Managing complex taxonomic data in an object-oriented database

Cédric Raguenaud

A thesis submitted in partial fulfilment for the
requirements of Napier University
for the degree of Doctor of Philosophy

January 2002

Abstract

This thesis addresses the problem of multiple overlapping classifications in object-oriented databases through the example of plant taxonomy. These multiple overlapping classifications are independent simple classifications that share information (nodes and leaves), therefore overlap.

Plant taxonomy was chosen as the motivational application domain because taxonomic classifications are especially complex and have changed over long periods of time, therefore overlap in a significant manner.

This work extracts basic requirements for the support of multiple overlapping classifications in general, and in the context of plant taxonomy in particular. These requirements form the basis on which a prototype is defined and built. The prototype, an extended object-oriented database, is extended from an object-oriented model based on ODMG through the provision of a relationship management mechanism. These relationships form the main feature used to build classifications. This emphasis on relationships allows the description of classifications orthogonal to the classified data (for reuse and integration of the mechanism with existing databases and for classification of non co-operating data), and allows an easier and more powerful management of semantic data (both within and without a classification). Additional mechanisms such as integrity constraints are investigated and implemented.

Finally, the implementation of the prototype is presented and is evaluated, from the point of view of both usability and expressiveness (using plant taxonomy as an application), and its performance as a database system. This evaluation shows that the prototype meets the needs of taxonomists.

Acknowledgements

I would like to express here my gratitude to the people that made this work possible:

First of all, I would like to thank my PhD supervisors, Prof. Jessie Kennedy and Dr Peter J. Barclay. Prof. Kennedy is a Professor at the School of Computing, Napier University. She is at the origin of this work, as she directed me first to databases when I knew very little about them, and later to plant taxonomy that became a motivation for both this PhD and my current job. Without her and without the (sometimes heated) discussions we have had over the years, none of this would have been possible. I am indebted to her for her patience with me, particularly for my English writing at first, then for my stubbornness to not understand her and what she wants, especially when she is right. I would also like to thank her for having made sure that I could continue my PhD by providing me with a continuous source of income (even when it was not easy).

I would like to thank my second supervisor, Dr Barclay, a lecturer at Napier University who left the nest to explore the real world, for his support all along my PhD. His presence helped me bring my English to an (nearly) acceptable level, which was a necessity to go anywhere near the completion of this PhD and its associated papers. His ideas and his opinions have helped me building mine. Also, our interesting discussions about science fiction, French, English, Gaelic, and other non-computing matters were a welcome break from computers and plants, and his support toward the end of my research when the finalising of this thesis became painful was invaluable. I would also like to thank him for giving me feedback when he was too busy to do so.

I am grateful for the help the taxonomists at the Royal Botanic Garden Edinburgh, Dr Mark Watson, Dr Martin Pullan, and Dr Mark F. Newman have provided me during the years of this work, and for working in the weird area that is plant taxonomy. Their patience to explain how counter-intuitive things work made this work possible and interesting. I would like above all to thank Dr Newman for his patience in testing my (numerous, flawed) prototypes until a viable version was finally reached.

I would also like to thank Philippe Li-Thiao-Té for having shown me the way through a PhD in doing it first, and thanking me in his thesis acknowledgements. I hope you have a good life in Hong-Kong.

Finally, I would like to thank my parents, who have always helped me, in particular in offering me my first computer when I was 12 (they did not know the consequences), even though they don't approve of my doing a PhD instead of working for real money in the real world.

Thank you to you all.

Edinburgh, 04th December 2001

Contents

MISSING PAGE

IN

ORIGINAL

List of figures

List of tables

Contributing papers

Martin R. Pullan, Mark F. Watson, Jessie B. Kennedy, Cedric Raguenaud, Roger Hyam, "The Prometheus Taxonomic Model: a practical approach to representing multiple taxonomies", Taxon 49, pp 55-75, 2000

Cedric Raguenaