These disadvantages make the authors of [Khoshifian 90] conclude that

> ... although there seems to be some advantages in treating classes as objects, the theory, properties, and performance issues associated with the 'classes as instances of metaclasses' paradigm is still in its infancy. Metaclass systems are difficult to understand (and, alas, explain!). Neither the Smalltalk (hidden metaclass approach) nor the explicit metaclass approach seems to be satisfactory. The C++ 'classes as types' approach is a clear and mature technology that has been successfully implemented in numerous conventional as well as object-oriented languages.

Although this could be true for programming languages, our experience is that supporting metaclasses explicitly is a powerful mechanism for improving **database** extensibility, uniformity and accessibility. Although it takes some time to get used to metaclasses, the results are worthwhile.

[3] This problem does not arise in SMALLTALK where metaclasses are implicitly defined by the system and arranged in a structure that mimics the class hierarchy.

4.7 Conclusions

This chapter has shown how introducing metaclasses in OODB provides a uniform treatment of data and metadata. Since data and metadata are both represented as objects, only one mechanism is required for browsing, querying, relating and specialising objects. As a result, the distinction between data and metadata is removed and it becomes a question of the level of abstraction at which one is working. Indeed, extensibility and accessibility can both be seen as by-products of uniformity that were already provided for the user domain, and uniformity moves upwards to the system domain. Therefore, the only differences stem from the UoD that is being modelled. Application-based classes model for example, the entities of interest in a university or business environment, whereas metaclasses represent entities describing the behaviour and structure of the DBMS. It only depends on the level of abstraction, and no new mechanism is required.

Besides uniformity, two main advantages can be drawn from this approach: *accessibility* and *extensibility*. The former allows users to query, update and delete metadata dynamically as any other data in the system. Extensibility stems from using objects to define the DBMS which can be enlarged and specialised using the well-known subclass mechanism. In this way, whereas the class mechanism allows the knowledge domain to evolve, metaclasses allow the knowledge about the system itself (i.e. metaknowledge) to evolve to cope with future requirements (e.g. new constructs).

Moreover, any new facility introduced for objects is automatically applicable to both data and metadata (e.g. display facilities, active rules, etc). Although metaclasses are not free from some drawbacks and further research is needed, we see uniformity as a major achievement whose advantages have been 'put to work' as shown in the following chapters. The use of metaclasses to extend the systems represents one of the principal contributions of this thesis.

Chapter 5

Making integrity constraints explicit

5.1 Introduction

In [Morgenstern 89], Morgenstern gives a threefold definition of constraints:

> "The *purpose* of constraints is to describe a desired relationship among one or several variables or data-objects.
>
> The *specification* of a constraint is intrinsic rather than extrinsic. That is, a constraint is typically defined implicitly -such as by a symbolic expression- rather than by enumeration of instances.
>
> The *operation* of a constraint limits the combination of values which can simultaneously satisfy the variables or objects participating in the constraint."

In DBs integrity constraints can be seen as restrictions which must hold between different pieces of information to keep the database *consistent.* Besides *structural constraints* i.e. those provided by the data model, there are other kinds of constraints that cannot be reflected in a structural way. These constraints can range from simple domain restrictions (e.g. the age of a teenager must be between 13 and 19) to more complex relationships between different pieces of information (e.g. the projects a lecturer has responsibility for are to be the same as the set of projects which his/her research_assistants work on). Such constraints, restricting the valid DB states, are known as *static constraints.* In addition, *dynamic constraints* limit the possible DB state transitions (e.g. salaries can only increase).

Usually the specification and checking of these constraints is left to the user. Constraints then, are hard coded into application programs or even worse being only in the users' heads. Hence, constraints cannot be *uniformly* enforced for all users. Including constraints within the DB results in major advantages, namely

- it reduces cost of software development since constraints are coded only once and shared by all users,
- it provides uniform constraint enforcement, improving the reliability and consistency of the DB by managing constraint centrally,
- it reduces maintainability. The DB administrator is the only person to be in charge of defining and updating constraints

Through integration of data and programs, the object oriented paradigm allows uniform enforcement by coding integrity constraints as part of methods *within the database.* However this is not a satisfactory solution. Several disadvantages can be outlined:

- Domain constraints are often used to specify definitional properties (e.g. the *age* constraint in the subclass *teenager*). Being implicit in method description, these constraints cannot be properly specialised to define subclasses. Moreover, since overriding is a common practice in object oriented systems, a subclass redefining the *put_age* method may no longer maintain constraints defined at a higher level in the hierarchy. Constraints and methods therefore, should be two separate mechanisms since the underlying philosophies are quite different, namely that constraints represent invariant states of the database but method definitions can sometimes be overridden.
- "A designer may not correctly consider the effects of the constraint from the point of view of all objects involved" [Urban 89]. Similarly, as is illustrated by Urban, let's take the lecturer/research_assistant example. A trigger can be associated with the insertion of an research_assistant to automatically enforce that his/her projects are some of the projects of his/her lecturer. However, the constraint must also be considered when the research_assistant's lecturer is changed, or when the lecturer's projects are changed. If the constraint is declaratively stated, it is the system's

duty to enforce the constraint correctly, releasing the user from this cumbersome task and allowing him to focus on the constraint specification.

- Being implicit, constraints cannot be used to enhance other system capabilities. Semantic query optimisation is a case in point [Chakravarthy 90]. For instance, if all *teenagers* in their fifties are requested, the *age* constraint can be used to answer without looking up the database. If this constraint is buried, e.g. in the *put_age* method, the optimiser cannot use it.

As a result, not only is 'the updateability' of the system diminished, but so are its ease of use, legibility and inferential power. To overcome these drawbacks, a constraint specification mechanism is proposed for ADAM using a constraint equation (CE) approach where constraints are attached to the attributes as an additional facet. This work is restricted to static constraints.

The remainder of this chapter is organised as follows. A review of related work is presented in section 2. In section 3, the CE language is described where a Horn clause representation for CEs is outlined in section 4. The problem of constraint inheritance and how it has been addressed is discussed in section 5. In section 6 how CEs have been introduced in ADAM is explained. Finally, conclusions are presented.

5.2 Related work

Constraints have been widely used in PLs, AI and DBs [Morgenstern 89, Shepherd 84]. In PLs and AI constraint propagation is seen as a paradigm where the solution is found *by restriction*. TRILOGY [TRILOGY 89] and CLP($\mathcal{R}$) (Constraint Logic Programming over the domain $\mathcal{R}$ of real arithmetic) [Lassez 87] are good examples of this paradigm in PLs. CLP($\mathcal{R}$) merges the power of constraint solving and logic programming. Constraints look like Prolog predicates where no distinction is made between input and output parameters, and more interesting where there is no need to specify values for all the variables. In this case, the result is not a specific answer but rather an implicit relation held between the variables in order to meet the constraint. In AI, the expert system MOLGEN [Stefik 81] used this paradigm to minimize search by exploiting domain knowledge in the form of constraints.

```
Object: EMPLOYEE
Type: CLASS
Generalisation: OBJECT
    DEPARTMENT[department]:
        CM/MAINTENANCE: employees
    MIN-WAGE[kilo$]: 30
        CM/NOTIFY: (wage)
    WAGE[kilo$]:
        CM/CONDITION: (AND (NUMBERP *V*)
            (>= *V* MIN-WAGE)
            (<= *V* MAX-WAGE))
        CM/BINDINGS: ((MIN-WAGE MIN-WAGE)
            (MAX-WAGE (PATH DEPARTMENT MAX-WAGE))
```

Figure 5.1: Constraint definition in CONMAN.

However, in other systems constraints are mainly used for preserving a given situation rather than for obtaining a solution. In AI, several environments have been built to support the description of constraints attached to some object-oriented representation. In general, in these systems, constraint definition involves the specification of the situation to be preserved, where variables of the constraint are object slots, and a set of actions to be executed as a result of the occurrence of such a situation.

In [Wald 89] an extension of the STROBE system is presented. The extension called CONMAN (CONstraint MANanger), enlarges slot descriptions with additional facets to support constraint maintenance. An example taken from [Wald 89] is shown in figure 5.1. The situation to be preserved is described through the following facets:

- CM/CONDITION. This is a LISP function where some predefined functions can be used. The variables of this function represent the frame slots among which the constraint is established. When an operation is performed, the constraints are checked. If the condition is satisfied, the operation returns the value of the slot. Otherwise, CONMAN adds a violation entry to the slot and the operation returns

FAIL [Wald 89].

- CM/BINDING. This is a list of pairs of the variables appearing in the LISP function of the condition and a path indicating from where in the knowledge base the value of these variables can be obtained.

The action part is represented through the facets:

- CM/ACTION. This contains a set of actions taken as a result of condition satisfaction. By default, the functions executed are CM/MAINTENANCE* and CM/NOTIFICATION*. The former executes the actions described in the MAINTENANCE facet. The latter sends a *check* message to all the slots participating in the constraint (i.e. the locations appearing in the CM/NOTIFY facet). Other functions can be specified by the user.

- CM/MAINTENANCE. This contains a LISP form to be evaluated once for each changed element in order to test the consistency of the knowledge base.

- CM/NOTIFY. This refers to the set of locations to be checked when the value of the slot is modified.

- CM/VIOLATION. This keeps track of the violations that have occurred on in the slot.

These facets are used to maintain constraints on a *single* location. Although conditions can be specified using values from slots in other objects, these values are not updated to satisfy the constraint. When a set of locations can be modified, a *constraint object* is instead used. Now, the constraint is no longer represented as facets but as a proper object.

The shortcomings of this system are the lack of a truly declarative language for specifying constraints and the provision of two mechanisms for constraint definition depending on whether there is more than one location liable for modification or not. Moreover, the problem of constraint inheritance is not addressed. Since constraints are represented as facets, and facet values can be inherited and *overridden*, the adequate enforcement of constraint can be compromised.

SOCLE (Structured Object Constraint Language) [Harris 86] is a hybrid system containing a structured object component (essentially FRL) and a constraint component (based on the Constraint Based Programming Language presented in [Steele 80]). In SOCLE constraints are attached to the CONSTRAINT slot of the most general concept. "When an individual structure object is created, procedural attachments are placed along the path to the variable referred to in the constraint. These procedural attachments are charged with installing and maintaining the constraint network when changes are made in the participating structure" [Harris 86]. In the paper only numerical constraints are shown and the problem of constraint inheritance is not addressed.

In OOPLs constraints are used to test the correctness and completeness of the abstract data type represented by the class. Two approaches have been followed to introduce constraints in OOPLs. As in frame-based systems, constraints can be attached to the class itself, either as a proper attribute or as a facet. However, the most popular approach to constraint enforcement is through methods, either coded into the method's body or as pre- and post-conditions of methods. Preconditions allow the testing of certain constraints on the arguments and the object state, before a given method is executed. Postconditions describe a given situation to be satisfied once the method has been executed. As an example, consider the *get* and *put* operations on the *stack* type. The pre- and post-conditions of these operations can be specified as follows:

```
{true}    put(n)  {S = So * [n]}
{S <> []} get(r)  {So = S * [r]}
```

where S is the sequence representing the current state of the stack and So in the post-condition, represents the value of S before method execution. Furthermore, '*' stands for the concatenation operator and $[]$ represents the empty sequence. As a further example, the *increase_salary* method could be specified as:

```
{I > 0}   increase_salary(I)   {S = So + I}
```

where S and So stand for the *salary* value after and before method execution. Eiffel [Meyer 88] is a language where this approach has been followed.

In DBs, constraints have always been a major concern to guarantee consistency with the application domain. In the relational data model several approaches have been proposed

to improve efficiency in constraint maintenance [Eswaran 75, Lafue 82, Bernstein 80]. In [Morgenstern 84], constraint equations (CEs) are presented as a declarative representation for inter-relational constraints. Constraints are represented as connection paths where different set operations can take place. Automatic constraint enforcement can be accomplished by compilation of CEs into executable routines. One of the main contributions of CEs is a declarative way of expressing compensating changes to be performed to enforce the constraint. So, the system can automatically re-establish the validity of the DB from some 'clues' given by the designer in the constraint definition.

OODBs frequently embed constraints into method code. A nice example of constraints in CAD applications can be found in [Buchmann 86]. In this system, constraints can be established either for classes or instances. Both are defined at the level of the class, but whereas the former affect the class itself and are not inherited, the latter express restrictions on the instances of the class. In addition, constraints can accept exceptions, i.e. constraints can be relaxed for special cases. To improve retrieval of constraints for a given object, a *constraint header* is kept and indexed on the object identifier. These headers are sequences of bits, each with a special meaning. When a new object is inserted into the DB, the constraint checker has to verify whether there exists a particular constraint that has to be inherited. If so, the system automatically creates a header for the instance which point to the constraint header of the class where the constraint is defined. In this way constraint inheritance is done only once, when the instance is created, and constraints do not have to be traced later during modification operations. The main shortcoming of this approach is that constraints introduced after instance creation are not considered.

In [Urban 89] the language ALICE is presented. ALICE is a user-oriented version of first order logic for static logic-based constraints in an object-oriented environment. In this approach, constraints are converted to an internal, logic-based representation rather than being directly translated into a procedure. Such a representation is afterwards used for *constraint analysis*, i.e. to understand constraint interaction and to identify alternative propagation actions [Urban 89]. ALICE constraints can be strongly or weakly translated as decided by the constraint designer.

In the following sections, CEs are used as a declarative constraint language for OODBs, where active rules are automatically generated by the system to preserve constraints.

These rules are obtained from the logic-based representation counterpart of CEs.

5.3 Constraint equations

Constraint specification can be seen as answering the following questions:

- **What is the subject matter of the constraint?** i.e. what is the relationship to be maintained? The answer can range from simple domain restrictions to more complex relationships. To specify these relationships a constraint equation (CE) approach has been chosen. For instance, consider a constraint similar to the one proposed in [Morgenstern 84] which enforces that the *projects* which a *lecturer* has a responsibility for, are to be the same as the set of *projects* his/her *research_assistants* work on. This can be expressed as the CE:

  ```
  projects OF lecturer :: projects OF research_assistants OF lecturer
  ```

  CEs provide a formalism that closely mimics the structure of the object oriented model. As the user has to navigate through the OODB following different links, CEs are expressed by means of chains of relationships called *paths*. This formalism is already familiar to the user who does not have to express constraints in a different paradigm such as first order logic. Hereafter, the term *body of the constraint* will be used to refer to this part. In this work, we have restricted the comparison operators to set equality and scalar comparisons since our aim is to test the feasibility of this approach in OODBs rather than to provide a full integrity constraint language. For a proposal of an extension of these operators see [Morgenstern 84]. A constraint grammar can be found in appendix A.

- **What is the scope of the constraint?** i.e. what are the objects affected by the constraint? In Morgenstern's original paper, the constraint scope is the class that is used as the *anchor*, i.e. the beginning of the path. However this is not always the case. For instance, let us suppose that the above constraint applies only to lecturers in the 'computing science' department. To define a new subclass, e.g. *computing_science_lecturers* just to attach this constraint seems artificial, and can lead to an explosion of subclasses that hide the real hierarchy. To solve this

problem, CEs have been extended with a *scope clause* that specifies the scope of the constraint. Now, the above constraint could be expressed as:

```
projects OF lecturer :: projects OF research_assistants OF lecturer
      (WHERE  department OF lecturer = 'computing-science')
```

- **What must be done to maintain the constraint?** When a violation is detected, several reactions can follow:
  - reject the operation that causes the violation
  - display a warning message and accept the operation
  - make further changes that re-establish a valid state of the database

Usually this behaviour is embedded into the system and it is not available for inspection. One of the main contributions of CEs is a declarative way of expressing compensating changes to be carried out to enforce the constraint. Thus, when a change made in one path causes a constraint violation, the system knows how to restore a valid state. In [Morgenstern 84] the operator '!' (*WOF* in our notation) is introduced to represent the *weak bond*. The weak bond is the attribute "more readily modified in response to an initial change to the other side of the CE". For instance, in the *lecturer/research_assistant* example, two changes could be made as a result of adding a new it project to a *lecturer*. Either the new *project* can be added to existing *research_assistants* working with this *lecturer*, or the *lecturer's research_assistants* can be extended with the new *research_assistants* already working in the new *project*. The option chosen can be expressed as follows:

```
projects WOF lecturer :: projects OF research_assistants WOF lecturer
```

meaning that *projects* stay with the *research_assistant* if there are any other changes. Thus if a *lecturer* adds a *project* then, he adds those *research_assistant(s)* who work on that *project*. Here, the user just specifies the CE and the system makes the appropriate changes to assure that the constraint is properly enforced. Similarly, a weak bond can also be specified in the left path so that violations resulting from changes in the right path are restored by adding the *projects* to the *lecturer*. If no weak bond is provided, changes will not be allowed.

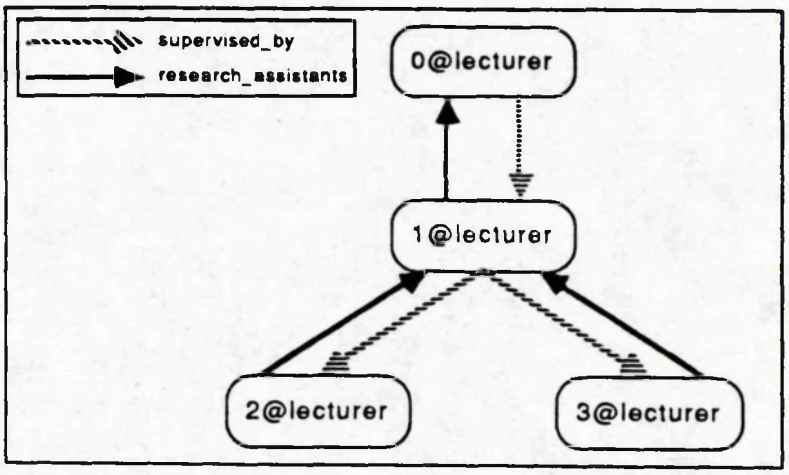


Figure 5.2: An example of constraint maintenance.

Finally, compensating actions can include modifications in other attributes. This can result in a chain of constraint maintenance actions taking place: update operations made to preserve a given constraint can lead to modify attributes which in turn can violate other constraints that would require compensating actions and so on.

The procedure to preserve these constraints can be quite complex. Consider the example in figure 5.2 where the attribute *supervised_by* is declared to be the inverse of *research_assistants*. Notice that a *lecturer* can have several *research_assistants* and at the same time be *supervised_by* any other *lecturer*. Consequently, if a new *project* (e.g. 8@work) is added to a *lecturer* (e.g. 1@lecturer), the constraint has to be preserved in both of the roles played by a *lecturer*: as a supervisor, the *project* has to be added to his/her *research_assistants* (e.g. 2@lecturer, 3@lecturer) ; as one who is supervised, the *project* has to be added to his/her *supervisors* (e.g. 0@lecturer). From this point of view, the operational semantics of CEs can be seen as rules that can be fired forward or/and backward, depending on the role played by the object in the relationship.

5.4 Horn rule representation for constraint equations

CEs have an equivalent first order logic representation. Class names can be seen as predicates of a single argument, whereas attributes can be translated into binary predicates. From this point of view, paths can have either a *weak translation* or a *strong translation*

[Urban 89]. In a weak translation, the attributes representing the links of the path, are *not* required properties. This is achieved by using universal quantifiers and implication connectives to translate the path. For example the following constraint:

```
age of student  >  0
```

ca be weakly translated as,

$$\forall x(student(x) \rightarrow \forall y(age(x,y) \rightarrow y > 0))$$

or, in its Horn rule form,

$$student(x) \& age(x, f(x)) \rightarrow f(x) > 0$$

The Horn rule reveals that the clause is trivially satisfied [1] if either x is not a *student* or x does not hold an *age*. Thus, *age* is not a required property for the constraint to be satisfied.

In contrast, attributes are seen as required properties if the path is strongly translated, i.e. it is essential that the values of these attributes be known if the constraint is to be satisfied. Thus, a strong translation is not tolerant of null values. To achieve this result, existential quantifiers and conjunction connectives are used. For instance, the previous constraints can be strongly translated as,

$$\forall x(student(x) \rightarrow \exists y(age(x,y) \& y > 0))$$

or its Horn rule form,

$$(student(x) \rightarrow age(x, f(x))) \quad \& \quad (student(x) \rightarrow f(x) > 0)$$

For a *student* to satisfy this constraint, it is not only required that his/her *age* be greater that zero but also that the *age* of the *student* is known. Otherwise, the first Horn rule is not satisfied and the whole constraint is not met.

When the constraint is established between two paths rather than between a path and a scalar value, a strong translation has to ensure that the relationships represented by each of the paths exist, i.e. attributes participating in any of the paths have to be specified. For instance, a strong translation of the following constraint,

[1] An implication is said to be trivially satisfied if it is evaluated to true as a result of the condition being evaluated to *false*.

```
working_address of employee :: address of company of employee
```

does not allow null values for the *working_address, company* and *address* attributes. This can be seen as a double implication: giving the *working_address* of an *employee*, the existence of the *company* of this *employee* and its *address* is mandatory, and vice versa, giving the *company* of an *employee* where the *company address* is specifed, it becomes compulsory to provide the *employee's working_address.* The above constraint is therefore, represented in first order logic as:

$\forall$e (employee(e) $\rightarrow$
(($\forall$w (working_address(e,w) $\rightarrow$
$\exists$c (company(e,c) & $\exists$a (address(c,a) & a=w))))
&
($\forall$c' (company(e,c') $\rightarrow$
$\forall$a' (address(c',a') $\rightarrow$
$\exists$w' (working_address(e,w') & a'=w' [2]))))
)

or, in its Horn rule form,

employee(e) & working_address(e,w) $\rightarrow$ company(e,f(e))
&
employee(e) & working_address(e,w) $\rightarrow$ address(f(e),g(e))
&
employee(e) & working_address(e,w) $\rightarrow$ w = g(e)
&
employee(e) & company(e,c) & address(c,a) $\rightarrow$ working_address(e,h(e))
&
employee(e) & company(e,c) & address(c,a) $\rightarrow$ a = h(e) [2]

As far as CEs are concerned, paths specifying the scope of the constraint are weakly translated whereas paths appearing in the constraint body are strongly translated. For example, the following CE:

```
patron OF student :: sponsor OF projects OF supervisor OF student
     (WHERE  department OF student = 'computing')
```

can be transformed into the following set of Horn rules, where variables with the same name refer to the same object:

[2]When the comparison is not an equality, the operator predicate appearing here is preceded by a negation.

student(S) & department(S,computing) & patron(S,P) → supervisor(S,f(S)) &
student(S) & department(S,computing) & patron(S,P) → projects(f(S),g(S)) &
student(S) & department(S,computing) & patron(S,P) → sponsor(g(S),P) &
student(S) & department(S,computing) &

supervisor(S,V) & projects(V,J) & sponsor(J,P) → patron(S,P)

Notice that attributes participating in the scope of the constraint appear only in the condition part of the Horn rules. Hence, *students* not having 'computing' as their *department*, trivially satisfy the constraint. In other words, the constraint is not applicable for objects out of the constraint's scope. For *students* within the scope, the constraint must not be trivially satisfied. So if *projects* and *supervisor* attributes are single valued, this constraint requires that, in order to have a *patron*, the *student* must have a *supervisor* and this *supervisor* must be involved in a *project*. And vice versa, if a *student* has a *supervisor* working in a sponsored *project* then he/she must have a *patron*.

Although, in other approaches the user can choose between a strong or weak translation [Urban 89], our system provides a fixed interpretation to minimize user interaction. Nevertheless, some degree of flexibility is achieved by seeing relationships as objects. As presented in the next chapter, constraints can be specified as part of the semantics of relationships. Such constraints are enforced for the objects *participating* in the constraints *as a result of the establishment of the relationship*. For example, as it is stated, the above constraint requires that a *student* can have a *patron* only if he/she has a *supervisor* working in a *project*. However different interpretations can be intended:

1.- If a student has a *supervisor* then he has a patron. In this case, the system requires that the supervisor has a project to support his/her students. The constraint would be part of the definition of *supervising* relationship. The constraint will be enforced for a student as long as he/she is supervised. Otherwise the constraint is not applicable.

2.- If a student is *working* in a project then he has a patron. Thus, the project has to be supervised. In this case, the constraint would be attached to the *working* relationship. Only working students have to satisfy this constraint.

3.- In the above cases, students can have their own patron without being supervised or working in a project (e.g. having a grant from the government). However if the intended meaning is that all students having patrons have to be supervised then the attribute *patron* should be an attribute of student *but as a result of being supervised.* In other words, the *patron* attribute should be part of the semantics of the *supervising* relationship.

Therefore, a finer granularity in constraint definition can be achieved by seeing relationships as proper objects. This is further discussed in the following chapter.

5.5 Inheritance of constraints

Unlike relational DBs, OODBs do not have a flat structure but they provide a subclass mechanism to enhance reusability and organisation of the application domain. Such a mechanism allows the structure and behaviour of the superclasses to be inherited by their subclasses. Integrity constraints should not be an exception. Nevertheless, in the literature this topic is hardly addressed and, in general, systems do not support constraint inheritance that has to be realised by the user himself. For instance, if constraints are embedded into methods, it is up to the programmer to decide whether a call to *super* is made when a method is overridden so that the constraints of the superclasses can still be checked. The problem is that complete method overriding could be the desired behaviour but this would also lead to constraints being put aside. Some systems overcome this problem by providing different parts within method definition that have a different overriding behaviour. For example, in Flavours [Weinreb 81], three parts are distinguished: the *before*, the *after* and the *main* part, each of which is optional. The *main* part overrides any inherited *main* part, whereas the *before* and *after* parts are all done in a nested order determined by the class precedence list [Stefik 81].

In other systems such as CONMAN, constraints are specified as facets. But here again, facet value inheritance and combination is left to the programmer. Besides putting an extra burden on the user, such practice jeopardizes constraint maintenance since new values specified lower down in the hierarchy can override constraints specified higher up. It is thus, very convenient to move constraint inheritance to the system realm.

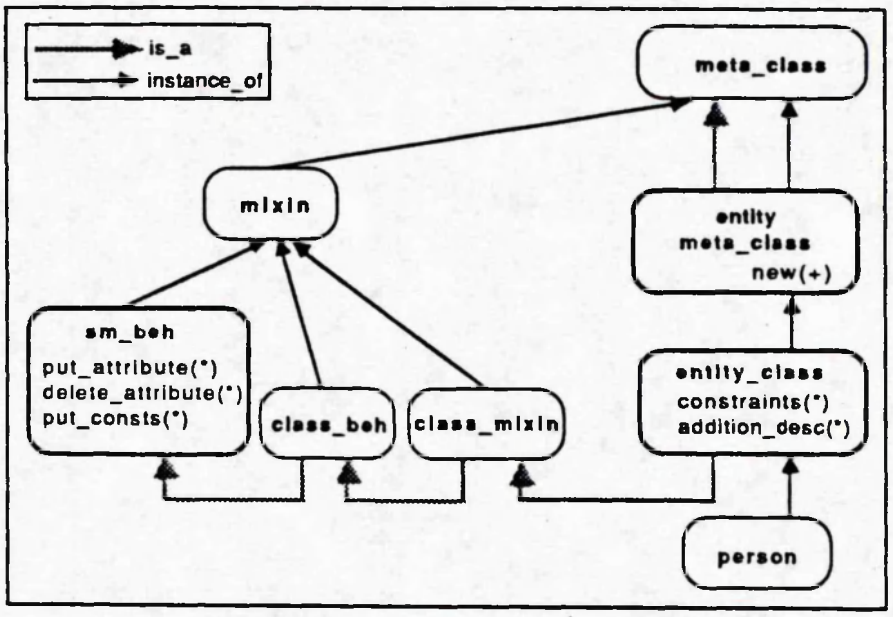


Figure 5.3: An ADAM extension to support constraints.

In this work, an approach based on event-condition-action rule have been followed. The system generates a set of rules from the declarative specification of the constraint. Such a rule set preserves the constraint when changes to the DB occur. It is the rule mechanism itself that guarantees that all applicable rules, even those defined higher in the hierarchy, are fired.

Rules are triggered in sequence from the more general to the more specific. This follows the heuristic that rules have to be fired in an adequate context, i.e. more specific rules wait till more general rules are fired to provide the right framework. For example in a hierarchy *person-academic-lecturer*, constraints attached to *person* are checked first, followed by those associated with the *academic* class and finally, those of *lecturers*. But the point to make here is that it is the system's responsibility to ensure constraint maintenance and constraint inheritance.. In this way, constraint management is moved out from the application domain to the concern of the system.

In the following section the way in which these ideas have been implemented in ADAM is presented. Constraint inheritance and maintenance is postponed until chapter 7 when rules are introduced.

5.6 Extending ADAM to support constraint definition

This section describes an extension to ADAM to support constraints. In figure 5.3 a diagram of the extended system is shown where the symbols '*' and '+' stand for new method or attribute definition, and specialisation of previous existing methods, respectively.

The extension mainly refers to the enlargement of property definitions with an additional facet to represent the list of constraints attached to the property. Hereafter, the term *attribute* is used to refer to properties with constraints.

In ADAM, attributive definition is kept in the meta-property *slot_desc* and it is accomplished through the *put_slot* method. Such method is defined in the *SM_BEH* mixin and it has a *slot_tuple* tuple as its input argument. Now, the attributive definition refers not only to the facets described in the *slot_tuple* but also to the constraints. To store this information the meta-property *addition_desc* has been defined which has the following *addition_desc_tuple* as its type:

```
:- new_tuple(meta_class,
     addition_desc_tuple(name:string,consts:plog,rule:plog)
).
```

Attributes (i.e. properties with constraints) can be defined in metaclasses, classes or mixins. So the behaviour to support attributes is included within the *SM_BEH* mixin. An attribute can be defined by invoking the *put_attribute* method whose definition is as follows,

```
method((put_attribute(global,[],[attribute_tuple],[],[AttTuple]) :-
   message_recipient(ClassName),
   AttTuple = attribute_tuple(AttName,Vis,Car,Status,Type,Consts),
   SlotTuple = slot_tuple(AttName,Vis,Car,Status,Type),
   put_slot([SlotTuple]) => ClassName,
   (Consts = [] -> true ; put_consts([Consts]) => ClassName)
))
```

The standard facets are collected in a *slot_tuple* tuple and then, the *put_slot* method is called to create the appropriate property. Constraints are then introduced using the *put_consts* method. This method generates the appropriate rules for enforcing the constraints, and updates the corresponding *addition_desc* property.

Once *put_attribute* is available any object within the system can have attributes by calling this method. However, the normal procedure is to define attributes at the time the

class is created rather than posteriorly. This implies the specialisation of the method *new* for class creation in order to cope with attribute definition. Hence, a specialisation of the instantiation hierarchy of ADAM is required (refer to chapter 4). The ENTITY_META_CLASS metaclass is introduced as a subclass of META_CLASS where the method *new* is specialised as follows [3],

```
new([entity_meta_class,[
  is_a([meta_class]),
  method((new(global,[],[string,plog],[],[ClassName,TheAtts]) :-
    delete(TheAtts,constraints([Consts]),Atts),
    findall(NoAtt,
            (member(NoAtt,Atts), \+ NoAtt = attribute(_)),
            NoAtts),
    new([ClassName, NoAtts]) => super,
    (member(attribute(AttTuple), Atts),
     put_attribute([AttTuple]) => ClassName,
     false ; true))),
    (Consts = [] -> true ; put_consts([Consts]) => ClassName)
]]) => meta_class.
```

First the class is created with the attribute definitions and the *constraints* attribute removed. Afterwards, attribute definitions are added by invoking the *put_attribute* method, and constraints corresponding to attributes defined higher in the hierarchy are inserted through the *put_consts* method (see below for an example).

As shown in figure 5.3, the metaclass ENTITY_CLASS is introduced as an instance of ENTITY_META_CLASS so that when instances of ENTITY_CLASS are created, the method *new* defined in ENTITY_META_CLASS is used. This allows ENTITY_CLASS instances to specify attributes at creation time. The definition of ENTITY_CLASS is as simple [4] as:

```
new([entity_class,[
    is_a([class_mixin]),
    slot(slot_tuple(is_a,system,set,optional,object)),
    slot(slot_tuple(slot_desc,system,set,optional,slot_desc_tuple)),
    slot(slot_tuple(addition_desc,global,set,optional,addition_desc_tuple)),
```

[3] This specialisation allows ENTITY_META_CLASS instances to specify attributes at creation time. For any other objects in the system, attributes can be defined only through the *put_attribute* method. Nevertheless, similar specialisations can be introduced for other objects if required.

[4] Although conceptually, the slots *is_a, slot_desc, addition_desc* and *constraints* can be part of the definition of any class regardless of whether they are instance classes, metaclasses or mixins, storage requirements force these static properties (i.e slots) to be attached to each metaclass rather than being defined in *SM_BEH*.

```
    slot(slot_tuple(constraints,global,set,optional,plog))
]]) => entity_meta_class.
```

As an example, the definition of an application-based class after the extension looks like:

```
new([lecturer,[
  attribute(attribute_tuple(age,global,single,optional,string,
    [age of lecturer > 20])),
  attribute(attribute_tuple(projects,global,set,optional,string,
    [projects wof lecturer :: projects of research_assistants wof lecturer]))
  ..................
]]) => entity_class.
```

Further constraints on defined attributes can be added when a subclass is created through the *constraints* construct. As an examples, if *professor* is a subclass of *lecturer* a constraint restricting his/her age to be below 85, can be introduced as follows

```
new([professor,[
  constraints([[age of professor < 85]]),
  ...........
]]) => entity_class
```

It is worth mentioning that these changes have been accomplished without modifying the core system thanks to the availability of metaclasses. Metaclasses allow the system to be extended based on the same subclass mechanism used in the user domain.

5.7 Conclusions

In this chapter an approach to support integrity constraints in ADAM has been presented. Constraints are explicitly specified using a constraint equation approach. The path-based notation of this approach has the advantage of being already familiar to the user that does not have to cope with a new formalism such as first order logic. Constraints are attached to the attributes as an additional facet and thus, they are no longer embedded into methods.

The main contributions of this work can be summarised as follows:

- Although CEs have already been proposed for relational DBs [Morgenstern 84], moving then to an object-oriented framework pose new challenges. Constraints

no longer refer to attributes in flat relations but to classes arranged in hierarchies. This arises the problem of constraint inheritance which has been tackled by moving constraints out of method definition.

- A rule-based mechanism is proposed for constraint maintenance. A set of rules is automatically generated by the system from the declarative specification of the constraint. Such a set is obtained based on the first order logic counterpart of CEs.

Chapter 6

Making user-defined relationships explicit

6.1 Introduction

SDMs have objects, relationships, dynamic properties and integrity constraints. Traditional data models attempt to overcome the shortage of powerful constructs by using integrity constraints. Some integrity constraints are part of the model itself. For instance in the relational model the unique key constraint, the referential integrity constraint, the domain constraint and the non null constraint are defined. These are called *structural constraints* and they are supported by the model. In fact, normalisation can be seen as a process designed to preserve the semantics of UoD, reflected by functional dependencies, through mechanisms supported by the DB such as keys. However these structural constraints are not sufficient to model all the complexity of the UoD. In this case, the semantics has to be embedded into user programs or reflected by means of a constraint language expressing the so-called *behavioural constraints.* From this point of view "a primary objective of many semantic models has been to provide a coherent family of constructs for representing in a structural manner the kinds of information that the relational model can represent only through constraints. Indeed, semantic modelling can be viewed as having shifted a substantial amount of schema information from the constraint side to the structural side" [Hull 87].

This 'shifting' has mainly concerned abstract relationships. These relationships stand

for abstractions already used both in AI and philosophy, such as generalisation, aggregation, classification and association. A clear semantics must be defined specifying how insertion, deletion and modification operations made at a higher abstraction level (e.g. *person*) can affect the object abstracted (e.g. *student*, *lecturer*, other subclasses) and vice versa. Nevertheless, most of these insertion, deletion and modification constraints have to be expressed in a rule-based or procedural way. As an example, in [Amy 89] an approach to derived data update is presented for the SDM data model. Different update rules are defined based on schema information. Some of these rules just reflect the semantics of abstract relationships. For instance the following rule is provided: "Let Tc be a subclass of Tp with derivation 'specified by the user'. Inserting an instance A to Tc will cause the same instance to be inserted to Tp if it is not there already. Deleting an instance B from Tp will cause deletion from Tc if B is also an instance of Tc. Insertion to Tp or deletion from Tc will not be propagated (by default)". This rule specifies the insertion and deletion semantics of the generalisation abstraction. Similar rules are provided for other subclass constructs in the SDM data model. Unfortunately all these rules are put together in a program so that the semantics is split and embedded throughout the program instead of being attached to the abstract relationship whose semantics is represented.

However, one of the main points of the *object oriented* approach is the gathering together of all the information concerning an object. This refers not only to the structural but to the behavioural features as well. Inclusion of the operational semantics is a step ahead in modelling the UoD. Since SDM's lack this facility, the behavioural semantics of an abstract relationship is distributed throughout the program.

Whereas abstract relationships, provided by SDMs, have somehow a semantic definition attached to them, no mechanism is provided to describe the semantics of user-defined relationships. As a result these semantics are still wired into the user programs instead of being in the schema definition of the DB. Hence, in the same way that SDM's provide a system-maintained semantics for abstract relationships allowing all users to have a clear notion of the consequences of DB manipulation, in this chapter we propose a system-maintained mechanism to specify the semantics of user-defined relationships. To cope with the variety and complexity of relationships of the UoD is seen as a requirement of the new environments to which OODBs are being applied [Oxborrow 91].

The DB designer should be able to specify the semantics of a relationship, e.g. what conclusions can be drawn as a result of relationship establishment. For instance, let *working_in* be a relationship between a *company* and a *person*. When a relation is established, can the person obtain his *working_address* attribute from the *address* attribute of the *company*? If the system does not specify anything, different users can have different understandings of what *working_in* a *company* means and therefore, different conclusions can be drawn. Hence, the semantics of insertion, deletion and modification operations have to be specified not only for abstract relationships but for user-defined relationships as well. OODBs make this possible by encapsulating behaviour as well as structure. Therefore, enhancing the semantics of user-defined relationships in OODBs, not only helps to increase the UoD semantics kept in the DB, but also to preserve the 'behavioural' integrity of the system.

The organisation of the remainder of this chapter is as follows. In section 2 a brief review of how relationships are represented in SDM's, OO systems and AI is presented. An enhanced description of user-defined relationships is discussed in section 3. In section 4 a description language for relationships in ADAM is introduced. In section 5 the subject of relationship specialisation is addressed as an interesting consequence of seeing relationships as objects. An extension of the ADAM system to support user-defined relationships is presented in section 6. Finally, conclusions are outlined.

6.2 Relationships in SDM's, OO systems and AI

User-defined relationships have been represented either as pointers (i.e attribute-values) or as aggregations (i.e list of pairs). An aggregation-based approach has been chosen for some SDM's (e.g. entity-relationship model). An attribute-based approach can be found both in SDMs (i.e. functional data model) and in OO models. This approach has several disadvantages, illustrated for OO programming in [Rumbaugh 87]. For instance consider the relationship *working_in_a_project* between a *lecturer* and a *work*. Here an attribute (an instance variable in OO programming) would be introduced in each participating class (e.g. *projects* and *partners*), together with some methods. Figure 6.1 shows this situation.

Some disadvantages can now be outlined:

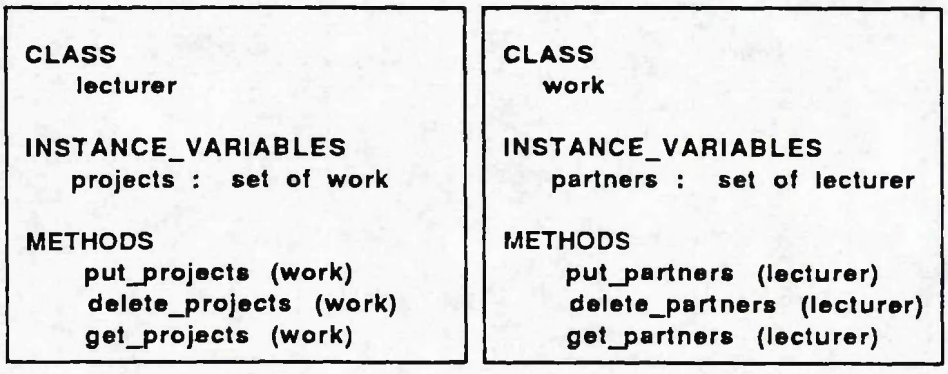


Figure 6.1: An attribute-based approach for *working_in_a_project.*

- the inverse link constraint is not expressed declaratively. In this example the instance variables *projects* and *partners* are constrained . So the method *put_projects* of the *lecturer* has to access the instance variable *partners* of the *work* in order to preserve the constraint. The same can be said for the *put_partners* method. Similar access should be provided for the delete methods.

- encapsulation can be lost. In order to maintain the inverse link constraint, methods of one object have to access the instance variable representing the inverse link in the other participant object.

- the relationship semantics are split between different objects.

- the conceptual and implementation levels cannot be kept distinct when relationships are represented.

- an instance variable must be reserved in each object instance for each relationship that an object of a given class can participate in. This tends to discourage the use of sparsely populated relationships. This is quite important in DBs. In the example, just a few *lecturers* can be involved in projects. Nevertheless, the instance variables *projects* must be reserved in each instance of the *lecturer* class. The alternative of defining a new subclass for each relationship in which *lecturers* could be involved is impractical, since a lot of subclasses would make the schema unmanageable.

- it is difficult to add a new relationship once the classes are already populated since a new local variable needs to be created.

- operations on the relation as a whole are not possible in a straightforward way.

- where are attributes of the relationship itself to be placed?

For OO programming a proposal is made in [Rumbaugh 87] to overcome the above disadvantages. A class *relation* is defined in which several methods are attached for scanning a relation, testing membership of a relation, adding an element to a relation, etc. Relation objects contain a description part and a variable-length value part. The description part contains the degree, the cardinality and a list of fields. The value part is a set of tuples of values from the respective object classes. Although this approach provides relationships as main constructs, it still considers relationship semantics in a traditional way.

In [Escamilla 90] the distinction between *vertical* (i.e. abstract relationships) and *horizontal relationships* (i.e. user-defined relationships) is also supported by an object-oriented knowledge base. Every attribute within the system is seen as an object. Relationships then, are a special kind of attribute that can have a richer associated semantics. such as the *the mathematical properties* of the relationship (reflexive, symmetrical and anti-symmetrical) and the nature of the link. The latter includes *the dependence* (similar to *the status* facet of ADAM) and *the diffusion* property whereby some attributes can be delegated to other objects through the relationship. In our approach a different semantics is proposed and only relationships as such are seen as objects.

Knowledge representation languages within the AI field, capture the UoD in a richer and more flexible manner. Relationships are often represented as pointers. For instance CRL (Carnegie Representation Language) [Carnegie 85] represents the fact that a *lecturer* frame is *working_in_a_project* by introducing a slot *projects* with the name of the *work* as its value. However the term *relation* in CRL is misleading. CRL has a functional view of the world. Attributes and therefore relations, are seen as functions, described in special frames called *slot-control schema* having as their name, the name of the slot. When a value is given to the function, i.e. a value is added to the slot, the description of the function is checked. This description can be the domain, the range, the cardinality or special restrictions enforced by *demons* in a procedural way. The most generic function description is kept in the frame SLOT. The rest of the slot-control schemata have to be is-a related with SLOT. As a specialisation of SLOT, the frame RELATION introduces several primitives to describe the inheritance semantics of functions playing the role of relationships. The slot-control schema of a slot representing a relation, has to be is_a related with RELATION. Figure 6.2 shows this hierarchy. The important point here is

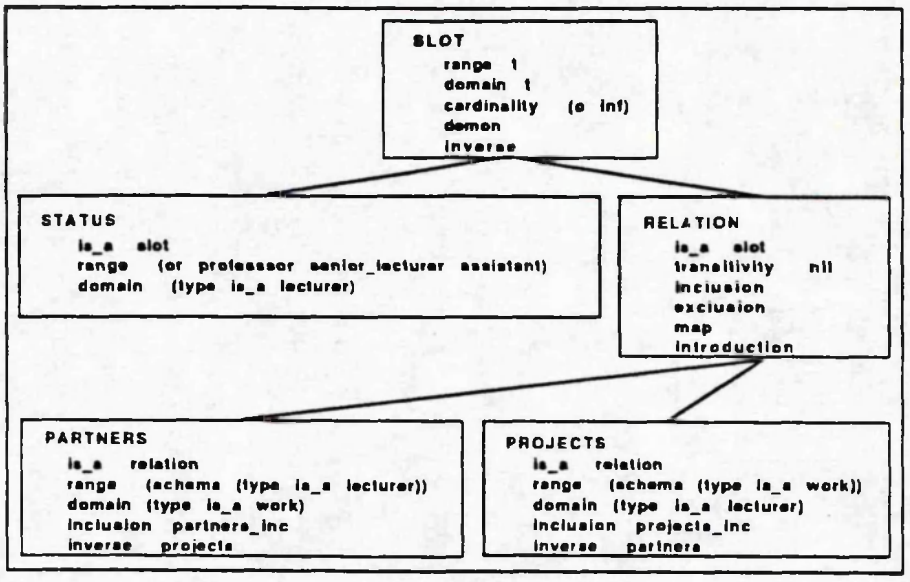


Figure 6.2: Function hierarchy in CRL.

the introduction of special *primitives* to describe the semantics of inheritance. We shall now attempt something similar for relationships in OODB.

6.3 Semantic-rich User-defined Relationships

For our research an aggregation approach has been chosen that allows relationships to be seen as objects. In this way, the entire semantics of the relationship can be kept in just one place, which gets round of some of the problems mentioned above and is more in keeping with the object-oriented philosophy.

Having decided that relationships are going to be represented as objects, the question arises of what can be said about a relationship. What are the primitives that describe its semantics? Traditionally, relationships have been characterised through the degree, the cardinality constraint, the participant objects and the attributes of the relationship itself. A new set of primitives is proposed to enhance this semantics, not only in its static (structural) aspects but also in its dynamic (behavioural) aspects. Static aspects represent permanent, invariant descriptions of the UoD. For instance, definition of attributes and constraints can be included in this category. On the other hand, dynamic aspects refer to how the state changes under certain conditions. The preservation of constraints, update rules for derived data and the definition of methods in general, can be considered dynamic aspects.

As an illustration consider the *marriage* relationship. For relationships the following is known:

1. the domain of the participants in the relationship and their role (e.g. a *marriage* is established between instances of class *person*, playing the roles of *the_husband* and *the_wife*)
2. attributes of the relationship itself (e.g. the *marriage* takes place on a *wedding_date*)
3. the cardinality of the relationship (e.g. one *husband* can have just one *wife* and vice versa)
4. restrictions to be satisfied by the participants in order to be related (e.g. both persons must be older than sixteen, one male and the other female)
5. how the relationship is affected by updating its participants and vice versa (e.g. if either of the participants die, the marriage ends. If the marriage is broken up, the participants are still alive!)
6. how the participants are affected by the establishment of the relationship. New asserted facts can be drawn such as new attributes, either constrained or unconstrained (e.g. if a person is married, the person has a *mother_in_law*, being the mother of the other participant), and new constraints (e.g. two *persons* participating in the same *marriage* should have the same *address*)

Traditionally, only the first four points have been considered.

Point 5 is related with the composite object problem. A relationship can be seen as a simple kind of composite object where dependences are defined at the class level. In ADAM this is represented by *total* slots, slots that cannot be empty and any attempt to delete the value from the slot results in deletion of the object itself [Paton 89a]. So if *the_husband* and *the_wife* are defined as *total* slots in the *marriage* relationship, any attempt to delete a *person* playing the role of *husband* or *wife*, will lead to the deletion of the *marriage* instance. Another alternative could be to disallow the deletion of a participant object unless the relationship in which it is involved is first dissolved.

Until now, structural and behavioural features of the UoD have been discussed. However, point 6 above describes a sort of knowledge that does not fit exactly under the labels

structural or behavioural. To refer to the knowledge about how to obtain these asserted facts, we prefer to use the term *inferential*. Introducing inferential knowledge, creates a new dimension in DB systems. We focus on how to use this inferential knowledge to characterise a user-defined relationship.

As a result of the establishment of a relation, new information can be drawn about the participant objects. This information can either refer to new attributes to be stored or new constraints to be enforced. For instance, when the relationship *marriage* is established between two *persons*, the new attribute *mother_in_law* needs to be gathered for the *persons* participating in the relationship. These are attributes that exist for as long as the relationship exists. A doubt can arise about when an attribute should be considered an attribute of the relationship itself or an attribute of some of the participant objects. If the meaning of an attribute is given by both participants, then the attribute is owned by the relationship. Otherwise it can be considered a new attribute of one of the participants. To avoid some of the problems outlined in the last section, it is better to collect these new attributes at the relationship level. Hence if the relationship is deleted, so are these attributes. However, these implementation details should be hidden from the user. So if we wanted to know who is *Kate's mother_in_law*, the corresponding message would be sent to the instance *Kate* instead of to the *marriage* instance. The user does not need to know anything about the semantics of *marriage*. We just know that *mother_in_law* can be an attribute of *Kate* and so, we ask *Kate*. From the user's point of view, *mother_in_law* should be an attribute of the *person* class in the same way that the attribute *name* is. Hence, a method to access this attribute of the *person* has to be provided transparently to the user. The only difference from a 'normal' attribute is that the user can only add or delete a new *mother_in_law* if a new *marriage* relationship is created or deleted. But this is just the behaviour required since these new attributes exist for only as long as the relationship exists.

Besides new attributes, new constraints may need to be enforced. For example, when two *persons* get married, the constraint that they have to live in the same *address* has to be enforced. This constraint has not have to be enforced for unmarried *persons*. There is an apparent similarity here with how CRL relations are represented, but in fact there is a difference. In CRL, relations are seen as a way to describe a kind of user-defined inheritance from the range to the domain, represented as pointers in the participant

frames. Owing to this directional nature, it is difficult to represent bi-directional concepts in CRL, such as the address constraint seen above. For instance a CRL relation can be defined such that *wife's address* would be inherited from *husband's address*. But here, changes to the *wife's address* do not affect the *husband's address*, since changes are just transmitted from the range to the domain of the CRL relation. In our approach, relations do not have a directional nature.

6.3.1 Operational Semantics for relationships

As mentioned in the introduction, SDMs have to specify how abstract relationships behave regarding insertion, deletion and modification operations on the related abstract objects (i.e. classes). Since, in OODBs, objects are described not only by their attributive features but also by their behaviour (i.e. methods), a similar question to the one arising in SDMs can be posed but now in the context of user-defined relationships and *ground* objects (i.e. instances). How does a user-defined relationship behave in response to operations performed on its participant objects? As an example, consider the *marriage* relationship and assume that the class *person* has a method *moves* attached to it. The designer may be interested in modelling the situation where the 'movement' of a married *person* involves the movement of his/her partner, i.e. when the message *moves* is sent to a *person*, this message is propagated to his/her partner *through the marriage relationship*.

This leads to a new mechanism for sharing behaviour. However, unlike previous approaches where sharing is defined at the class level, now relationship-based sharing can be specified. Hence, the behaviour of an object comes not only from the class to which it belongs but also from the classes to which it is related. This is after all quite realistic!

In [Diaz 91a] we have explored two kinds of sharing, namely

- *propagation* whereby a message should be sent to other objects *besides* the one which receives the message in the first place, and
- *delegation* whereby a message should be sent to another object *instead of* being answered by the object which receives the message in the first place

Using the terminology of the *Treaty of Orlando* introduced in chapter 2, both propagation and delegation can be seen as a kind of static, implicit and per group sharing mechanisms, where here the group is based on participation in a given relationship.

Furthermore, in [Rumbaugh 88] it is shown how the propagation of an operation through a network of objects is often determined by the nature of the relationships between the objects, rather than the actual operation. For example, if the message *display* is sent to a *vehicle* object, it could be desirable to have the display message propagated to certain objects related to the *vehicle* in question, thereby providing more useful information to the user who requested that the *vehicle* be displayed. In [Diaz 91a] it is mentioned how in terms of the standard *copy* operation defined in [Khoshifian 86], two extremes are supported in the propagation of *copy*:

- **Shallow-copy**: The attributes of the original object are moved directly to the copy, with no recursive copying of related objects.
- **Deep-copy**: The scalar attributes of the original object are moved directly to the copy, and all object-valued attributes are assigned deep copies of their original values.

As pointed out by [Rumbaugh 88], these extremes of behaviour may not always represent the required behaviour. For example, in taking a copy of a *part* it may be desirable to also copy the *subparts*, but it is not likely that the *company* which makes the part should be copied as a side-effect of the copying of the part. In this case, the copy operation should be propagated over the *subpart* relationship, but not over *made_by*. In terms of *display* on *vehicle*, it may be desirable to display subparts of the *vehicle*, but uncontrolled propagation over database relationships is likely to result in the display of the *company* which made the *vehicle*, the *employees* of the company, the *children* of the employees of the company and so on.

For the *subpart* relationship this can be specified as,

```
propagating display from the_part to the_subpart
```

The relationship class *subpart* is a relationship between a *part* and its (possibly many) *subpart*. The single entry defining the *operational_semantics* of the relationship indicates

that the message called *display* is propagated from a *part* to its *subparts*. Thus when the message *display* is sent to a *part*, not only is that *part* displayed, but all *subparts* of the object are also displayed.

For the *marriage* relationships similar constructs can be used to specify its 'sharing' behaviour,

```
propagating  moves    from  the_husband  to  the_wife
propagating  moves    from  the_wife     to  the_husband
```

where *the_husband* and *the_wife* are the roles played by each of the *persons* participating in the relationship. If a message *moves* is sent to *the_husband*, besides moving himself, the message *moves* has to be propagated to the *person* playing the role of *the_wife*. A similar situation occurs if the message *moves* is sent to *the_wife* in the first place. It is worth noticing that the system has to prevent infinite loops. Such loops arises when cyclic graphs are formed by the relationships. A simpler example is illustrated by the *marriage* relationship whereby the message *moves* can be propagated to any of the participants. Hence the participant objects can keep propagating the message *moves* in turn for ever. The system has then to prevent the message from being executed more than once with the same object.

An extension to relationship definition is made to support the *operational semantics* of the relationship. The mechanism is available to any relationship class so that the user can *declaratively* specify whether messages sent to either of the related objects are to be propagated, delegated or ignored by the relationship. In the next section, a construct to define user-defined relationships in ADAM is presented.

6.4 A description language for relationships in ADAM

The aim is to provide relationships as 'first-class' objects where all their semantics are gathered together. Until now, the only class of objects available were instances of the metaclass *entity_class*. Now the question is what are the differences between those objects already provided and the relationship objects. As with any other object in the system, relationships can also have attributes and methods attached to them. However, unlike entity objects, relationships can also have special semantics, as presented in the previous

```
new([marriage,[
     related_class1([
         related_class_tuple(person,the_husband,husband_of,single,
            [sex of the_husband of marriage = male,
             age of the_husband of marriage > 15])
     ]),
     related_class2([
         related_class_tuple(person,the_wife,wife_of,single,
            [sex of the_wife of marriage = female,
             age of the_wife of marriage > 15])
     ]),
     operational_semantics([
          propagating moving   from the_husband to the_wife,
          propagating moving   from the_wife    to the_husband
     ]),
     inferred_constraints([
       address wof the_husband of marriage :: address wof the_wife of marriage
     ]),
     attribute(
        attribute_tuple(wedding_date,global,single,optional,string,[])),
     attribute(
        attribute_tuple(husband_mother_in_law,the_husband,single,optional,string,[]
     attribute(
        attribute_tuple(wife_mother_in_law,the_wife,single,optional,string,[]))
]]) => relationship_class.
```

Figure 6.3: The *marriage* relationship definition in ABEL.

section. A set of meta-attributes are provided to allow the designer to explicitly capture such semantics. In the following these meta-attributes are described. As an example, the *marriage* relationship definition is shown in figure 6.3 according to the ABEL syntax.

The meta-attributes **related_class1** and **related_class2** describe the objects between which the relationship is established. Both have a *related_class_tuple* tuple as their value. This tuple is defined as:

```
:- new_tuple(meta_class,related_class_tuple(
              class: string,
              role: string,
              view: string,
              cardinality: integer,
              consts: plog
)).
```

where the following information is declared,

- the name of the class participating in the relationship (e.g. *person*),
- the role played by the class in the relationship (e.g. *the_husband* or *the_wife*),
- the attribute-based view of the relationship. As in [Rumbaugh 87] relationships can be treated either as objects or as attributes or any of the participant classes. For instance, a given instance of the *marriage* relationship can be seen either as the attribute *husband_of* of a given *person* playing the role of *the_husband*, or as the attribute *wife_of* of a given *person* playing the role of *the_wife* in the relationship. These views, provided when the relationship class is created, act as normal attributes of the participant classes. For example, since *husband_of* is an attribute of a *person*, the methods *put_husband_of*, *delete_husband_of*, *update_husband_of* and *get_husband_of* are provided. Sending the message *get_husband_of* to a given male *person* will retrieve the object identifier of the *person* playing the role of *the_wife* in the *marriage* relationship instances having the object identifier of the male *person* as its *husband*. These views have proved to be very useful for the user and represent the links of the CE paths,
- the cardinality of the relationship, either *single* or *multi*,
- a set of constraints on the object participating in the relationship (e.g. person to be male)

Attributes of the relationship itself such as *wedding_date* can be specified as in any other class.

The meta-attribute *operational_semantics* specifies whether messages sent to any of the related objects are to be propagated, delegated or just ignored by the relationship. This attribute is multivalued and thus, a set of propagating and delegating constructs can be specified according to the following syntax,

```
propagating  <method_selector>  from  <role_name>  to  <role_name>
delegating   <method_selector>  from  <role_name>  to  <role_name>
```

where role names must be different and coincide with any of those specified in the *related_class* attributes.

So far only features of the relationship itself have been presented. Nevertheless, one of the novelties of this approach is that relationship semantics also involve what we have called *inferential knowledge*, i.e. participant objects can have new attributes and new constraints as a result of the establishment of the relationship. This is a quite new idea that need further research but it has already proved useful for modelling [Segler 91].

New 'inferred' attributes can be defined as for any other attribute except that the visibility facet has to be either one of the role names. In figure 6.3 the *husband_mother_in_law* attribute is defined. As a result, the object playing the role of *the_husband* has a mother in law once the relationship is established.

New 'inferred' constraints are declared as values of the meta_attribute **inferred_constraints**. These constraints are specified as for any other constraint in the system where the anchor (i.e. the beginning of the path) is the relationship class. In figure 6.3 an example is shown where *the_husband* and *the_wife* are constrained to live at the same *address* once the *marriage* has taken place.

Once a relationship is defined, instances can be created in the usual way, i.e. by sending the message *new* to the class. In the following, a *marriage* relationship is established between the *6@person* and the *32@person*, playing the role of *the_husband* and *the_wife* respectively, where the *wedding_date* is also specified.

```
new([OID,[
      the_husband([6@person]),
      the_wife([32@person]),
```

```
        wedding_date([1991])
]]) => marriage.
```

The attribute-based view of the relationship can be used to query the relationship. For example, to retrieve the husband and wife of a given *person* the following message can be invoked,

```
| ?- get_husband_of(X) => 6@person.

X = 32@person;

| ?- get_wife_of(X) => 32@person.

X = 6@person;
```

In the same way, relationships can be created using the attribute-based view. For instance, any of the following two messages have the same effect [1] as the previous *new* message, i.e. the creation of a new instance of *marriage*,

```
put_husband_of([32@person]) => 6@person.

put_wife_of([6@person]) => 32@person.
```

However the underlying process is transparent from the user, that just sees *wife_of* as another attribute of *person*. This transparent behaviour can be provided since all attributes are only accessed by methods. Hence, it is transparent to the user whether or not an attribute is defined together with the class or derived from a relationship. It is just another advantage of data encapsulation: uniform access to data regardless of its nature.

6.5 Specialisation of relationships

Specialisation allows a designer to begin by modelling general concepts and then to proceed with consideration of the more specific cases. Since now relationships are seen as objects, this abstraction mechanism is available to relationship definition.

Specialisation of relationships is achieved by 'specialising' any of the related class attributes. Let RC1 be a related class attribute of a given relationship which has as value

[1] This is not completely true since relationship attributes cannot be specified if the attribute-based view mechanism is used to create the relationship.

```
related_class_tuple(Class1,Role1,View1,Card1,Consts1)
```

A related class attribute RC2 having as value

```
related_class_tuple(Class2,Role2,View2,Card2,Consts2)
```

is a valid 'specialisation' of RC1 if:

- Class2 is a subclass of Class1
- Role1 is equal to Role2. Roles cannot be changed
- Card1 is equal to Card2. Cardinality has to rest unmodified
- Further constraints Consts2 can be added

The specialised relationship can specify other attribute-based views, and further inferred attributes and constraints can be added. As an example, consider the *working_in* relationship between the class *person* and the class *place*. This relationship is shown in figure 6.4. *Working_in* can be specialised into *working_in_projects* and *working_in_research* to consider additional features of *working_in* in different contexts.

Working_in_projects specialises the participant class of the *related_class2* to be an *enterprise* -a subclass of *place*- and adds a new inferred attribute called *project_incomes*. Since *related_class1* is not specified, it has the same *related_class1* that its superclass, i.e. the other participant class is a *person*. *Working_in_research* specialises both participants. The *person* involved in research has to be a *lecturer* -a subclass of *person*- whereas the *place* where this research is carried on is specialised to be a *research_institution* -a subclass of *place*. Two inferred attributes, *publications* and *grants*, are defined so that only *lecturers working_in_research* can have a value for these attributes. Of course, any characteristic of the superclass relationship are inherited by its subclasses. Other relationships can be defined as subclasses of *working_in* and only the specialised or additional features have to be defined.

Besides the normal advantages attached to the use of generalisation (e.g. reusability, legibility, economy, etc), general queries can now be easily expressed. For example, suppose that the specialisation of *working_in_projects* and *working_in_research* is intended to cope with a different tax regulation concerning the incomes obtained from each kind

```
:- new([working_in,[
    related_class1([
      related_class_tuple(person,the_person,working_places,set,[]]),
    related_class2([
      related_class_tuple(place,the_place,employees,set,[]),
    attribute(
      attribute_tuple(duration,global,single,optional,string,[]))
]]) => relationship_class.

:- new([working_in_projects,[
    is_a([working_in]),
    related_class2([
      related_class_tuple(enterprise,the_place,workers,set,[]),
    attribute(
      attribute_tuple(project_incomes,the_person,single,optional,integer,[]))
]]) => relationship_class.

:- new([working_in_research,[
    is_a([working_in]),
    related_class1([
      related_class_tuple(lecturer,the_person,works,set,[]]),
    related_class2([
      related_class_tuple(research_institution,the_place,researchers,set,[]),
    attribute(
      attribute_tuple(publications,the_person,set,optional,string,[])),
    attribute(
      attribute_tuple(grants,the_person,set,optional,integer,[]))
]]) => relationship_class.
```

Figure 6.4: Relationship specialisation in ABEL.

of activity. If now we are just interested in the *working_places* of a given *lecturer* (e.g. 2@lecturer) regardless of whether they are projects or research, we can work at a higher level in the hierarchy, above the specialisation, just by using the attribute-based view of *working_in* (i.e. *working_places*). For instance the following message,

```
get_working_places(X) => 2@lecturer.
```

retrieves all the *working places* of *2@lecturer* regardless of its nature (projects or research). Without the generalisation, it would have been necessary to ask for each of the relationships in turn. Besides allowing the user to work at the level of his interest, additional specialised relationships can be added and the above single-instruction program does not need to be changed!

The notion of specialisation of relationships is not new in AI. It appears in KL-ONE [Brachman 85] where *roles* (KL-ONE relations) can be specialised. However, unlike our approach KL-ONE sees relations as attributes of *concepts* (KL-ONE entities). Hence some distinctions can be drawn:

- In KL-ONE specialisation of concepts and roles have a different representation. Two links, *a_kind_of* and *restriction*, are used to specify the specialisation of concepts and roles respectively. On the other hand, an object approach to relationship representation allows one to use the same sort of link regardless of whether the object is an entity or a relationship, conveying the same meaning in both cases.

- Whereas in KL-ONE, the specialisation of relationships is based on specialisation of the participating concepts, our proposal also includes the specialisation of the semantics attached to the relationship. For instance in the above example, *working_in_projects* specialises *working_in* not only in the participating entities, but in adding a new semantics as well, e.g. the introduction of the *project_incomes* for the *person* class.

In the following section the way in which these ideas have been implemented in ADAM is presented.

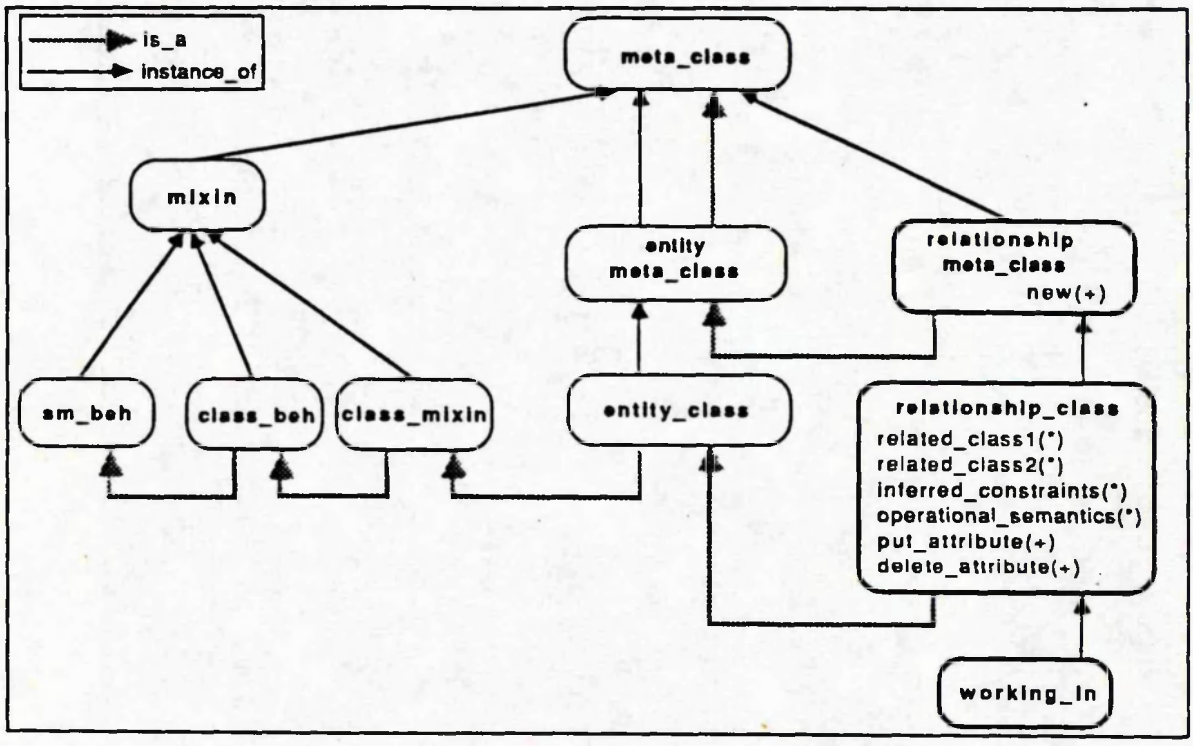


Figure 6.5: An ADAM extension to support user-defined relationships.

6.6 Extending ADAM to support user-defined relationship definition

In this section the extension to ADAM that supports constraints is described. In figure 6.5 a diagram of the extended system is shown where the symbols '*' and '+' stand for new methods or attribute definition, and specialisation of previous existing methods, respectively.

An extension to ADAM can be made along two dimensions:

- the inheritance hierarchy where the description of objects as instances is considered,
- the instantiation hierarchy where the description of objects as classes is taken into account

The question is now how are these dimensions affected by the introduction of relationship objects.

Relationship objects seen as instances are handled as any other object in the system: they are created by sending the message *new* to their classes, they can have attributes,

and messages can be sent to them. Since no further features are required, the inheritance hierarchy does not need to be extended.

Nevertheless, from the point of view of relationship classes, new requirements have been considered. Like entity classes, relationship classes can have attribute and method descriptions attached to them. So, relationship classes can be seen as entity classes. However, relationship definition involves additional semantics missing in entity definition such as the characteristics described by the meta-attributes *related_class1, related_class2* and *inferred_constraints*. Hence, the metaclass knowing how relationship classes can be defined (known as **relationship_class**) is a subclass of the metaclass **entity_class**. *Relationship_class* inherites the normal behaviour of an entity object, besides defining new features to accomplish for the extended semantics of relationships.

Our definition of *relationship_class* metaclass is as follows:

```
:- new([relationship_class,[
is_a([entity_class]),
slot(slot_tuple(related_class1,global,single,optional,related_class_tuple)),
slot(slot_tuple(related_class2,global,single,optional,related_class_tuple)),
slot(slot_tuple(inferred_constraints,global,set,optional,plog)),
slot(slot_tuple(operational_semantics,global,set,optional,plog)),

    % This methods are specialised to consider the new requirements of the
    % inferred attributes whose visibility is not the standard one

method((put_attribute(global,[],[slot_tuple],[],[Slot]) :-
     ..................)),

method((delete_attribute(global,[],[attribute_desc_tuple],[],[AttributeDesc]) :-
     ..................))
]]) => relationship_meta_class.
```

The four slots [2] described in the above definition hold part of the additional semantics required for relationship objects. But these slots alone are only repositories of data that needs to be interpreted. For instance, values of the *operational_semantics* will convey real meaning if adequated procedures are available that achieve the expected behaviour specified declaratively in the slot.

Since values to both *operational_semantics* and *inferred_constraints* [3] can be inserted

[2] For reasons mentioned in chapter 5, attributes cannot be defined when metaclasses are created.

[3] It is interesting to note that the designer has to be careful not to define new constraints that are not satisfied by already existing instances. Otherwise, the DB will become inconsistent.

once the relationship is created, the corresponding 'interpreters' are placed within the *put_* and *delete_* method attached to these slots. This can be achieved as follows:

```
:- replace_method([
      (put_operational_semantics(global,[],[plog],[],[OpSem]) :-
        ..............)
]) => relationship_class.

:- replace_method([
      (delete_operational_semantics(global,[],[plog],[],[OpSem]) :-
        ..............)
]) => relationship_class.

:- replace_method([
      (put_inferred_constraints(global,[],[plog],[],[Consts]) :-
        ..............)
]) => relationship_class.

:- replace_method([
      (delete_inferred_constraints(global,[],[plog],[],[Consts]) :-
        ..............)
]) => relationship_class.
```

However, *related_class1* and *related_class2* have to be considered only when the relationship is created, i.e. when the message *new* is sent to *relationship_class*. This method is then defined at the meta-meta-level. By default, the method *new* to create entity classes is inherited from *entity_meta_class*. But this behaviour is not enough. It has to be specialised to include the interpretation of the related-class attributes. Therefore, the **relationship_meta_class** object is defined as a subclass of *entity_meta_class* where the method *new* is specialised. This definition is as follows

```
:- new([relationship_meta_class, [
    is_a([entity_meta_class]),
    method((new(global,[],[string,plog],[],[Name,Atts]):-
      ( \+ member(is_a([_]),Atts)
        -> member(Class1,Atts),
           member(Class2,Atts),
           % Check that classes to be related exist
           ..............
           % Create role attributes
           ..............
           % Create attribute-based view of the relationship
           ..............
        ;  % It must be a specialisation of an existing relationship
```

```
            % Check that class to be related exist
            ..............
            % Check that the specialisation is correct
            ..............
            % Create role attributes (if required)
            ..............
            % Create attribute-based view of the relationship (if required)
            ..............
        ),
        % Create attributes and generate rules for constraint enforcement
        ..............
    ))
]]) => meta_class.
```

This semantics is now declared in the meta-attributes defined in the relationship class (e.g. *working_in*) and interpreted by methods specified either in the *relationship_class* object or in the *relationship_meta_class* object.

6.7 Conclusion

To cope with the increase in complexity of the domains tackled by DBs, more powerful semantic tools are needed. Here a proposal has been presented to enhance an OODB with semantic-rich user-defined relationships. As with any other object, relationships can have attributes and methods or be arranged in hierarchies. An attribute based approach is also described so that programs accessing the DB before this integral view of relationships was considered do not need to be changed.

Besides increasing the domain semantic kept in the DB, the enhanced relationship specification allows users to have a uniform understanding of the data stored in the DB as well as gathering together all the information about a relationship, that would otherwise be spread throughout different objects and user-programs. So far, only binary relationships have been taken into account. By allowing us to represent both structural and behavioural aspects of the UoD, OODBs have been shown to offer the right paradigm to implement user-defined relationships. These ideas can be considered as another step on the way to enhancing knowledge in DBs.

The main contributions of this work can be summarised as follows:

- Although we do not claim to have the novelty of defining relationships as objects, it is less common to find in the DB literature work investigating the 'inferred' properties of relationships. In this chapter some insights into inferred attributes and constraints have been presented and implemented.

- The idea of the operational semantics for relationships, originally proposed for programming languages in [Rumbaugh 87], has been extended to consider both propagation and delegation in the context of OODBs. Furthermore, unlike the approach in [Rumbaugh 87], a rule-based mechanism has been used for supporting this semantics.

- As with objects, relationships can be arranged in hierarchies. In this chapter, an approach has been presented and implemented for relationship specialisation.

Chapter 7

Making active behaviour explicit

7.1 Introduction

Active database systems have been defined as "database systems that respond *automatically* to events generated internally or externally to the system itself *without user intervention*" [Bauzer 90]. System responses are declaratively expressed using event-condition-action rules (ECA rules proposed in [Dayal 88]). An ECA rule has an *event* that triggers the rule, a *condition* describing a given situation, and an *action* to be performed if the condition is met. In this way, not only does the system know **how** to perform operations, but also **when** those operations must be performed.

ECA rules should not be confused with methods or situation-action rules. An ECA rule definition includes not only what to do but also when to do it, and can be seen as behaviour exhibited as a result of some event taking place, such as accessing or updating an attribute. As an example, if a *boiler* object has an attribute *temperature*, and it is known that when the value of the temperature rises to 30 degrees the alarm has to go off, an ECA rule seems the right paradigm to represent this event-driven behaviour within the system. Unlike ECA rules, methods are invoked explicitly by sending a message to an object, so method invocation can be seen as a kind of procedure call [Stefik 86].

On the other hand, situation-action rules, as found in production systems such as OPS5 [Forgy 81], lack the trigger part. For instance, in OPS5 the triggering process is done by the evaluation phase of the inference engine, where the situation part of the rule is matched against the current state of the problem described by working memory elements,

i.e. tuples. Unlike production systems, DBs have to cope with larger domains where this matching mechanism is not feasible. However, an event-driven system can provide a mechanism for explicit support of integrity constraints, derived data or rule-based inferencing, besides providing timed responses and modularity to support time-constrained applications.

In [Beech 90] rules are seen as a major feature of future database systems, and it is remarked that "object-oriented database (OODB) researchers have generally ignored the importance of rules". The research presented here is an attempt to provide an insight into rules in an OO context. The focus is on providing a **uniform approach.**

What is meant by a uniform approach is that rules have to be defined and treated in the same way as other objects in the system, without defining any additional mechanisms or auxiliary structures. Rules are seen as 'first-class' objects, and are described using attributes and methods. In this way, rule management operations are conceived and implemented as methods. This brings all the advantages of the OO paradigm into rule management: encapsulation, modularity, reusability. In a uniform approach the system should not distinguish rules from other kinds of object. As a result, rules can be related to other objects, and also arranged in hierarchies: Since methods attached to objects can trigger rules, and rules are themselves objects, rules can be defined which are triggered by methods attached to rules. As with any other entity, the meaning of a rule lies in the attributes attached to the rule. and their interpretation by the associated methods. From the point of view of the system, however, no distinction should be made. Treating rules as objects also has the advantage that any new facility introduced for objects is automatically applicable to rules (e.g. transaction mechanisms, locking mechanisms, display facilities).

Rule evaluation imposes an overhead on every possible event that can be detected by the system. Whereas in relational databases events are generally restricted to be database updates, the approach presented here allows any message to raise an event. Thus, the efficiency requirements for rule support in OODBs are even greater that in relational databases. Here, an attempt is made to enhance system performance by indexing rules by class. A single thread of execution is assumed, and topics such as transactions and optimisation have not been addressed.

This chapter is organised as follows. A review of related work is given in section 2. In section 3, the components involved in rule management are identified. Issues relating to events in an object oriented context are discussed in section 4. In section 5 the implementation of a rule manager in ADAM is described. An approach for deriving rules for constraint maintenance is introduced in section 6. Finally, conclusions are presented.

7.2 Related work

Research on active behaviour has been conducted in the areas of programming languages, Artificial Intelligence(AI) and DBs. ACTOR [Hewitt 77] was a pioneer programming language in providing objects with active behaviour. Modelling parallel and distributed applications are among the research interests in this area [Ellis 89]. Active behaviour in AI is provided through procedural-attachments. So procedural-attachments such as *if-needed* or *if-added* are associated with slots to compute their values on demand, or to perform some other test or action.

In relational DBs, active capabilities have been used to enforce integrity constraints, define views, translate update requests and compute derived attributes [Eswaran 75, Stonebraker 90, Morgenstern 84]. In [Beech 90] rules are seen as a unifying paradigm for providing a broad range of DB facilities. However, in relational DBs, rules are implemented as a distinct layer, and additional mechanisms and structures are required to support rule management.

Several OO systems that support rules are described in the literature [Kotz 88, Dayal 89, Hudson 89, Chakravarthy 89, Bauzer 90]. In [Bauzer 90] a review of different mechanisms for supporting rules is given, namely:

- method-based mechanisms: the rule is precompiled into each place in the code where it might be activated.
- object-based mechanisms: enlarging the object description to indicate which rule to invoke whenever message sending takes place. This is the approach followed in this paper
- external mechanism: additional structures are defined which support checking when some event occurs (e.g. [Bauzer 90, Kotz 88])

Several drawbacks can be enumerated for the first approach:

1.- Rules are buried inside methods, and thus it is difficult to enquire about any of the rules attributes, e.g. the condition, the action, or whether it is enabled or not.

2.- Modification of any of the attributes of a rule implies making changes to every method supporting the rule.

3.- Since rules can interact, coding of rules within methods requires that the programmer understands all the rules that appear in the method, so that interaction can be handled properly.

4.- The rule definition is scattered, compromising the OO philosophy that encourages all information about a given object to be gathered together.

5.- Method code now includes two things: how the operation itself is implemented and the enforcement of the rule. This severely compromises *method overriding.* Overriding of methods is a useful mechanism in OO systems for customising an operational implementation for special requirements. The problem is that in this case not only are we overriding the operation but also the embedded concept described by the rule (e.g. an integrity constraint).

In [Beech 90] some of these drawbacks are pointed out and the following conclusion is drawn: *"In our opinion there is only one reasonable solution; rules must be enforced by the DBMS but not bound to any function (i.e. method) or collection"*. The other two approaches to supporting rules overcome these disadvantages by providing a mechanism supported by the DBMS.

In [Bauzer 90] a rule management mechanism is proposed for O_2. Rules are objects having the event as an attribute, and auxiliary structures are defined for storing rule lists which are checked when specific events occur. However, events are not seen as objects in themselves, and thus, system extensibility can be compromised in the sense that composite events or events with special requirements are difficult to introduce. Further, a local mechanism is used to provide rule 'inheritance' instead of using a mechanism based on the object hierarchy itself.

In HiPAC [Dayal 88] rules and events are seen as different entities with their own at-

tributes and methods. A sound approach is taken to rule support, paying special attention to transaction management and optimisation techniques. However, some of the idiosyncrasies of the OO paradigm have not been considered, such as the primary role that classes play, in which methods are part of the class definition.

7.3 An overview of rule management

The OO paradigm provides a different approach to system design. Whereas procedural design emphasizes the decomposition of the problem into a set of tasks to be executed sequentially, OO design focuses on the entities involved and how they interact. Thus, to provide a rule manager in the context of an OODB, a primary requirement is to identify the significant entities and their interaction.

Briefly described, the function of a **rule manager** is to provide quick response through the use of *rules*, to *events* generated by some *system*. Three components can be identified in this process:

- *the rule* describes both when and how the system reacts to an event.
- *the event* is an indicator that signals that a specific situation has been reached to which reactions may be necessary [Kotz 88]. Not all systems consider events as first class objects. For example, events can be treated as simple attribute values. However, this approach can compromise the ability to extend the system to cope with events coming from different sources, or events that need special treatment (e.g. composite events [Dayal 88]).
- *the event generator* can be seen as any system producing events which may need a special response in terms of rule triggering. Events can be generated by the DBMS itself or by any other external system such as a clock or an application program.

Figure 7.1 shows an Entity-Relationship diagram where these entities are depicted together with the relationships between then. First, a rule can be triggered by an event, but an event can trigger several rules. Second, an event can be generated by several systems and a system can generate several events.

The main interaction between these entities can be described as follows:

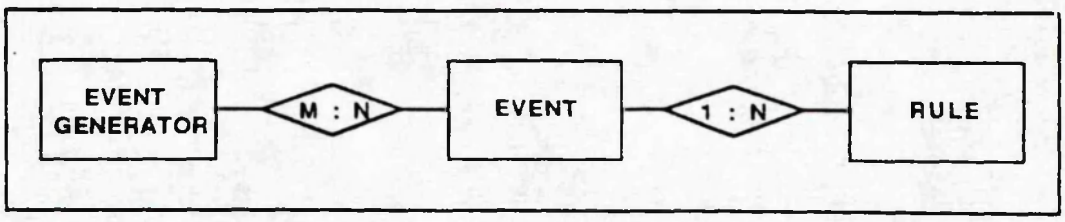


Figure 7.1: E/R diagram for rule management

1.- an event is produced by any event-generator, and is signalled to the event manager through the message *signal*,

2.- the event manager checks whether any rule can be triggered by the event signalled. If so, it sends the message *fire* to the appropriate rules,

3.- when the message *fire* is received by a rule, the rule condition is then checked and, if satisfied, the rule action is executed

Other kinds of interactions are also possible, such as the 'awakening' of events as a result of rule creation.

In the following sections the object **rule** and the object **event** are defined. Event generators have not been described as objects, although conceptually they are seen as the senders of the signals.

7.4 Events in an object oriented context

An event is an indicator that signals that a specific situation has occurred to which reactions may be necessary. In relational DBs, an event can be described by the operation together with the moment when this operation takes place (i.e. before or after method invocation). For instance, the pair *(insert,before)* could specify that the event arises *before* the operation *insert* occurs. In this context, OODBs present some differences from relational DBs. In OO systems, operations (i.e. methods) are not isolated but are part of the class definition. The class is not just an argument of the method, but the method itself is subordinate to the class. As a result, the same method name can be implemented in different ways in distinct classes, the process known as overloading, or a method can be specialised down the hierarchy by any of the subclasses, thereby revising the behaviour of the superclass. Now, let us consider the situation shown in figure 7.2

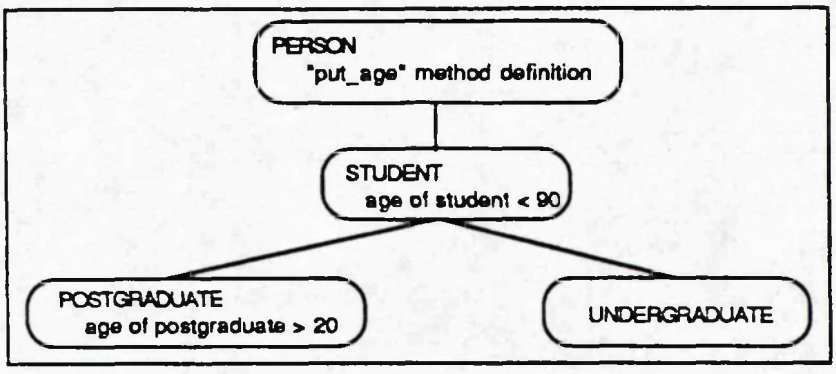


Figure 7.2: Person hierarchy

where an integrity rule to prevent students from being older that ninety is defined. This rule should be fired for instances of the class *student before* the message *put_age* begins execution, i.e. before the student age is altered. Since OO systems allow methods to be inherited from superclasses, the method *put_age* can be defined at the level of the class *person* and inherited by the subclass *student*. For this integrity rule to be fired it is necessary not only that an attempt has been made to insert the *age* of a *person*, but also that this *person* happens to be a *student*. Otherwise, the rule should not be invoked even if the message *put_age* is detected.

In other words, the method alone does not completely specify the context of invocation, since a method gets its meaning from a class (subsequently called the **active class**). Several alternatives are possible for supporting the idea of an active class, for instance:

1.- The active class could be embedded in the condition part of a rule. For example, the previous rule would have as an event the pair *(put_age,before)* and the condition part of the rule would be extended to check that the receiver of the message is an instance of the class *student*. Besides making the context where a rule is invoked difficult to understand, this approach prevents the system from taking full advantage of the active class as an indexing mechanism, as shown later.

2.- The event definition could be enlarged with an active class attribute. Thus, in the previous example, the event would become *(student,put_age,before)*. However the message receiver can be an instance of some subclass (e.g. *postgraduate*), and thus the active class is not its immediate class. Two options are now possible. One is to check whether the message receiver is an instance of the active class (i.e.

student). This process can turn out to be quite expensive since this checking has to be done for every message sent and for every possible event. Another approach is to generate all possible 'inherited' events automatically. For instance, the events *(postgraduate,put_age,before)* and *(undergraduate,put_age,before)* would be generated, providing that *postgraduate* and *undergraduate* are subclasses of *student.* It is worth mentioning that some of the generated events may already be defined (e.g. an integrity rule constraining postgraduate students to be older than twenty). In this case, instead of creating a new event, the set of rules activated by this event has to be extended by the identifier of the *younger-than-ninety* rule. Moreover, if a new subclass is introduced, the appropriate events have to be generated. For instance, if *phd_student* is introduced as a subclass of *postgraduate*, all the events for postgraduate students have to be 'inherited' by PhD students. This process can be quite cumbersome and expensive to maintain. In our opinion, the rule identification process should make use of the class hierarchy itself, rather than making use of some additional mechanism.

3.- The rule definition is extended with an **active_class** attribute. Previous work either does not consider explicitly the role played by the active class, or provides a local mechanism for 'inheriting' events. Since rules are truly objects, the extra *active_class* attribute can be implemented as a two-way relationship between rules and classes. The inverse of *active_class* is declared to the system to be held in the **class_rules** attribute of a class. For instance, the *younger-than-ninety* rule would have *student* as the value of its *active_class* attribute, and thus the *student* class would have the object identifier of this rule as the value of its *class_rules* attribute. This approach has two important advantages:

- Rules are indexed by class. The *class_rules* attribute has as its value the set of rules to be verified when a message is sent to any instance of this class. In this way the search for applicable rules is considerably reduced.
- The 'inheritance' of rules has been moved to the class hierarchy, without defining any additional mechanism. As discussed above, the rules affecting a given instance are not just the ones attached to its immediate class, but also those attached to its superclasses. For example, if the message *put_age* is sent to an instance of the class *postgraduate*, the rules applicable (e.g. representing

integrity constraints on the *age* attribute) are those attached to *postgraduate* itself together with those attached to *student* and *person*. To handle this situation, the definition of each class has been enlarged with the attribute: **activated_by**. This attribute is defined just like any other attribute:

```
attribute(
  attribute_tuple(activated_by, global,set,optional,generic_rule,
   [activated_by  wof class_beh :: activated_by of is_a of class_beh
                                        union
                                  class_rules  of class_beh]))
```

This definition states that *activated_by* contains objects of type *generic_rule*. The constraint, enclosed between brackets and specified using the constraint equation approach described in chapter 5, enforces that the value of the *activated_by* attribute for a given class has to be equal to the union of the rules of the *class_rules* attribute attached to the class and the rules obtained from the *activated_by* attribute attached to its superclasses. It is worth noticing the recursive nature of this constraint. This together with the idea of *weak bond* [1] achieves the desired behaviour: when an update is made to the *class_rules* attribute of any class, the update is propagated to the *activated_by* attribute of all its subclasses. This is done automatically by the system as a result of the enforcement of the above constraint, without any further mechanism being required. Furthermore, when a new subclass is introduced, the appropriate rules are 'smoothly inherited'.

The latter approach is described in detail in the next section, where a rule manager is described for ADAM.

7.5 Extending ADAM to support rule management

7.5.1 The event object

Events are not always seen as first-class objects. In [Bauzer 90], events are seen as rule attributes, and hence they cannot have attributes or methods of their own. Although this approach may result in performance gains, it can compromise the ability to extend

[1] A weak bond -syntactically represented as *wof*- can be seen as the link to be broken to preserve the constraint when the equality is violated (see chapter 5 for further details).

the system for coping with events coming from different places, or which need special treatment.

As with other objects in the system, event definition involves the description of structure (i.e. attributes) as well as behaviour (i.e. methods). An event can be seen as specifying the moment when a rule is to be fired. This moment can be described by the message firing the rule (the **active_method** attribute of the event) and the status of the message (the **when** attribute of the event).

Unlike some previous approaches, a richer and more complex event definition can be created as a result of working in an OO environment, namely:

1.- Events are not restricted to update operations but can be any message defined in the system (e.g. display, create a new class, move, get_age).

2.- The possible values describing the status of a message can be enlarged. Previous work has considered just two values: *before* and *after* operation execution. In the OO context, operations are materialised by methods. Besides *before* and *after*, the range of values has now been extended to take into account situations where the method cannot be found, or other options which reflect the nature of method invocation supported by the underlying PROLOG evaluation strategy (e.g. backtracking into a method) [Diaz 91a]. This broader spectrum of situations attempts to reflect the core role that methods play in OO systems and the variety of situations that can arise during message sending.

In other approaches, the event description includes the arguments to be passed when a rule is fired. In our approach, all the methods' arguments, regardless of whether they are input or output parameters, are passed by the system without any previous declaration. The rule manager makes these arguments available to the condition and action part of the rule through the system-defined predicate *current_arguments*. The instance to which the rule is applied is also available through the predicate *current_instance*. Examples are given in the next section.

Being objects, events can be related to other objects (e.g. to the rules that a given event activates), or arranged in hierarchies. This allows the system to be enlarged to cope with later extensions. Events can be manipulated and signalled by some event generator as

well as created, modified or deleted in a *uniform* fashion. For example, an event can be created by sending the following message:

```
new([OID,[
      active_method([put_age]),
      when([before])
]]) => generic_event.
```

This event is raised *before* the method *put_age* is executed.

The classes *db_event*, *clock_event* and *application_event* share some attributes and behaviour which are abstracted at a higher level in *generic_event*. Moreover, the procedure which takes place when a new event is created is the same as the one followed for the creation of any other object (e.g. instances of *person*). Nor are the deletion or modification methods distinguishable from those used for deleting or modifying other objects. As a result, this behaviour can be inherited from that already provided by the system, i.e. from the *entity_class*.

7.5.2 The rule object

Rule structure is described mainly by *the event* that triggers the rule, *the condition* to be checked and *the action* to be performed if the condition is satisfied. The condition is a set of queries to check that the state of the database is appropriate for action execution. The action is a set of operations that can have different aims, e.g. enforcement of integrity constraints, user intervention, propagation of methods, etc. Condition and action definitions can refer to the object to which the rule is being applied and to the current arguments of the method firing the rule through the system-provided predicates *current_object* and *current_arguments* respectively.

As discussed in the last section, the complete context of invocation is described by the **active_class** and **event** attributes. The *event* attribute has as its value the object identifier of an event instance. In figure 7.3 an example is given of an active rule to prevent students from being older that ninety. If *3@db_event* is the object identifier of the event shown in the last section, this rule will fire *before* executing the *put_age* method. The condition checks whether the argument of the method (i.e. the age to be introduced) is greater than ninety or not. If the condition is not met, the rule is not applicable and the method can continue. Otherwise, the action is executed. In this case, the action fails

```
new([OID,[
    event([3@db_event]),
    active_class([student]),
    is_it_enabled([true]),
    disabled_for([1@student,23@student]),
    condition([(
        current_arguments([StudentAge]),
        StudentAge > 90
    )]),
    action([(
        current_object(TheStudent),
        current_arguments([StudentAge]),
        get_cname(StudentName) => TheStudent,
        writeln(['The student ',StudentName,
                 'with age ',StudentAge,
                 'exceeds the expected age']),
        fail
    )])
]) => integrity_rule.
```

Figure 7.3: A rule to prevent students from being older than ninety.

after displaying a message, preeventing *put_age* from proceeding. A condition result can also be passed to the action part through the *condition_result* predicate, the argument of which is instantiated with any value required after condition evaluation.

As well as the event-condition-action description, two more attributes are added to specify the status of the rule itself, i.e. whether the rule is enabled or disabled. The attribute **is_it_enabled** describes the status at the level of the whole class appearing as the *active_class* value, whereas the **disabled_for** attribute describes the status for specific instances of this class. In the above example, the rule is enabled for all the students (i.e. the value of *is_it_enabled* is *true*) except for those instances with object identifier *1@student* and *23@student*. Thus, this rule will not be fired if either the *is_it_enabled* attribute is *false* or if the object identifier of the current object appears as one of the values of the *disabled_for* attribute.

As part of their structural description, rules can be related to other objects in the system and arranged in hierarchies. Actually, the *active_class* attribute is a relationship between classes and rules that has been used to speed up the system: the inverse of *active_class*, i.e. *class_rules* attribute, is used as a class-based index where the inverse constraint is

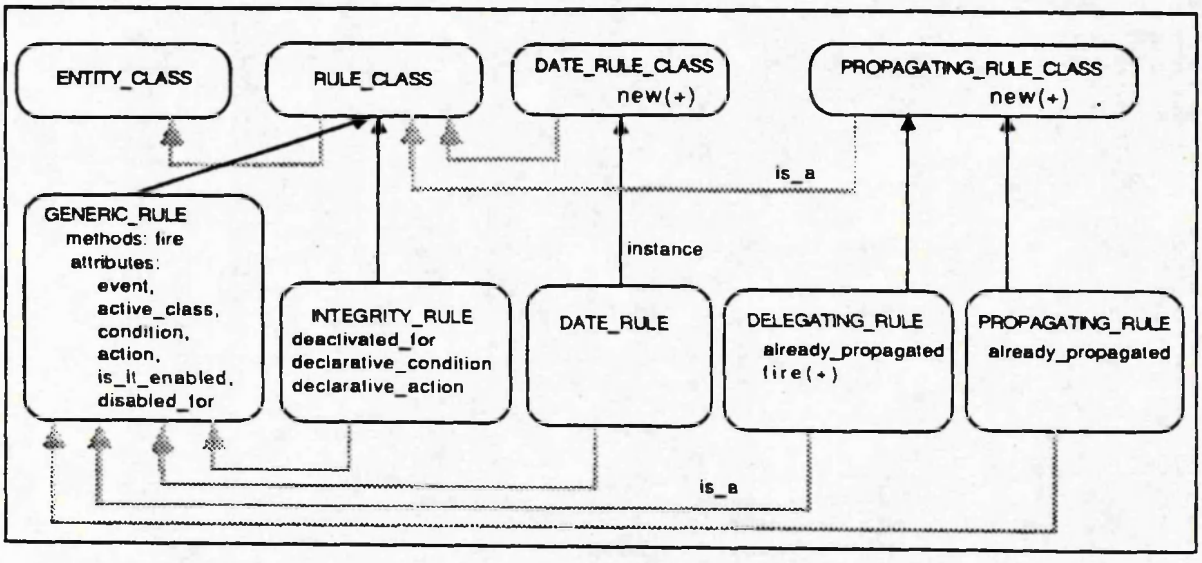


Figure 7.4: Rule hierarchy

maintained by the system (see chapter 5). Other relationships can be defined, even between rules themselves, e.g. a precedence relationship in the order of execution.

Arranging rules in hierarchies brings all the advantages of inheritance to rules. In figure 7.4 the *generic_rule* class is specialised into several subclasses, such as *date_rule*, *integrity_rule* or *propagating_rule*. User-defined rules are those defined by the user, whereas integrity rules and propagating rules are system-generated, the former from declarative specification of integrity constraints and the latter from the operational semantics of relationships. Rules for constraint maintenance are interesting examples involving several methods and classes. For instance, to preserve that *the age of a PhD student is always smaller than the age of his/her supervisor*, all methods modifying the *age* (i.e. *put_age*, *delete_age* and *update_age*) either of a *student* or of a *lecturer* must be considered (assuming that the class *lecturer* is defined and that only lecturers can supervise PhDs). An approach to derivation of rules for constraint maintenance in this context is presented later in this chapter.

The next question to be addressed is the behaviour of rules. Unlike other objects, rules can be fired, i.e. the condition of the rule is evaluated, and if satisfied, the action is executed. In order that it may be inherited by all the instances, this rule firing method is defined at the level of *generic_rule* class. The method **fire** can be specialised to account for further requirements in any subclass. Enabling and disabling of rules is managed through modification of the *is_it_enabled* and *disabled_for* attributes, and no

```
new([_,[
    event([day 10 month 'Oct' year every hour 20:15]),
    is_it_enabled([yes]),
    condition([.....]),
    action([(.....)])
]]) => date_rule.
```

Figure 7.5: A rule triggered by a time event.

special methods are required.

Finally, rule management is implemented using the mechanism already provided by the system for handling other kinds of objects. However, special requirements arise when instances of *integrity_rule* and *propagating_rule* are created and therefore, the method *new* has to be specialised for these classes. Owing to the metaclass mechanism available in ADAM, this specialisation can be easily and cleanly supported by defining the *special_rule_class* class. This situation is shown in figure 7.4. All rule classes are handled in the same way. However, when the message *new* is sent to the *user_defined_rule* class, the 'standard' definition of *new* is inherited, whereas a specialised definition is used when this message is sent to the *integrity_rule* or *propagating_rule* classes.

7.5.3 Some examples

For the sake of simplicity, in the current implementation where only atomic events have been considered, event definition has been embedded within rule definition. Therefore, rule definitions have been extended with the *when* and *event* attributes representing the *when* and *active_method* attributes of the *event* object respectively.

Three examples are given in this section, illustrating the use of rules for timed reactions, security enforcement and meta behaviour. In the former the use of clock events is shown. Events are not restricted to method selectors but any date or time may also be used as a trigger. These events are declared according to the following syntax:

```
day <day> month <month> year <year> hour <hour> : <minutes>
```

```
new([_,[
    active_class([generic_rule]),
    event([new]),
    when([before]),
    is_it_enabled([yes]),
    condition([(
        \+ current_user(graham),
    )]),
    action([(
        write('You are not authorised to create generic rules'),nl,
        fail
    )])
]]) => generic_rule.
```

Figure 7.6: A rule on rules.

where <day>, <month>, <year>, <hour> and <minutes> have the obvious range restrictions. Furthermore, for <day>, <month> and <year> the key word *every* can be specified, meaning that the event must be detected every day, month or year respectively. In figure 7.5 a rule is shown that must be fired on 10th October of every year at a quarter past eight in the evening. Whereas message-sending events are detected by the DBMS, time events are signalled by the external UNIX system. When a clock event is created, a UNIX demon is placed in the operating system and from then on, the demon will be responsible for detecting and signalling the clock event.

In the second example the advantage of following a uniform approach can be seen. Since rules are objects, rules can be defined on the rules themselves. In figure 7.6 a rule is shown that prevents users other than *graham* from creating instances of *generic_rule* class. It is thus a rule about rules. When an attempt is made to create a rule by sending the message *new* to the *generic_rule* class, this rule is fired, and the identity of the user checked through *current_user*, a predicate which returns the name of the current user.

Finally, the third example illustrates the use of rules to accomplish metabehaviour in relationships. As described in chapter 6, relationships are 'first-class' objects in ABEL. However an attribute-based view of the relationship is provided so that the objects between which the relationship is established can handle the relationship as any other attribute. Hence a relationship instance can be created either directly by sending the message *new* to the relationship class, or indirectly from any of the participant objects by

```
new([_,[
 event([new]),
 active_class([relationship_class]),
 when([before]),
 is_it_enabled([yes]),
 condition([(
    current_object(RelClass),
    condition_result([Role1,View1]),
    get_related_class1(related_class_tuple(_,Role1,View1,_)) => RelClass
 )]),
 action([(
    current_object(RelClass),
    current_arguments(Args),
    condition_result([Role1,View1]),
    Args = [Oid,Atts],
    Role1Pred =.. [Role1,[Role1Val]],
    member(Role1Pred,Atts),
    get_related_class2(RC2,RelClass),
    RC2 = related_class_tuple(_,Role2,_,_),
    Role2Pred =.. [Role2,[Role2Val]],
    member(Role2Pred,Atts),
    put_name(View1,View1Put),
    signal([event_tuple(View1Put,before,Role1Val),[Role2Val]]) => generic_rule
  )])
]]) => generic_rule.
```

Figure 7.7: A rule on metaclasses.

sending the message *'put_* view' to the appropriate object. As an example consider the *working_in* relationship shown in figure 6.4, with *works* and *employees* as the attribute-based view of the relationship. The following three messages each have the same effect: the creation of an instance of *working_in*,

```
new([Oid,[the_person([1@person]),the_place([3@place])]]) => working_in.

put_employees([1@person]) => 3@place.

put_works([3@place]) => 1@person.
```

Since all three methods have the same consequence, any rule attached to any of these methods (e.g. *put_works*) should be fired if any of the other methods is invoked (i.e. *new*, *put_employees*). Of course, three different rules could be defined for each of the possible triggering methods but this could place an extra burden in the rule designer who would have to be aware of when an attribute represents a relationship view. To hide this from the user, rules could be defined for *each* relationship so that before and after a new relationship instance is created, the system also signals that an insertion has occurred in each of the attribute-based views. In this way an improvement in transparency is achieved: the rule designer ignores whether an attribute represents a relationship view or not [2]. Since such rules have to be defined for each relationship, this behaviour can be considered as part of the relationship definition and thus moved to the relationship metaclass. Four rules then, are needed in total. One of these metarules is shown in figure 7.7. This rule is fired *before* a *new* relationship instance is created regardless of the relationship class to which it will belong. The condition checks whether the current relationship class has a view and if so, signals the corresponding event for this view where the *active_method* is the *'put_*view' method (*View1Put*), the current active object is the object playing the role indicated by the view (*Role1Val*) and the argument of the method is the object playing the other role in the relationship (*Role2Val*). The four rules are quite similar, signalling the before and after events for the two relationship views. In [Diaz 91b] another example of a metarule can be found which handles derived classes.

[2] Such transparent support becomes even more necessary if relationship hierarchies are considered. Here, the creation of a relationship instance must to signal not only the insertion of the attribute-based views of its own relationship class but also the insertion of the attribute-based views of any of its superclasses!

7.6 Deriving rules for constraint maintenance

In ABEL a constraint language (described in chapter 5) is provided so that users can specify constraints as attribute facets instead of hard-coding into method definitions. In this section, the way in which the system generates a set of rules to enforce these declaratively-stated constraints is described. In this way, as well as easing the definition of constraints, constraint enforcement is moved to a different, rule-based mechanism, thus overcoming some of the problems pointed out in chapter 5.

In object-oriented systems, related work has been done in [Lafue 87, Caseau 89, Urban 90]. In [Lafue 87] constraints are stored together with the affected objects using demons. Although fast location of affected objects is achieved, constraint specification and enforcement are often still hard-wired into demons. In [Caseau 89] demons are obtained from logical rules. The stress of this paper is on improving the declarativeness of the system and its resolution power, while maintaining the efficiency. In [Urban 89], the constraint language ALICE is proposed. Instead of immediately translating constraints to specific actions, Urban obtains a set of equivalent Horn rules which "are analysed to help the designer understand how constraints affect operations on objects and to identify alternative propagation actions". This process, known as *constraint analysis*, is also used to generated a set of rules to maintain the constraint [Urban 90].

However, previous approaches do not fully address certain features of the object oriented paradigm, such as inheritance of constraints through specialisation hierarchies and the description of situation violating constraints not only by the name of the method but also with the name of the class. All these features are addressed here where constraint maintenance and enforcement are achieved by generating a set of rules from the constraint definitions.

Figure 7.8 shows a kind of 'rule template' at the beginning of the generation process. The attributes *when* and *is_it_enabled* have default values, and the system has to obtain values for the *active_method, active_class, condition* and *action* attributes of the rule. In the following subsections, the way in which these values are generated is discussed in detail. Before continuing, the reader is strongly advised to read the first-order logic counterpart of CEs described in section 5.4.

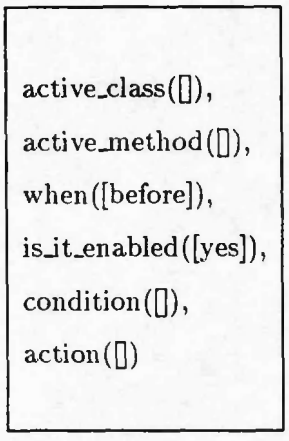


Figure 7.8: A rule template at the beginning of the derivation rule process.

P	Q	P $\longrightarrow$ Q
T	T	T
T	F	F
F	T	T
F	F	T

Figure 7.9: Truth table for the implication operator

7.6.1 Active_method and active_class value generation.

The **active_method** attribute contains the name of the method firing the rule. To obtain these methods the approach described in [Nicolas 78] from deductive databases is taken. This approach is based on the Horn logic counterpart of CEs. The truth table for the implication operator is shown in figure 7.9. Several remarks can be pointed out from the implication truth table:

1. If Q is false, P must also be false to make the implication true. Delete operations on predicates appearing in the head of the clause can falsify the head. To keep the validity of the implication, delete operations can be necessary to falsify the body of the clause.

2. If P is true, Q must also be true to satisfy the implication. Insert operations on predicates appearing in the body of the clause can satisfy the body. Therefore, insertions may be needed in the head of the clause to make it true, so that the entire clause can be satisfied.

3. If P is false, the implication is evaluated to true regardless of what Q evaluates to. The clause is *trivially satisfied* [Urban 91]. This means that delete operations on the body of the clause do not need any compensating action to make the implication true.

4. If Q is true, the implication is evaluated to true regardless of what P evaluates to. Therefore, compensating actions are not required when insert operations are made on the head of the clause.

So the approach in [Nicolas 78] is used not only as a way of detecting the operations that can violate a constraint but also as a mean of indicating the changes to be made to satisfy it.

As an example, consider a constraint similar to the one proposed in [Morgenstern 84] which enforces that the *projects* which a *lecturer* has a responsibility for, are to be the same as the set of *projects* his/her *research_assistants* work on. This can be expressed as the CE:

```
projects wof lecturer  ::  projects wof research_assistants of lecturer
```

Translating this constraint into first-order logic, one of the Horn rules is the following:

$$\text{lecturer(M)} \wedge \text{research\_assistants(M,S)} \wedge \text{projects(S,P)} \rightarrow \text{projects(M,P)}$$

from which several conclusions can be drawn. When a *research_assistant* is inserted, the body of the above clause can become true, forcing the head of the clause to be true also. On the other hand, the head of the clause can become false when a *project* of a *lecturer* is deleted. To make the whole implication true, deletion of the *project* from this *lecturer's research_assistants* or deletion of the *research_assistant* itself may be necessary.

Since the path of a CE is translated into a conjunction of existentially qualified predicates, any path's link appears in both the left and right side of a Horn rule. Hence, insertions and deletions on any relationship occurring in the description of a CE, have to be checked to verify that the constraint will be satisfied after the update. Therefore, two events are considered for any of these relationships, triggered by *put* and *delete* methods, respectively. However, links in the constraint scope definition, appear only on the left hand side of Horn rules and then only *put* events are considered. As an example, the

```
event(before,put_projects,lecturer)
event(before,delete_projects,lecturer)
event(before,put_research_assistants,lecturer)
event(before,delete_research_assistants,lecturer)
event(before,put_projects,person)
event(before,delete_projects,person)
```

Figure 7.10: Set of *before events* generated for the projects-of-lecturer constraint.

project-of-lecturer giving above constraint generates the set of before events shown in figure 7.10.

It is worth noticing that since links mainly represent relationships and an attribute-based view of relationships is provided, methods affecting CEs are not only those obtained from the relationships directly involved in the CE but also those methods which modify the value of their inverse relationships. For instance, the above constraint referring to *research_assistants*, can be modified either directly by methods updating the relationship *research_assistants* or indirectly by methods updating its inverse (i.e *supervised_by*), e.g. the *put_supervised_by* method. However, in our opinion, the constraint mechanism should not be in charge of this process, i.e. generation of the appropriate inverse events. It is the relationship mechanism that has to propagate changes to a relationship to its inverse. In this way, an improvement in transparency is achieved: the constraint mechanism is unaware of whether a change has occurred directly in a relationship or in its inverse. In a previous section, a metarule to achieve such behaviour has been given.

The **active_method**, is obtained from the name of the relationship by adding the prefix *'put_'* or *'delete_'*. The **active_class** attribute is obtained by the system from the CE anchor and the domain of the relationship view involved. These two attributes together with the *when* attribute define, the event that causes a rule to be fired.

7.6.2 Rule condition value generation.

The condition is a set of queries to check that the state of the database is appropriate for action execution. A single rule could be defined to be invoked when any of the above events arise. In this case, the condition would check that the constraint was still valid for all the objects involved. However, this could result in evaluation of the rule's condition over a large number of objects. An improvement can be achieved if the condition is checked only for those changes caused by the current update and assuming that the DB was in a valid state before the update. Hence, if, for example, a *project* is added to an *research_assistant* in the *project-of-lecturer* constraint, the condition and action of the rules can be restricted to this update instead of checking the constraint for all the *projects* of all the *research_assistants* of the *lecturer*. Therefore, a rule will be defined for each possible event that can violate the constraint so that the condition and the action attribute of the rule will consider only the changes produced by this event. Since rules are indexed by class, an increase in the number of rules does not lead to a significant system slow down. In our proposal, a rule-based version of the algorithmic solution for CE maintenance proposed in [Morgenstern 84] is adapted to an OO environment.

Rule conditions are used to check whether a constraint will be violated by a given update. If the condition is satisfied the action is executed either to put the DB into a valid state or to reject the update.

Unlike in a deductive approach where relationships can be represented as predicates whose arguments stand for the entities related, in the object oriented paradigm an attribute-based view of relationships is more popular. Whereas a predicate view provides a *non*-directional representation (i.e. the same predicate is used regardless of the argument being instantiated), in an object oriented system relationships are seen as pointers to the related objects. In this case, the method invoked depends on the direction in which the relationship is traversed. As a result, the constraint, declaratively stated, by the user, cannot be directly translated to the rule's condition as in [Urban 90], but it is necessary to consider the direction in which the constraint is checked. For example, in the *projects-of-lecturer* constraint the *research_assistants* link is transformed to the *supervised_by* link (i.e. its inverse) if the constraint is traversed backwards.

The process of defining a rule's condition can be better understood if paths are seen as

function compositions. In this way, a constraint such as:

$$r_m \text{ of } r_{m-1} \text{ of } \ldots\ r_2 \text{ of } r_1 \text{ of anchor} :: l_n \text{ of } l_{n-1} \text{ of } \ldots\ l_2 \text{ of } l_1 \text{ of anchor}$$

can be seen as a comparison of functions S and T:

$$\underbrace{r_m \circ r_{m-1} \circ \cdots r_2 \circ r_1}_{S}(anchor\_instance) = \underbrace{l_n \circ l_{n-1} \circ \cdots l_2 \circ l_1}_{T}(anchor\_instance)$$

When an insertion occurs, for instance the object O_1 is inserted [3] as an r_i of the object O_2 the equality between functions S and T has to be maintained. Since r_i is the modified link, the condition takes into account only the change produced in r_i. Let LS be the set of r_m*-range objects* that have been affected by the update, which can be worked out as:

$$LS = r_m \circ r_{m-1} \circ \cdots r_{i+2} \circ r_{i+1}(O_1)$$

Let AS be the set of *anchor objects* that have been affected by the update, which can be obtained as:

$$AS = r_1^{-1} \circ r_2^{-1} \circ \cdots r_{i-2}^{-1} \circ r_{i-1}^{-1}(O_2)$$

where r_i^{-1} stands for the inverse of r_i.

As a result of the above insertion, function S has been enlarged with the set of pairs obtained from the Cartesian product of AS and LS, except those already in S [4]:

$$\triangle S = (AS \otimes LS) - S$$

The rule's condition has to take into account the following cases:

1. **if** LS = $\emptyset$ or AS = $\emptyset$ **then** the constraint is trivially satisfied
2. **if** $\triangle S = \emptyset$ **then** the constraint is satisfied
3. **if** $\triangle S \neq \emptyset$ **then** the constraint is violated

In cases 1 and 2, the condition is evaluated to false and the rule does not fire. In the third case the action is executed since the constraint has been violated, i.e. the

[3] The message $put\_r_i([O_1]) => O_2$ would achieve this insertion in ADAM.

[4] Once the update has been made, the original value of S can be obtained from T.

equality is no longer valid. In figure 7.11 the rule generated for an insertion on the *projects* relationship in the *projects-of-lecturer* constraint is shown. Functions AS and LS are obtained as compositions of methods after the specification of the constraint. In this case ($Anchor = InstanceOid$) and ($Leaf = NewValue$) represent such a composition for AS and LS respectively. Once these functions have been calculated, the rule's condition ascertains which of the above cases (1,2 or 3) has occurred. It is worth noticing that the set of pairs violating the constraint, i.e. $\triangle S$, is passed to the action part as the result of condition evaluation. In re-establishing the constraint, the action focuses on this set of pairs instead of considering the whole of S.

When a deletion occurs, e.g. the object O_1 is deleted as an r_i of the object O_2, function S is reduced to the set of pairs:

$$\triangledown S = \{AS \otimes LS\} - S'$$

where S' is the function S after the deletion has taken place. As before, three situations are possible namely, the constraint is trivially satisfied, the constraint is satisfied or the constraint is violated.

7.6.3 Rule action value generation.

A **rule action** is a set of operations directed towards constraint recovery. One of the strengths of CEs as originally proposed by [Morgenstern 84] is that they provide a way to declaratively state the changes to be done to re-establish a valid state of the DB. In [Morgenstern 84] the concept of a *weak bond* is proposed as the attribute "more readily modified in response to an initial change to the other side of the CE". In our example, if function S is enlarged with $\triangle S$ as a result of an insertion, function T must be extended with the same set of pairs if equality is preserved. $\triangle S$ can be seen as the set of pairs violating the constraint. The question is: which of the l_i functions have to be modified. The weak bond indicates such a function. If no weak bond is provided by the user, the action part rejects the update, displaying an error message. If a weak bond is available (e.g. l_w) the action has to find how to extend l_w so that the function composition T is enlarged and the constraint re-established.

For each anchor (e.g. a_k) in $\triangle$S, i.e.the *violating-constraint-pairs*, the set of pairs with which function l_w can be extended, is worked out in the following way:

$$l_w - range = \{rl_i \mid l_{w-1}^{-1} \circ l_{w-2}^{-1} \circ ... l_2^{-1} \circ l_1^{-1}(b_k) = rl_i \forall b_k \in \{(a_k, b_k) \in \triangle S\}\}$$

$$l_w - domain = \{dl_i \mid l_n \circ l_{n-1} \circ ... l_{w+2} \circ l_{w+1}(a_k) = dl_i \forall a_k \in \{(a_k, b_k) \in \triangle S\}\}$$

To avoid inserting the Cartesian product of l_w-range and l_w-domain, a *select* operator can be introduced to filter the pairs to be added to the DB, taking into account that for each *dl* an l_w link has to be established with at least a *rl* (and at most, if *lw* is single valued). If either l_w-range or l_w-domain are empty it means that some associations are not defined and the system will reject the update. In [Morgenstern 84], some kind of specification or user interaction is proposed to obtain these missing associations. This process must be repeated for each anchor in the violating constraint pairs.

In figure 7.11, a rule's action can be seen. The variable *ConstViolPairs* represents $\triangle$S which is obtained after condition evaluation. The objects a_k and b_k are represented by the variable *AAnchor* and *ALeaf*, respectively. To obtain l_w_range and l_w_domain the sequence of methods $(get\_research\_assistants(R_l) => AAnchor)$ and $(D_l = ALeaf)$ are executed, respectively. Different values of R_l and D_l are obtained by backtracking. For each pair (R_l, D_l) the system inserts a *project* since this link has been marked as the weak bond.

As far as deletion is concerned, the rule's action will reduce T by the set of pairs $\triangledown S$. This is accomplished by reducing the weak bond function. The l_w-range and l_w-domain can be worked out as presented above. For each pair, the l_w relationship is removed

7.7 Conclusion

Unlike current DBs, active DBs aim to provide automatic answers to events generated internally or externally to the system itself. System responses are expressed declaratively through event-condition-action rules. The research presented here is an attempt to provide an insight into rules in an OO context, stressing uniformity.

Uniformity stems from seeing rules as 'first-class' objects described by attributes and methods. In this way, rule management operations are conceived and implemented as methods. This brings all the advantages of the OO paradigm into rule management allowing rule management operations (e.g. *fire, signal*) to be specialised to cope with

```
new([RuleOid,[
    active_class([lecturer]),
    active_method([put_projects]),
    when([before]),
    is_it_enabled([yes]),
    condition([(
        condition_result(ConstViolPairs),
        current_object(InstanceOid), %This is O2
        current_arguments([NewValue]), %This is O1
        findall(Anchor,(Anchor = InstanceOid),Anchors),
        findall(Leaf,(Leaf = NewValue),Leaves),
        .....
        ((Anchors = [] ; Leaves = [])
            -> % first case: Constraint trivially satisfied
               fail
            ;  findall([AAnchor,ALeaf]),
                 (member(AAnchor,Anchors),
                   member(ALeaf,Leaves),
                   \+ (get_research_assistants(Emp) => AAnchor,
                         get_projects(ALeaf) => Emp))
                 ),
               ConstViolPairs),
               (ConstViolPairs = []
                 -> fail % second case
                 ;  true % third case
               )
        )
        .....
    )]),
    action([(
        condition_result(ConstViolPairs),
        current_object(InstanceOid),
        .....
        obtaining_anchors(ConstViolPairs,Anchors),
        (member(AAnchor,Anchors),
          member([AAnchor,ALeaf]),ConstViolPairs),
          ((get_research_assistants(R1) => AAnchor),
          (P1 = ALeaf),
           % adding P1 as a project of R1 if required. Project is the weak bond
           % the variable Result reflects the result of this addition
            .....
        )),
        Result
        .....
    )]),
    declarative_condition([projects wof lecturer]),
    declarative_action([projects wof research_assistants of lecturer])
]]) => integrity_rule.
```

Figure 7.11: A constraint maintenance rule

special requirements. As a result, rules can be related to other objects or arranged in hierarchies, and rules can even be defined which are triggered by methods attached to rules themselves. Treating rules as objects also has the advantage that any new facility introduced for objects is automatically applicable to rules.

However rules are not an end on themselves. Rule conditions and actions can be difficult to state for non-experienced users. Therefore, higher level constructs can be provided whose semantics are declaratively specified and preserved by system-generated rules. This approach has been used in this thesis to enforce integrity constraints and the operational semantics of relationships.

Although it has not been the main concern of this work, efficiency plays a decisive role in active DBs. Several benchmarks have been performed to measure the overhead imposed by the rule management system. The results show that the introduction of rules makes programs on average about *twice* as slow as they are when the rule mechanism is disabled. Such a slow-down is predictable, as rule evaluation imposes an overhead on every possible event that can be detected by the system. However, the *scale up factor* (i.e. how the number of rules affects system performance) has been kept *low* by indexing rules by class. In this way the search for applicable rules is considerably reduced.

The principal contribution of the work presented in this chapter lies on providing some insights into rule management in OODBs through the extension of ADAM with active capabilities. Unlike the work presented in [Bauzer 90, Kotz 88] no additional structures have been defined. Every feature is catered for by enlarging object description with attributes and/or methods. Our work then, is more in tune with that presented in [Chakravarthy 89, Dayal 88] as part of the HiPAC project. However, whereas in HiPAC the underlying DB is relational, we have considered the idiosyncrasies of OODBs by using ADAM.

Chapter 8

Conclusions

8.1 Making object-oriented databases more knowledgeable

The next generation of DBs will support *knowledge independence*, i.e. knowledge will be removed from the procedural setting in which it is embedded and described in a declarative format within the DB. Several advantages can be arise from this approach:

- *Understandability.* Since the knowledge is in a declarative format, it is easier to capture what its subject matter is. Otherwise, implementation details could obscure its real meaning.
- *Maintainability.* Knowledge can be easily modified if it is explicitly stated, without having to tackle procedural side-effects.
- *Reusability.* Removed from a particular application, knowledge can be used for many purposes, even for some not foreseen at the beginning.

OODBs are seen as a promising direction in DB evolution. Two features are considered to be the principal achievements of this approach. First, being object-oriented, a direct mapping between entities in the real world and the constructs of the DB is supported, thus preventing entities from being split. Second, the features modelled refer not only to the structural aspects of the UoD but also to the behavioural ones. An entity is described by its attributive structural and relational characteristics as well as by its behaviour, described by methods. In this way, OODBs attempt to cope with the impedance mismatch problem. Nevertheless, methods can jeopardize knowledge independence by

encapsulating not only the implementation of the operation to which it refers but also other kinds of knowledge.

Our aim has been to extend an OODB, ADAM, with a set of constructs that allow the explicit representation of knowledge that was previously hidden in method implementations. These constructs allow integrity constraints, user-defined relationships and active behaviour to be represented explicitly.

Integrity constraints in ABEL are explicitly stated as *facets* of the attributes. A constraint equation (CE) approach has been chosen, because of to its path-like syntax which is more familiar to the user, who does not have to express constraints in a different paradigm such as first order logic. In this way, the responsibility for constraint enforcement is removed from method definition, allowing methods to be overridden without jeopardising constraint maintenance. From the point of view of the user, this approach also relieves him/her from considering constraint maintainability and constraint interaction. Furthermore, system capabilities can be enhanced using the constraints explicitly stated for other purposes (e.g. semantic query optimisation).

The main contributions of our approach can be summarised as follows:

- Although CEs have already been proposed for relational DBs [Morgenstern 84], moving then to an object-oriented framework poses new challenges. Constraints no longer refer to attributes in flat relations but to classes arranged in hierarchies. This raises the problem of constraint inheritance which has been tackled by moving constraints out of method definitions.

- A rule-based mechanism is proposed for constraint maintenance. A set of rules are automatically generated by the system from the declarative specification of the constraint. Such a set is obtained from the first order logic counterpart of CEs.

User-defined relationships are seen as *first-class objects*: attributes and methods can be used to describe relationships that can be arranged in hierarchies. Furthermore, novel features have been considered such as the description of the operational semantics of the relationship and the introduction of 'inferred' constraints and/or attributes of the objects among which the relationship is established. The operational semantics allow messages to be *propagated* and *delegated* from one participant to the other. In this way, sharing

through other than the *is_a* and *instance* links is considered. This is a quite new feature in OODBs that requires further research. It is our opinion that seeing relationships as first-class objects provides a new way of conceptualising the application, encouraging a more relational view of the UoD.

Three points can be seen as the main contributions of this work:

- Although we do not claim to have the novelty of defining relationships as objects, it is less common, in the DB literature, to find work investigating the 'inferred' properties of relationships. In this chapter some insights into inferred attributes and constraints have been presented and implemented.
- The idea of the operational semantics for relationships, originally proposed for programming languages in [Rumbaugh 87], has been extended to consider both propagation and delegation effects in the context of OODBs. Furthermore, unlike the approach in [Rumbaugh 87], a rule-based mechanism has been used for supporting this semantics.
- Like objects, relationships can be arranged in hierarchies. Here an approach has been presented and implemented for relationship specialisation.

Event Condition Action Rules (ECA rules) provide an active dimension to DBMSs. The system is no longer a passive repository of data but can undertake its own actions autonomously. Because they are supported in an object-oriented manner, rules are manipulated in the same way as any other object in the system: rules can be queried, arranged in hierarchies, related to other objects and so on. But what is more important, the rule manager itself can be specialised to meet new requirements by using the same OO philosophy: introducing and/or specialising either classes, methods or attributes. Full advantage of this feature has been taken during the implementation of the system.

Several examples have been given of the direct use of rules. Nevertheless, higher level tools can be provided of which rules are simply a vehicle for implementation. Integrity constraints and the operational semantics of relationships have been supported in this way: rules are generated by the system from some declarative specification. However, some efficiency penalty has to be paid. The introduction of rules make programs on average about *twice* as slow as they are when the rule mechanism is disabled. However,

the scale-up factor (i.e. how the number of rules affects system performance) has been kept *low* by indexing rules by class.

The principal contribution of this work lies in providing some insights into rule management in OODBs through the extension of ADAM with active capabilities. Unlike the work presented in [Bauzer 90, Kotz 88] no additional structures have been defined. Every requirement is accomplished by enlarging object description with attributes and/or methods. Our work then, is more in tune with that presented in [Chakravarthy 89, Dayal 88] as part of the HiPAC project. However, whereas in HiPAC the underlying DB is relational, we have considered the idiosyncrasies of OODBs by using ADAM.

During the conceptualisation and implementation of the whole system we have kept in mind two principles: *uniformity* and *declarativeness*. Both help to enhance the understandability, maintainability and reusability of the system which had previously been jeopardised by embedding more information into some method definitions than just method implementation. As pointed out in [Khoshifian 90] "... in the evolution of databases and data models, OODBs form an important and necessary phase. The future of DBs, however, is with *intelligent databases*". In our opinion, ABEL is a contribution towards this end.

8.2 System implementation

One of the main contributions of this thesis is the illustration of the use of metaclasses for extending the system along the lines originally established in [Paton 89a]. Paton's OODB, ADAM, has shown itself to be a reliable, sound and flexible system for implementing our ideas. The availability of metaclasses in ADAM has proved to be a *revolutionary* mechanism. Since data and metadata are both represented as objects, only one mechanism is required for browising, querying, relating and specialising objects. As a result, the distinction between data and metadata is removed and it becomes a question of the level of abstraction at which one is working. Besides uniformity, two main advantages can be seen from this approach: *accessibility* and *extensibility*. The former allows users to query, update and delete metadata dynamically like any other data in the system. Extensibility stems from using objects to define the DBMS which can be enlarged and specialised using the well-known subclass mechanism. In order to cope

with new features, ADAM can be extended along two dimensions: the inheritance hierarchy where the description of objects as instances is considered, and the instantiation hierarchy where the description of objects as classes is taken into account. Both kind of extensions have been required to provide the new features, and their implementation has been shown throughout the thesis. In our opinion, metaclasses are a breakthrough in system design, whose importance has not yet been stressed enough.

ABEL has been implemented in QUINTUS Prolog (Version 2.5) running under SUN. The system can be executed by invoking 'abel', an executable file containing a Prolog save state and obtained once the system has been booted. The system is availabled for research from

- Computing Science Department, University of Aberdeen, U.K.
- Facultad de Informatica, Universidad del Pais Vasco, Spain

provided permission has been granted for using ADAM (contact: Norman Paton, Heriot-Watt University, Edinburgh).

Finally, the following features of Prolog have been particularly useful throughout the implementation:

- Meta-level programming. The ability to create data structures that are subsequently treated as Prolog programs has been widely used during rule generation.
- Homogeneous treatment of procedures and data. This proved to be especially useful in ECA rules.
- The Prolog unification mechanism has been advantageous, specially when implementing delegation sharing for the operational semantics of relationships.
- The flexibility of the structure inspection mechanism through pattern matching.

8.3 Future directions

8.3.1 Integrity constraints

To cope with the increase in complexity of the domains tackled by DBs, more powerful integrity constraint mechanisms are required. This concerns not only the availability of

a constraint definition language but also a new range of issues such as constraint exceptions or the definition of a rule language to specify recovery actions if constraints are violated, in the way initiated by Morgenstern's weak bond. Such issues have become particularly important in CAD/CAM system [Buchmann 86].

In the current implementation, constraint equations have been defined to check set equality. A richer constraint language can be provided in the way suggested by Morgenstern [Morgenstern 84] where more complex set operations would be available.

Besides, as the number of constraints increases and they become more complex, it is harder for the designer to foresee constraint interaction and possible side-effects. A constraint interaction tool like the one described by Urban [Urban 90] could be very useful.

Finally, since constraints are defined explicitly, they can be used for purposes other than constraint maintenance. We envisage further use of constraints for semantic query optimisation [Chakravarthy 90, Demolombe 90] and intensional query processing [Motro 89].

8.3.2 User-defined relationships

Enhancing the semantics of relationships should be further investigated. For example, the system can be extended to consider the mathematical properties of relationships either in a simple way (e.g. the transitivity, symmetry and reflexivity of the relationship) as presented in [Escamilla 90] or with a more sophisticated language such as the one found in CRL [Carnegie 85]. N-ary relationships can also be addressed within this new framework. Some applications should be developed to test the usefulness of the constructs already provided.

Also, derived classes are a useful mechanism as proved by the SDM data model. Although a rudimentary mechanism has been sketched in [Diaz 91b], a more sound and richer implementation is required.

8.3.3 Rule management

At this stage, rules are fired in sequence from the more general to the more specific. This follows the heuristic that rules have to be fired in an adequate context, i.e. more specific

rules wait till more general rules have fired to provide the right framework. Although this is valid for integrity constraints, other kinds of control can be required. A finer granularity mechanism is needed.

As the number of rules increases, optimisation and control of rules become fundamental issues. Work already done in relational and deductive DBs can help here. A rule language that allows conditions and actions of rules to be described in a more declarative fashion, would facilitate not only the specification of rules but also would provide the basis for reasoning about rules' conditions and actions. Such reasoning could check, for instance, the similarity between rule's conditions to avoid re-evaluation (in the way of the RETE mechanism [Forgy 81]). Also some processes can be made to detect when a rule's action is affected by another rule's condition so that a network of rules can be supported to enhance system performance.

Some gains can also be achieved by providing more sophisticated rule indexes other than by class as in the current implementation. A quicker event detector would also enhance the efficiency of the rule manager.

8.4 Can we claim to have made ADAM more knowledgeable?

The answer is '**YES,BUT...**'. Such a cautious answer stems from the different views that DBs and AI practitioners have about what knowledge really means.

Our '**YES**' part of the answer comes from the DB side. In this area the view is:

> "... that databases represent sets of definite atomic statements (facts) and that knowledge bases may, in addition, represent sets of general statements and conditional statements (rules)..." [Bubenko 89]

From this perspective, ADAM has definitively been made more knowledgeable, increasing the set of general statements (i.e. universally quantified) kept in the DB. Integrity constraints, relationship semantics and active behaviour were previously embedded in method implementations and can now be stated explicitly.

BUT... the AI view of knowledge is quite different. Knowledge in AI means:

> "... a collection of data structures and interpretive procedures ... which together produce an intelligent behaviour in some sense. An instance of a data structure is not seen as knowledge but only as a *source of knowledge.* Without procedures which can interpret and use it, data is nothing more than an arbitrary collection of (electronic-magnetic) marks." [Bubenko 89]

This view is also supported by M.L. Brodie and J. Mylopoulos when they state:

> "... any knowledge representation language must be provided with a (rich) semantic theory for relating an information base to its subject matter, while a data model requires an (effective) computational theory for realising information bases on physical machines." [Brodie 86b]

The important point is that knowledge can be *interpreted* based on a *semantic theory.* And this is what is generally missing in OODB. As an example, classes in OODB are characterised by the set of message they can respond to. Such a set of messages refers not only to the methods explicitly stored with the class but to the ones that are inherited from its superclasses as well. The problem stems from the fact that inheritance is an operational notion where, for example, the order in which superclasses are specified is important in deciding the method to be inherited. As pointed out by Patel-Schneider:

> "... the meaning of a class can be determined only by a very close examination of the operational details of the inheritance mechanism, the data structures in the class that control inheritance, and the data structures in more-specific classes. If classes were governed by a separate, representational definition, as in knowledge representation systems, then these operational details would not be important, and would not have to be investigated" [Patel-Schneider 90]

This problem worsens if, instead of viewing inheritance as a technique for specialisation, it is seen as an implementation technique where reusability is maximised by allowing operations to be excluded from subclasses [Snyder 86].

Some systems are being developed based on these premises by using a terminological approach, e.g. CLASSIC [Borgida 90]. But no solution is thoroughly satisfactory. These systems encompass only structure (not behaviour) and have to limit their expressiveness to avoid becoming intractable.

Unfortunately, a general solution might not be possible [Mylopoulos 90] and in the near future a range of different systems with distinct capabilities may appear. After all, file

systems have not completely died out with the arrival of DBMSs, being still alive in transaction systems. As concluded by Mylopoulos:

> "... greater modelling power (to capture knowledge in arbitrary 'real world' applications) and better programming facilities (through the integration of database management and programming language features) are two largely exclusive sets of requirements that will eventually lead to two different types of object-oriented notations and supporting systems." [Mylopoulos 90]

In our opinion, the extensions proposed in this work reach a good balance between enhancing modelling capabilities and programming facilities.

Bibliography

[Abardanel 87] R.M. Abarbanel, M.D. Willians *A Relational Representation for Knowledge Bases*, in [Kerschberg 87], pp. 191-206

[Abiteboul 87] S. Abitebout, R. Hull *IFO: A Formal Semantic Database Model*, in ACM Transactions on Database Systems, 12(4), pp. 525-565

[Agha 86] G. Agha *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA

[Albano 89] A. Albano *Conceptual Languages: A Comparison of ADAPLEX, Galileo and Taxis*, in [Schmidt 89], pp. 395-408

[Almorade 89] J. Almorade *Rule-Based Delegation for Prototypes*, OOPSLA'89, pp. 363-370

[America 91] P. America *A Behavioural Approach to Subtyping in Object-Oriented Programming Languages*, in [Lenzerini 91], pp. 173-190

[Amy 89] I. Amy Chen, D. McLeod *Derived Data Update in Semantic Databases*, in 15th Intl. Conf on Very Large Data Bases, pp.225-2235

[Attardi 89] G. Attardi, C. Bonini, M. Boscoterecase, T. Flagella, M. Gaspari *Metalevel Programming in CLOS*, in Proc. ECOOP, Cook (Ed.), pp. 243-256

[Bancilhon 86] F. Bancilhon, R. Ramakrishnan *An amateur's introduction to recursive query-processing strategies*, in ACM SIGMOD Intl. Conf. on Management of Data, pp. 16-52

[Bancilhon 88] F. Bancilhon *Object-Oriented Database Systems*, in Proc. 7th ACM SIGART-SIGMOD-SIGACT Symp. Principles of Database Systems

[Bauzer 90] C. Bauzer Medeiros, P. Pfeffer *A mechanism for Managing Rules in an Object-oriented Database*, Altair Technical Report

[Beech 90] D.Beech, P. Bernstein, M. Brodie, M. Carey, B. Lindsay, L. Rowe, M. Stonebraker *Third-generation data base system manifesto*, in [Meersman 91], pp. 495-511

[Bernstein 80] P. Bernstein, B. Blaustein, E. Clarke *Fast Maintenance of Semantic Integrity Assertions using Redundant Aggregate Data*, 6th Intl. Conf. on Very Large Data Bases, Montreal, pp. 126-131

[Bertino 91] E. Bertino, L. Martino *Object-Oriented Database Management Systems: Concepts and Issues*, IEEE Computer, April, pp. 33-47

[Bobrow 81] D.G. Bobrow, M. Stefik *The Loops manual*, Tech. Rep. KB-VLSI-81-13, Knowledge Systems Area, Xerox, Palo Alto, Research Center

[Borgida 84] A. Borgida, J. Mylopoulos, H.K.T. Wong *Generalization/Specialization as a Basis for Software Specification*, in [Brodie 84a], pp. 87-114

[Borgida 90] A. Borgida, R.J. Brachman, D.L. McGuinness, L.A. Resnick *CLASSIC: A Structural Data Model for Objects*, ACM Sigmod Notice, pp. 58-67

[Brachman 83] R.J. Brachman, R.E. Fickes, H.J. Levesque *KRYPTON: a functional approach to knowledge representation*, IEEE Computer, 16(10)

[Brachman 85] R.J. Brachman, J.G. Schmolze *An overview of the KL-ONE knowledge representation system*, Cognitive Science, 9, pp. 171-217

[Brachman 85] Brachman, R.J., *I lied about the trees*, The AI Magazine, pp. 80-93, Fall 1985

[Brodie 84a] *On Conceptual Modeling*, M.L. Brodie, J. Mylopoulos, J.W. Schimidt (Eds.) Springer-Verlag, 1984

[Brodie 84b] M.L. Brodie, D. Ridjanovic *On the Design and specification of Database Transactions*, on [Brodie 84a], pp. 277-306

[Brodie 86a] *On Knowldege Base Management Systems*, M.L. Brodie, J. Mylopoulos (Eds.) Springer-Verlag, 1986

[Brodie 86b] M.L. Brodie, J. Mylopoulos *Knowledge Bases versus Databases*, in [Brodie 86a], pp. 83-86

[Brodie 86c] M.L. Brodie *Database Management: A Survey*, in [Brodie 86a], pp. 201-218

[Bubenko 89] J.A. Bubenko, I.P. Orci *Knowledge Base Management Systems: A Database View*, in [Schmidt 89], pp. 373-378

[Buchmann 86] A.P. Buchmann, R.S. Carrera, M.A. Vazquez-Galindo *A Generalized Constraint and Exception Handler for an Object-Oriented CAD-DBMS*, in Proc. 1st Intl. Workshop on OODB Systems, K. Dittrich, U. Dayal (Eds.), pp. 38-49

[Buneman 86] P. Buneman, M.P Atkinson *Inheritance and Persistence in Database Programming Languages*, in Proc. ACM SIGMOD, pp. 4-15

[Cardelli 85] L. Cardelli, P. Wegner *On understanding types, data abstraction, and polymorphism*, Computing Surveys, 17(4), pp. 471-523

[Carey 88] M.J. Carey, D.J. Dewitt, S.L. Vandenberg *A Data Model and Query Language for EXODUS*, in Proc. ACM SIGMOD, pp. 413-423

[Carnegie 85] Knowledge Craft Manual Carnegie Group Inc., Pittsburgh, 1985

[Caseau 89] Y. Caseau *A Formal System for Producting Demons from Rules in an Object-Oriented Database*, in 1st Intl. Conf on Deductive and Object Oriented Databases, Kyoto, pp. 188-204

[Ceri 90] S. Ceri, J. Widom *Deriving Production Rules for Constraint Maintenance*, in 16th Intl. Conf. in Very Large Data Bases, Brisbane, pp. 567-577

[Chandrasekaran 83] S. Chandrasekran, S. Mittal *Conceptual Representation of Medical Knowledge for Diagnosis by Computer: MDX and Related Systems* Advances in Computers, M. Yovits (Ed.), pp. 217-292

[Chakravarthy 89] S. Chakravarthy *Rule Management and Evaluation: an Active DBMS Perspective* SIGMOD RECORD, 18(3), pp. 20-28

[Chakravarthy 90] V.S. Chakravarthy, J. Minker, J. Grant *Logic Based approach to Semantic Query Optimization*, ACM Transactions on Database Systems, 15(2), pp. 162-207

[Codd 79] E.F. Codd *Extending the database relational model to capture more meaning*, ACM TODS, 4(4), pp. 397-434

[Cointe 87] Cointe, P., *Metaclasses are First Class: the ObjVlisp Model*, in Proc. OOPSLA, pp. 156-167

[Copeland 84] G.P. Copeland, D. Maier *Making Smalltalk a database system*, in Proc. ACM SIGMOD, pp. 316-325

[Dahl 66] O.J. Dahl, K. Nygaard *SIMULA: An Algol-based simulation language*, Communications of the ACM, 9(9), pp. 671-678

[Dayal 88] U. Dayal, A.P. Buchmann, D.R. McCarthy *Rules Are objects Too: A Knowledge Model for An Active, Object Oriented Database System*, in Proc. 2nd Intl. Workshop on OODBS, K.R. Dittrich (Ed.), Spring-Verlag, pp. 129-143

[Dayal 89] U. Dayal *Active Database Management Systems*, SIGMOD RECORD, 18(3), pp. 150-169

[Demolombe 90] R. Demolombe, A. Illarramendi, J.M. Blanco *Semantic Optimization in Data Bases Using Artificial Intelligence Techniques*, Artificial Intelligence in databases and Information Systems (DS-3), IFIP, North-Holland pp. 519-529

[Deux 90] O. Deux *The Story of O_2*, IEEE Trans. Knowledge and Data Engineering, 2(1), pp. 91-106

[Diaz 90] O. Diaz, P.M.D. Gray *Semantic-rich User-defined Relationship as a Main Constructor in Object Oriented Databases*, in [Meersman 91], pp. 207-224

[Diaz 91a] O. Diaz, N.W. Paton *Sharing Behaviour in an Object Oriented Database using a rule-based mechanism* in Proc. 9th. British National Conference on Databases, BNCOD'91, Wolverhampton (United Kingdom), Butterworth Publishers, pp. 17-37

[Diaz 91b] O. Diaz, P.M.D. Gray, N.W. Paton *Rule Management in Object Oriented Databases: a uniform approach* in 17th Intl. Conf. on Very Large Data Base, Barcelona, pp. 317-326

[Dittrich 86] K.R. Dittrich *Object-oriented databases: the notion and the issues*, Proc. 1st Intl. Workshop on Object-Oriented Databases, 1986

[Ellis 89] C.A. Ellis, S.J. Gibbs *Active Objects: Realities and Possibilities*, in [Kim 89], pp. 561-572

[Escamilla 90] J. Escamilla, P. Jean *Relations Verticales et Horizontales dans un modele de representation de Connaissances*, in Nouvelles Perspectives des Systems d'Information, A. Flory, C. Rolland (Eds.) EYROLLES, pp. 153-185

[Eswaran 75] K.P. Eswaran, D.D. Chamberlin *Functional Specifications of a Subsystem for Database Integrity* in 1st Intl. Conf. on Very Large Data Bases, pp. 48-68

[Forgy 81] Forgy, C.L., *OPS5 User's Manual*, Report CMU-CS-81-135, Carnegie-Mellon University, 1981

[Fox 86] M.S. Fox, J.M. Wright, A. David *Experiences with SRL: an analysis of a frame-based knowledge representation system*, in [Kerschberg 86], pp. 161-172

[Fox 79] M.S. Fox *On Inheritance In Knowledge Representation*, in IJCAI'79, pp. 282-284

[Freundlich 90] Y. Freundlich *Knowledge Bases and Databases: Converging Technologies, Diverging Interests*, IEEE Computer, November, pp. 51-57

[Goldstein 77] I.P. Goldstein, R.B. Roberts *NUDGE, a knowledge-based scheduling program* in IJCAI-5, pp. 257-263

[Goldberg 83] A. Goldberg, D. Robson *Smalltalk-80: The Language and Its Implementation*, Reading, MA, Addison-Wesley

[Gray 88] P.M.D. Gray *Expert Systems and Object Oriented Databases: evolving a new Software Architecture*, in Research and Development in Expert Systems V, B. Kelly and A. Rector (Eds.), Cambridge University Press, pp. 284-295

[Gray 92] P.M.D. Gray, K.G. Kulkarni, N.W. Paton *Object Oriented Databases: A Semantic Data Model Approach*, Prentice-Hall, 1992

[Hammer 81] M. Hammer, D. McLeod *Database description with SDM: A Semantic Database Model*, ACM Transactions on Database Systems, 6(3), pp. 351-386

[Harris 86] D.R. Harris *A hybrid structured object and constraint representation language*, in AAAI, pp. 986-990

[Hewitt 77] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence, 8, pp. 323-364

[Hudson 89] S. Hudson, R. King *Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system*, in Proc. ACM SIGMOD, pp. 237-246

[Hull 87] R. Hull, R. King *Semantic Database Modelling: Survey, applications and research issues*, ACM Computing Surveys, 19(3), pp. 201-260

[Jagannathan 88] D. Jagannathan, R.L. Guck, B.L. Fritchman, F.P. Thompson, D.M. Tolbert *SIM: A database system based on the semantic data model*, in Proc. ACM SIGMOD, pp. 46-55

[Kerschberg 86] *Proc. 1st Conf. Expert Database System*, L. Kerschberg (Ed.), The Benjaming/Cummings Publishing Company, 1986

[Kerschberg 87] *Proc. 1st Intl. Conf. in Expert Database Systems*, L. Kerschberg (Ed.),The Benjaming/Cummings Publishing Company, 1987

[Kerschberg 89] *Proc. 2nd Intl. Conf. in Expert Database Systems*, L. Kerschberg (Ed.), The Benjaming/Cummings Publishing Company, 1989

[Khoshifian 86] S.N. Khoshifian, G.P. Copeland, *Object Identity*, in Proc. OOPSLA, pp. 406-416

[Khoshifian 90] S. Khoshafian, R. Abnous *Object Orientation: Concepts, Languages, Databases, User Interfaces*, Wiley, 1990

[Kim 89] *Object-Oriented Concepts, Databases and Applications*, W. Kim, F.H. Lochowsky (Eds.) ACM Press, 1989

[King 84] R. King, D. McLeod *A Unified Model and Methodology for Conceptual Database Design*, in [Brodie 84a], pp. 313-324

[Kingston 87] J. Kingston *Technical Overview of Knowledge Craft*, Airing, No. 3-6, AIAI University of Edinburgh, 1987

[Klas 89] W. Klas, E.J. Neuhold, M. Schrefl *Tailoring Object-Oriented Data Models through Metaclasses*, Proc. Advanced Database System Symposium, pp. 169-178

[Kotz 88] A.M. Kotz, K.R Dittrich, J.A. Mulle *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism* in Advance in Database Technology, EDBT, Venice, pp. 76-91

[Lafue 82] G.M.E. Lafue *Semantic Integrity Dependencies and Delayed Integrity Checking*, in 8th Intl. Conf. on Very Large Data Bases, Mexico, pp. 292-299

[Lafue 87] G.M.E. Lafue, R.G. Smith *Implementation of a semantic Integrity Manager with a Knowledge Representation System*, in [Kerschberg 87]

[Lambert 88] S. Lambert *Functional approaches to Knowledge Representation*, in [Ringland 88], pp. 207-222

[Lassez 87] C. Lassez *Constraint Logic Programming*, BYTE Magazine, pp. 171-176

[Lenzerini 91] *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, M. Lenzerini, D. Nardi, M. Simi (Eds.), Wiley, 1991

[Levesque 85] H.J. Levesque, R.J. Brachman *A Fundamental Tradeoff in Knowledge Representation and Reasoning*, in *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque (Eds.), Morgan Kaufmann, Los Altos, CA, 1985

[Lieberman 86] H. Lieberman, *Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems*, in Proc. OOPSLA, pp. 214-223

[Masini 89] G. Masini, A. Napoli, D. Colnet, D. Leonard K. Tombre *Les Languages a objects*, InterEditions, 1989

[McCarthy 89] D.R. McCarthy, U. Dayal *The Architecture Of An Active Data Base Management System*, Proc. Intl. Conf. on the Management of Data, Portland, pp. 215-224

[Meersman 91] *Object-Oriented Databases: Analysis, Design and Construction (DS-4)*, R.A. Meersman, W. Kent, S. Khosla (Eds.), IFIP, North-Holland, 1991

[Meyer 88] B. Meyer *Object-Oriented Software Construction*, Englewood Cliffs, NJ, Prentice-Hall

[Minsky 75] M. Minsky *A framework for representing knowledge*, in The Psychology of computer vision, P. Winston (Ed.), McGraw-Hill, 1975

[Morgenstern 84] M. Morgenstern *Constraint Equations: Declarative Expression of Constraints with Automatic Enforcement*, Proc. Intl. Conf. on Very Large Data Bases, pp. 153-299

[Morgenstern 89] M. Morgenstern, A. Borgida, C. Lassez, D. Maier, and G. Wiederhold *Constraint-Based Systems: Knowledge About Data*, in [Kerschberg 89], pp. 23-43

[Motro 89] A. Motro *Using Integrity Constraints to Provide Intensional Answers to Relational Queries*, in 15th Intl. Conf. on Very Large Data Bases, Amsterdam, pp. 237-249

[Mylopoulos 86] J. Mylopoulos, H.K.T. Wong *Some features of the TAXIS Data Model*, Proc. 6th Intl. Conf. on Very Large Data Bases, Montreal, Quebec, Canada, pp. 399-410

[Mylopoulos 89] *Artificial Intelligence and Data Bases* J.M. Mylopoulos, M. Brodie (Eds.) Morgan Kaufmann Publishers, 1989

[Mylopoulos 90] M. Mylopoulos *Object Orientation and Knowledge Base Management* in [Meersman 91], pp. 23-38

[Nassif 90] R. Nassif, Y. Qui, J. Zhu *Extending The Object-Oriented Paradigm to Support Relationships and Constraints* in [Meersman 91], pp. 305-330

[Nicolas 78] J.M. Nicolas, K. Yazdanian *Integrity Checking in Deductive Data Bases*, *Logic and Data Bases*, Gallaire and Minker (Eds.), pp. 325-346

[Nii 86] H.P. Nii *Blackboard Systems*, AI Magazine, 7(2), pp. 38-53, 7(3), pp. 82-106

[Oxborrow 91] E. Oxborrow, M. Davy, Z. Kemp, P. Linington, R. Thearle *Object-oriented data management in specialized environments* Information and Software Technology, 33(1), pp. 22-30

[Patel-Schneider 84] P.F. Patel-Schneider *Small can be beautiful in knowledge representation*, in Proc. IEEE Workshop on Principles of Knowledge-Based Systems, 1984

[Patel-Schneider 90] P.F. Patel-Schneider *Practical, Object-Oriented Knowledge Representation for Knowledge-Based Systems*, Information System, 15(1), pp. 9-19

[Patel-Schneider 91] P.F. Patel-Schneider *What's Inheritance got to do with Knowledge Representation*, in [Lenzerini 91], pp. 1-12

[Paton 89a] N.W. Paton *A Prolog Implementation of an Object-Oriented Database*, Ph.D. Dissertation, University of Aberdeen, Department of Computing Science, 1989

[Paton 89b] N.W. Paton *ADAM: An Object-Oriented Database System Implemented in Prolog*, Proc. 7th BNCOD, M.H. Williams (Ed.), Cambridge University Press, pp. 147-161

[Paton 90] N.W. Paton, O. Diaz *Metaclasses in Object-Oriented Databases*, in [Meersman 91], pp. 331-348

[Paton 91] N.W. Paton, O. Diaz *Object-Oriented Databases and Frame-Based Systems: Comparison*, Information and Software Technology, 33(5), pp. 357-365

[Peckham 88] J. Peckham, F. Maryanski *Semantic Data Models* Computing Surveys, 20(3), pp. 153-189

[Peltason 89] C. Peltason, A. Schmiedel, C. Kindermann, J. Quantza *The BACK System Revised*, KIT-report, Fachbereich Informatik, Technische Universitat Berlin, 1989

[Ringland 88] *Approaches to Knowledge Representation. An Introduction*, G.A. Ringland, D.A. Duce (Eds.) Research Studies Press, 1988

[Rumbaugh 87] J. Rumbaugh *Relations as semantic Constructors in an Object-Oriented Language*, OOPSLA, pp. 466-481

[Rumbaugh 88] J. Rumbaugh *Controlling Propagation of Operations using Attributes on Relations*, in Proc. OOPSLA'88, pp. 285-296

[Schaffert 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpot *An Introduction to Trellis/Owl*, in Proc. OOPSLA, pp. 9-16

[Schmidt 89] *Foundations of Knowledge Base Management*, J.W. Schmidt, C. Thanos (Eds.) Springer-Verlag, 1989

[Segler 91] T.M. Segler *ADAM/ABEL Conceptual Modelling* MSc. Project Report, Computing Science Department, University of Aberdeen, 1991

[Shepherd 84] A. Shepherd, L. Kerschberg *Constraint Management in Expert Database Systems*, in [Kerschberg 86], pp. 309-331

[Snyder 86] A. Snyder *Encapsulation and Inheritance in Object-Oriented Programming Languages*, OOPSLA'86, pp. 38-45

[Snyder 87] A. Snyder *Inheritance and the Development of Encapsulated Software Systems*, in Research Directions in Object-oriented Programming, Shriver and Wegner(Eds.), pp. 165-188

[Steele 80] G.L. Steel *The Definition and Implementation of a Computer Programming Language Based on Constraints*, MIT VLSI memo. 80-32, Ph.D. Dissertation, 1980

[Stefik 81] M. Stefik *Planning with constraints (MOLGEN: Part 1)*, Artificial Intelligence, 16, pp. 111-140

[Stefik 86] M. Stefik, D. G. Bobrow *Object-Oriented Programming: Themes and Variations*, The AI Magazine, 6(4), pp. 40-82

[Stein 87] L.A. Stein *Delegation is Inheritance*, OOPSLA'87, pp. 138-146

[Stein 89] L.A. Stein, H. Lieberman, D. Ungar *A Shared View of Sharing: The Treaty of Orlando*, in [Kim 89], pp. 31-48

[Stonebraker 90] M. Stonebraker, L.A. Rowe, M. Hiohama *The implementation of POSTGRES*, IEEE Transactions on Knowledge and Data Engineering, 2(1)

[Stonebraker 90] M. Stonebraker, A. Jhingram, J. Goh and S. Potamianos *On rules, procedures, caching and views in database systems*, in Proc. ACM SIGMOD, pp. 281-290

[Stroustrup 86] B. Stroustrup *An Overview of C++*, SIGPLAN Notices, 21(10), pp. 7-18

[Szolovits 86] P. Szolovits *Knowledge-Based Systems: A Survey*, in [Brodie 86a], pp. 339-352

[TRILOGY 89] TRILOGY user's manual. Complete Logic Systems Inc., North Vancouver B.C., Canada

[Ullman 88] J.D. Ullman *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988

[Urban 86] S.D. Urban, L.M.L. Delcambre *An Analysis of the Structural, Dynamic and Temporal Aspects of Semantic Data Models*, Intl. Conf. on Data and Engineering, pp. 382-389

[Urban 89] S.D. Urban *ALICE: An Assertion Language for Integrity Constraint Expression*, in Proc. 5th Intl. Conf. on Computer Software and Applications, Los Angeles

[Urban 90] S.D. Urban, M. Desiderio *Translating Constraints to Rules in CONTEXT: A CONstrainT EXplanation Tool*, in [Meersman 91], pp. 373-392

[Urban 91] S.D. Urban, L.M.L. Delcambre *Constraint Analysis: A Design Process for Specifying Operations on Objects*, Transactions on Knowledge and Data Engineering, 1991

[Wald 89] J.A. Wald *Implementing Constraints in a Knowledge Base*, in [Kerschberg 89], pp. 163-183

[Weinreb 81] D. Weinreb, D. Moon *Lisp Machine manual*, Symbolics Inc., 1981

[Willians 88] T. Willians, S. Lambert *Expressive Power and Computability*, in [Ringland 88], pp. 223-235

[Woods 75] W.A. Woods *What's in a link: foundations for semantic networks*, in *Representation and Understanding: Studies in Cognitive Science*, D.G. Bobrow, A.M. Collins (Eds.), Academic Press, NY, 1975

Appendix A

A BNF grammar for constraint equations in ABEL.

[1]

```
<constraint>            ::= <restriction>
                          | <restriction> where <scope>
                            {if restriction.class <> scope.class then error}

<restriction>           ::= <integer_restriction>
                          | <string_restriction>
                          | <set_restriction>

<integer_restriction>   ::= <path> <integer_comparison> <integer>

<string_restriction>    ::= <path> <string_comparison> <string>

<set_restriction>       ::= <path> <set_comparison> <right_path>
                            {if path.class <> right_path.class then error}

<right_path>            ::= <path>
                          | <path> <set_operator> <path>

<path>                  ::= <breakable_path>
                          | <unbreakable_path>

<unbreakable_path>      ::= <unbreakable_rest_path> of <class_name>

<unbreakable_rest_path> ::= <attribute_name>
                          | <unbreakable_rest_path> of <attribute_name>

<breakable_path>        ::= <unbreakable_rest_path> wof <class_name>
                          | <unbreakable_rest_path> wof <unbreakable_path>
```

[1]The set operator *union* is not yet implemented.

```
<integer_comparison>    ::= '=' | '>' | '<'

<string_comparison>     ::= '='

<set_comparison>        ::= '::'

<set_operator>          ::= 'union'

<scope>                 ::= <unbreakable_path> <integer_comparison> <integer>
                          | <unbreakable_path> <string_comparison> <string>
```

Appendix B

Using ABEL: An Example

B.1 Purpose

There is notihng that can be done in ABEL that cannot be done in ADAM, in the same way that nothing that can be done in PASCAL cannot also be done in ASSEMBLER. It is merely a question of the amount of effort spent in writing, reading and maintaining the code. However, providing primitives to capture the application domain explicitily has more substantial consequencies: namely, it enhances data reusability and changes the designer's way of thinking.

The latter refers to the way in which designers conceptualise the world. In this context, supporting relationships as *'first-class'* constructs changes the terms in which the designer envisages the world to be modelled. The designer thinks of relationships as proper entities rather than as simple attributes subordinate to objects. It can be said that *the tool changes the user.*

Moreover, removing knowledge from the procedural setting in which it is traditionally embedded, allows the same knowledge to be used for several purposes. The reuse of knowledge fits perfectly within the database philosophy, whose main postulate is *data independence*, i.e. data removed from applications. *Knowledge independence* is just another step in the same direction.

This has been the rationale for this work. This appendix is intended to illustrate this approach.

B.2 Example

Suppose we are interested in modelling the structural and behavioural features of people and companies. People can be described by the attributes *projects*, *salary* and *age* and have, as a behavioural feature, the ability to *be moved.* The *age* attribute is constrained to be less that 120. People can be married and thus become the husband or wife of their partners. The happiness of being married is enhanced by having a charming mother-in-law. Also we would like to model the idea that when a married person has to be moved so has his/her partner. Academics are a special kind of people who, besides sharing the features of a person can have a *degree.*

As far as companies are concerned, they are described by their *name* and *area* of business

```
c1:-
new([person,[
      attribute(attribute_tuple(projects,global,set,optional,string,[])),
      attribute(attribute_tuple(salary,global,single,optional,integer,[])),
      attribute(attribute_tuple(age,global,single,optional,integer,
        [age of person < 120])),

      method((moving(global,[],[],[],[]) :-
        message_recipient(Person),
        writeln(['The person ', Person, ' is moving .....'])
      ))
]]) => entity_class.

c2:-
new([academic,[
      is_a([person]),
      attribute(attribute_tuple(degree,global,single,optional,string,[])),
]]) => entity_class.
```

Figure B.1: The *person* and *academic* class definition in ABEL.

and their operations are: *moving* (used when a company has to be moved) and *stability* (used to work out the stability of the company from the area of business and an external factor called *tendency* which takes as values: *t1*, *t2* or *t3*). Enterprise and research institutes are two kinds of companies. Companies can have employees, i.e. people working in the company. If a company has to be moved all of its employees have to be moved as well, but not vice versa, the movement of a person does not imply the movement of his/her company. Moreover, the stability of a person comes from the stability of the company in which he/she works.

B.3 Implementation

From the above description it is clear that at least two entities are of interest: *person* and *company*. The definition of the class *person* in ABEL is shown in figure B.1. Structural and behavioural features are represented by attributes and methods respectively. Notice that the constraint on the *age* of person is explicitily represented as a facet. In ADAM such a restriction would have had to be encoded within the method *put_age* (see chapter 5). Although, in this example it is obvious how to extend the method to check the constraint, more difficult changes involving more than one method are required for more complex constraints. In ABEL, the user just specifies the constraint and leaves to the system the generation of the appropriate code for maintaining the constraint.

Less obvious is how to represent the relationships between people, and between a person and a company. For example, the *marriage* relationship established between two *persons*, can be represented as pointers within the participating objects (e.g. *husband_of* and *wife_of* attributes can be added to the *person* class). Where appropriate, these attributes hold the object identifier of the corresponding partner.

```
rel1:-
new([marriage,[
     related_class1([
         related_class_tuple(person,the_husband,husband_of,single,[])
     ]),
     related_class2([
         related_class_tuple(person,the_wife,wife_of,single,[])
     ]),
     operational_semantics([
          propagating moving   from the_husband to the_wife,
          propagating moving   from the_wife    to the_husband
     ]),
     attribute(
        attribute_tuple(wedding_date,global,single,optional,string,[])),
     attribute(
        attribute_tuple(husband_mother_in_law,the_husband,single,optional,string,[]
     attribute(
        attribute_tuple(wife_mother_in_law,the_wife,single,optional,string,[]))
]]) => relationship_class.
```

Figure B.2: The *marriage* relationship definition in ABEL.

However, if consistency is to be maintained, the methods handling these attributes (e.g. *put_husband_of, put_wife_of, delete_husband_of* and so on) have to consider more than just the insertion or deletion of the attribute. For instance, if *Alberto* is inserted as the *husband_of Ana*, the system should automatically consider *Ana* as the *wife_of Alberto*. Otherwise, we can have the situation where *Ana* is married to *Alberto* but *Alberto* is not married to *Ana*. To avoid such inconsistencies either the user has to remember to update both links, or the default methods have to be enlarged with extra code to update the inverse link. Such situations arise because the system does not know *what a relationship is*. From the system's point of view, it is indistinguishable whether an attribute represents a relationship or not and thus, it is up to the user to maintain the relationship's semantics. In ABEL the system is made aware of relationships by the provision of several primitives to represent their semantics (see chapter 6).

The definition of the *marriage* relationship is shown in figure B.2. As explained in section 6.4, the related class attributes indicate the classes participating in the relationship (e.g. *person*), the role played by each class (e.g. *the_husband, the_wife*), the cardinality (*single, set*) and an attribute-based view of the relationship (e.g. *husband_of, wife_of*) that allows the user to manipulate the relationship from the participant objects. Futhermore, new attributes of the participant classes can be defined as a result of the establishment of the relationships. *Mother in law* is a case in point. The class *person* does not have the attribute *mother in law* **until** people can get married, i.e. till the *marriage* relationship is defined. It would not have been possible to model this situation in ADAM.

The example of people-and-companies also requires the modelling of active behaviour by which the movement of a *person* implies the movement of his/her partner. Such a feature would have been implemented in ADAM by enlarging the *moving* method with

```
c3:-
new([company,[
      attribute(attribute_tuple(name,global,single,optional,string,[])),
      attribute(attribute_tuple(area,global,single,optional,string,[])),

      method((moving(global,[],[],[],[]) :-
        message_recipient(Company),
        writeln(['The company ', Company, ' is moving .....'])
      )),

      method((stability(global,[],[string],[],[Tendency]) :-
        message_recipient(Company),
        get_area(Area) => Company,
        member(Area-Tendency-Stability,[
          computing-t1-high,computing-t2-medium,computing-t3-low,
          insurance-t1-medium,insurance-t2-low,insurance-t3-high,
          banking-t1-low,banking-t2-high,banking-t3-medium]),
       writeln(['The stability is: ',Stability])
      ))
]]) => entity_class.

c4:-
new([enterprise,[
       is_a([company])
]]) => entity_class.

c5:-
new([research_institution,[
       is_a([company])
]]) => entity_class.
```

Figure B.3: The *company* class hierarchy definition in ABEL.

instructions for sending the message to the corresponding partner. Other relationships can later be introduced that requires similar modification in the code, but not always; when the *working_in* relationship is introduced relating *persons* and *companies* (where *companies* can also be moved), the method *moving* of *person* is not updated since the movement of *persons* does not lead the movement of their *companies*. Thus, it is the relationship rather than the method that decides whether methods are to be propagated or not.

The definition of the class *company* and its subclasses *enterprise* and *research_institution* is given in figure B.3. The *working_in* relationship is established between the *company* class and the *person* class, where the attribute-based view of the relationship is based on the *working_places* and *employees* 'attributes'. The operational semantics of *working_in* specifies that the movement of *the_company* implies the movement of *the_person*. Moreover, we would like to model the situation where the *stability* of a *person* comes from

the *stability* of the *company* in which he/she works i.e., people inherit their *stability* from their company.

Whereas the propagation behaviour could somehow be modelled in ADAM, delegation would be much more difficult. The question is *where* could this active behaviour be encoded in ADAM? Which method(s) have to be enlarged? The answer is not trivial since it is the absence of a method for obtaining the *stability* of a person that makes the system automatically refer the request to the person's company. In ABEL such active behaviour is declaratively stated in the operational semantics of relationships, from which the system generates a set of rules for supporting the right behaviour. The definition of *working_in* is shown in figure B.4.

Representing relationships as *'first-class'* objects allows the user to specialise relationship classes. Figure B.4 shows an example where the *working_in* relationship is specialised into *working_in_projects* and *working_in_research*. The former restricts the *companies* among which the relationship can be established to be *enterprise* (a subclass of *company*), whereas the latter specialises *companies* to be *research_institutions* (a subclass of *company*) and the *person* to be an *academic* (a subclass of *person*).

The metaclasses, classes and some ground instances involved in our example are shown in figure B.5.

The following section shows an ABEL session during which the previous example is run. For simplicity's sake, instead of typing the whole definition of the classes, we will refer to the constants assigned to the definitions in the previous figures (e.g. *c1* stands for the definition of the class *person* whereas *rel1* stands for the definition of the *marriage* relationship).

B.4 A session with ABEL

The following is the output obtained from a real session in which the previous examples were run. It has been obtained using the UNIX *script* command. Comments have been added to further explain the examples and these are indicated by a double asterisk.

```
Script started on Mon Jan 20 10:16:02 1992
hawk% abel

Quintus Prolog Release 2.5 (Sun-3, SunOS 4.0)
Copyright (C) 1990, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['Examples/relationships.pl'].
[consulting /home/eagle/diaz/New/Examples/relationships.pl...]
[relationships.pl consulted 0.500 sec 3,428 bytes]

yes

** CHECKING THE 'PERSON' CLASS HAS NOT BEEN CREATED

| ?- get(person) => entity_class.
```

```
rel2 :-
new([working_in,[
    related_class1([
       related_class_tuple(person,the_person,working_places,set,[])
    ]),
    related_class2([
       related_class_tuple(company,the_company,employees,set,[])
    ]),
    operational_semantics([
         propagating moving        from the_company to the_person,
         delegating  get_stability from the_person  to the_company
    ]),
    attribute(attribute_tuple(duration,global,single,optional,string,[]))
]]) => relationship_class.

rel3 :-
new([working_in_projects,[
    is_a([working_in]),
    related_class2([
       related_class_tuple(enterprise,the_company,workers,set,[])]),
    attribute(
       attribute_tuple(project_incomes,the_person,single,optional,integer,[]))
]]) => relationship_class.

rel4 :-
new([working_in_research,[
    is_a([working_in]),
    related_class1([
       related_class_tuple(academic,the_person,works,set,[])
    ]),
    related_class2([
       related_class_tuple(research_institution,the_company,researchers,set,[])
    ]),
    attribute(attribute_tuple(publications,the_person,set,optional,string,[])),
    attribute(attribute_tuple(grants,the_person,set,optional,integer,[]))
]]) => relationship_class.
```

Figure B.4: The *working_in* relationship hierarchy definition in ABEL.

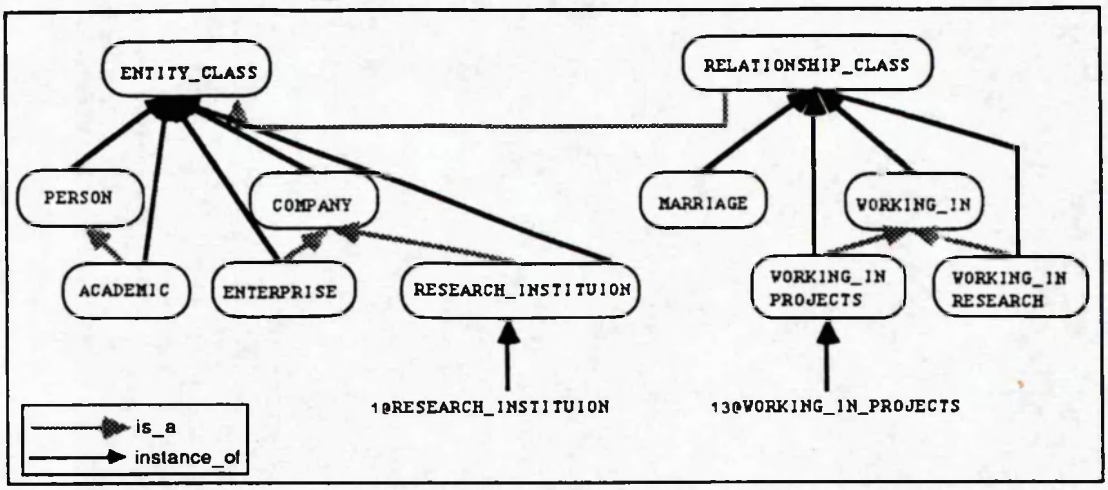


Figure B.5: Some of the objects involved in the *persons-companies* example.

```
no
| ?- get(X) => integrity_rule.

no

** CREATING THE 'PERSON' CLASS

| ?- c1.

yes

** CONSEQUENCIES OF CLASS CREATION
**    - 'PERSON' is made an instance of 'ENTITY_CLASS' metaclass
**    - a new integrity rule has been automatically generated by the system
**      to preserve the AGE constraint

| ?- get(person) => entity_class.

yes
| ?- get(X) => integrity_rule.

X = 1@integrity_rule ;

no
| ?- display => 1@integrity_rule.

Display of 1@integrity_rule
instance_of: integrity_rule
Slot values:
is_it_enable: yes
disable_for:
action:
```

```
  writeln([Integrity is violated by this update. Update rejected]),fail
condition:
  current_object(_7536),
  current_arguments([_7542]),
  findall(_7548,(_7548=_7536,true),_7550),
  remove_dups(_7550,_7562),
  findall(_7567,_7567=_7542,_7569),
  remove_dups(_7569,_7578),
  ((_7562=[];_7578=[])
      -> fail
      ;  findall(_7598,(member(_7598,_7578),\+_7598<120),_7600),
         !,
         (_7600=[] -> fail ; true)
  )
event: put_age
when: before
declarative_action: 120
declarative_condition: age of person
deactivated_for:

yes

** META-INFORMATION IS RETRIEVED REFERING TO STRUCTURAL AND
**  BEHAVIOURAL FEATURES.

| ?- get_attribute_desc(X) => person.

X = attribute_desc_tuple(projects,person,local,set,optional,string,[],[]) ;

X = attribute_desc_tuple(salary,person,local,single,optional,integer,[],[]) ;

X = attribute_desc_tuple(age,person,local,set,optional,integer,
         [age of person<120],[[1@integrity_rule]]) ;

no
| ?- get_method_desc(X) => person.

X = method_desc_tuple(object,display,global,[],[],[]) ;

X = method_desc_tuple(object,get_in_class,global,set,[],object) ;

X = method_desc_tuple(object,update_instance_of,global,[],[object,object],[]) ;

X = method_desc_tuple(object,delete_instance_of,global,[],[object],[])

** AN INSTANCE OF THE CLASS 'PERSON' IS CREATED

| ?- new([X,[
```

```
      salary([999]),
      projects([pr1,pr2])
     ]]) => person.

X = 0@person

** THE SYSTEM PREVENTS THE USER FROM INSERTING AN 'AGE' VALUE
**  ABOVE 120. THIS IS ACHIEVED BY FIRING THE '1@INTEGRITY_RULE'.

| ?- put_age([145]) => 0@person.
Integrity is violated by this update. Update rejected

no
| ?- put_age([33]) => 0@person.

yes

** CHECKING 'THE MARRIAGE' RELATIONSHIP HAS NOT BEEN CREATED

| ?- get(marriage) => relationship_class.

no
| ?- get(X) => propagating_rule.

no

** PEOPLE DO NOT UNDERSTAND THE MESSAGE 'GET_HUSBAND_MOTHER_IN_LAW'.
**  HENCE, THE SYSTEM DISPLAYS AN ERROR.

| ?- get_husband_mother_in_law(X) => 0@person.

ADAM Error in evaluate_message
Unable to respond to message get_husband_mother_in_law sent to 0@person


no

** CREATING THE 'MARRIAGE' RELATIONSHIP

| ?- rel1.

yes

** CONSEQUENCIES OF CLASS CREATION
**   - 'MARRIAGE' is made an instance of 'RELATIONSHIP_CLASS' metaclass
**   - a new integrity rule has been automatically generated by the system
**     to preserve the operational semantics of the relationship
**   - the 'PERSON' class has been enlarged with four 'virtual' attributes:
```

```
**        - 'WIFE_OF' and 'HUSBAND_OF' that support an attribute-based view of
**          the relationship 'MARRIAGE'
**        _ 'HUSBAND_MOTHER_IN_LAW' and 'WIFE_MOTHER_IN_LAW'

| ?- get(marriage) => relationship_class.

yes
| ?- get(X) => propagating_rule.

X = 0@propagating_rule

| ?- display => 0@propagating_rule.

Display of 0@propagating_rule
instance_of: propagating_rule
Slot values:
is_it_enable: yes
disable_for:
action:
  condition_result(_7500),
  current_object(_7505),
  current_arguments(_7510),
  (member(_7521,_7500),
   put_already_propagated([_7521])=>0@propagating_rule,
   moving(_7510)=>_7521,
   fail;true),
  get_already_propagated(_7551)=>0@propagating_rule,
  (_7505=_7551
     -> (get_already_propagated(_7574)=>0@propagating_rule,
         delete_already_propagated([_7574])=>0@propagating_rule,
         fail;true)
     ;  true)
condition:
  condition_result(_7506),
  current_object(_7511),
  (get_already_propagated(_7525)=>0@propagating_rule
     -> true
     ;  put_already_propagated([_7511])=>0@propagating_rule),
  findall(_7543,
     (instance_of(_7511,person),
      get_wife_of(_7543)=>_7511,
      instance_of(_7543,person),
      \+get_already_propagated(_7543)=>0@propagating_rule
     ;
      instance_of(_7511,person),
      get_husband_of(_7543)=>_7511,
      instance_of(_7543,person),
      \+get_already_propagated(_7543)=>0@propagating_rule),
```

```
  _7506),
  !,
  \+_7506=[]
event: moving
when: before
already_propagated:

yes

** NOW PEOPLE CAN UNDERSTAND THE MESSAGE 'GET_HUSBAND_MOTHER_IN_LAW'
** IN THE EXAMPLE THE ANSWER IS EMPTY.

| ?- get_husband_mother_in_law(X) => 0@person.

no

** USING THE ATTRIBUTE-BASED VIEW TO CREATE INSTANCES OF 'MARRIAGE'

| ?- new([X,[
      husband_of([0@person])
     ]]) => person.

X = 1@person

** CHECKING THAT THE INVERSE LINK HAS BEEN AUTOMATICALLY CREATED

| ?- get_wife_of(X) => 0@person.

X = 1@person

** LOOKING AT THE PREVIOUS CREATED RELATIONSHIP AS AN OBJECT
**  RATHER THAT AS AN ATTRIBUTE

| ?- get(X) => marriage.

X = 3@marriage

** DISPLAYING THE OBJECT '3@MARRIAGE'

| ?- display => 3@marriage.

Display of 3@marriage
instance_of: marriage
Slot values:
wife_mother_in_law:
husband_mother_in_law:
wedding_date:
the_wife: 0@person
```

```
the_husband: 1@person

yes

** CREATING RELATIONSHIPS AS ANY OTHER OBJECT, I.E. SENDING THE MESSAGE
**  'NEW' TO THEIR CLASS

| ?- new([X,[]]) => person.

X = 2@person

| ?- new([X,[]]) => person.

X = 3@person

| ?- new([X,[
      the_husband([3@person]),
      the_wife([2@person]),
      ed  wedding_date([july]),
      husband_mother_in_law([mary])
     ]]) => marriage.

X = 4@marriage

** RETRIEVING RELATIONSHIP INFORMATION USING THE ATTRIBUTE-BASED VIEW
**   OF 'MARRIAGE'

| ?- get_husband_of(X) => 3@person.

X = 2@person

| ?- get_wedding_date(X) => 4@marriage.

X = july

| ?- get_husband_mother_in_law(X) => 3@person.

X = mary

** CHECKING THE OPERATIONAL SEMANTICS OF THE 'MARRIAGE' RELATIONSHIP:
**  'HUSBAND' AND 'WIFE' HAVE TO BE MOVED SIMULTANEOUSLY

| ?- moving => 2@person.
The person 3@person is moving .....
The person 2@person is moving .....

yes
```

```
** EXAMPLES OF MORE COMPLEX RELATIONSHIPS

| ?- c2,c3,c4,c5,rel2,rel3,rel4.

yes

** CHECKING THAT RELATIONSHIPS HAVE BEEN CREATED

| ?- get(X) => relationship_class.

X = activation ;

X = marriage ;

X = working_in ;

X = working_in_projects ;

X = working_in_research ;

no

** DISPLAYING RULE GENERATED BY THE SYSTEM TO MAINTAIN THE DELEGATION
**  OPERATIONAL SEMANTICS OF THE 'WORKING_IN' RELATIONSHIP

| ?- get(X) => delegating_rule.

X = 1@delegating_rule

| ?- display => 1@delegating_rule.

Display of 1@delegating_rule
instance_of: delegating_rule
Slot values:
is_it_enable: yes
disable_for:
action:
  condition_result(_7500),
  current_object(_7505),
  current_arguments(_7510),
  (member(_7521,_7500),
   put_already_propagated([_7521])=>1@delegating_rule,
   stability(_7510)=>_7521,
   true;fail),
  get_already_propagated(_7551)=>1@delegating_rule,
  (_7505=_7551
     -> (get_already_propagated(_7574)=>1@delegating_rule,
         delete_already_propagated([_7574])=>1@delegating_rule,
```

```
        fail;true)
    ;  true)
condition:
  condition_result(_7506),
  current_object(_7511),
  (get_already_propagated(_7525)=>1@delegating_rule
     -> true
     ;  put_already_propagated([_7511])=>1@delegating_rule),
  findall(_7543,
     (instance_of(_7511,person),
      get_working_places(_7543)=>_7511,
      instance_of(_7543,company),
      \+get_already_propagated(_7543)=>1@delegating_rule),
      _7506),
  !,
  \+_7506=[]
event: stability
when: inexistant
already_propagated:

yes

** USING THE ATTRIBUTE-BASED VIEW 'EMPLOYEES' TO CREATE 'WORKING_IN'
**  INSTANCES AT THE TIME A 'COMPANY' INSTANCE IS CREATED

| ?- new([X,[
      employees([0@person,2@person]),
      area([computing])
     ]]) => company.

X = 4@company

** THE OPERATIONAL SEMANTICS OF 'WORKING_IN' IN ACTION:
**  - the employee gets his/her 'STABILITY' from his/her company
**  - the movement of a 'COMPANY implies the movement of its 'EMPLOYEES'

| ?- stability([t1]) => 0@person.
The stability is: high

yes
| ?- stability([t1]) => 4@company.
The stability is: high

yes
'** THE MOVEMENT OF A 'COMPANY' IMPLIES THE MOVEMENT OF ITS 'EMPLOYEES'.
**  ALTHOUGH ONLY '0@PERSON' AND '2@PERSON' ARE EMPLOYEES, '1@PERSON'
**  AND '3@PERSON' HAVE ALSO TO BE MOVED AS A CONSEQUENCE OF BEING
**  PARTNERS OF AN EMPLOYEE, I.E. BEING 'MARRIAGE' RELATED WITH AN
```

```
** EMPLOYEE

| ?- moving => 4@company.
The person 1@person is moving .....
The person 0@person is moving .....
The person 3@person is moving .....
The person 2@person is moving .....
The company 4@company is moving .....

yes

** SINCE RELATIONSHIPS ARE OBJECTS, NOTHING PREVENTS US FROM ARRANGING
**  RELATIONSHIPS IN HIERARCHIES. HERE IS AN EXAMPLE WHERE
**  'WORKING_IN_RESEARCH' AND 'WORKING_IN_PROJECTS' ARE SUBCLASSES OF
**  'WORKING_IN'

| ?- get_is_a(X) => working_in_projects.

X = working_in

| ?- get_is_a(X) => working_in_research.

X = working_in

** DEFINING RELATIONSHIP HIERARCHIES ALLOWS THE USER TO WORK AT
**  DIFFERENT LEVELS OF ABSTRACTION:
**    - if we want all the employees of a company regardless whether
**      they are workers or researchers, we retrieve the information
**      at the higher level (i.e. 'working_in' or its view 'employees')
**   - if we want only employees that are researchers, we retrieve the
**     information at a lower level (e.g. 'working_in_research' or
**     its view: 'researchers')

| ?- new([X,[]]) => academic.

X = 5@academic

| ?- new([X,[
      employees([1@person]),
      researchers([5@academic])
     ]]) => research_institution.

X = 6@research_institution

| ?- get_employees(X) => 6@research_institution.

X = 1@person ;
```

```
X = 5@academic ;

no
| ?- get_researchers(X) => 6@research_institution.

X = 5@academic ;

no

** DEFINITION OF A SECURITY RULE. IT PREVENTS USERS OTHER THAT 'GRAHAM'
**  FROM RETRIEVING THE 'SALARY' OF A 'PERSON'.

| ?- new([X,[
  active_class([person]),
  event([get_salary]),
  is_it_enable([yes]),
  when([before]),
  condition([(
    \+ current_user(graham)
  )]),
  action([(nl,write('You are not authorised to retrieve the salary'),nl,fail)])
]]) => generic_rule.

X = 12@generic_rule

| ?- get_salary(X)=> 0@person.

You are not authorised to retrieve the salary

no

** SWITCHING THE RULE OFF. BEING OBJECTS, RULES CAN BE MANIPULATED AS
**  ANY OTHER OBJECT. NOTICE THAT IF THIS RULE WERE EMBEDDED WITHIN
**  METHODS, THIS UPDATE WOULD HAVE BEEN MORE DIFFICULT TO ACCOMPLISH

| ?- update_is_it_enable([yes,no]) => 12@generic_rule.

yes
| ?- get_salary(X)=> 0@person.

X = 999

** SINCE THE INTRODUCTION OF A 'WORKING_PLACES' IMPLIES
**  THE INTRODUCTION OF AN 'EMPLOYEES', THE RULE BELOW IS
**  FIRED.

| ?- new([_,[
  active_class([company]),
```

```
  event([put_employees]),
  is_it_enable([yes]),
  when([before]),
  condition([(true)]),
  action([(nl,write('......introducing an employee....'),nl)])
]]) => generic_rule.

| ?- put_working_places([6@enterprise]) => 0@person.

......introducing an employee....

yes

** SINCE THE INTRODUCTION OF AN 'ACADEMIC' IMPLIES THE
**  INTRODUCTION OF A 'PERSON', THE RULE BELOW IS FIRED

| ?- new([_,[
  active_class([entity_class]),
  event([new]),
  is_it_enable([yes]),
  when([before]),
  condition([(
    current_object(Class),
    Class = person
  )]),
  action([(nl,write('=== A new person is being created ====='),nl)])
]]) => generic_rule.


| ?- new([X,[[ ]]) => academic.

=== A new person is being created =====

X = 9@academic

| ?- halt.
hawk% ^Z
script done on Mon Jan 20 10:33:21 1992
```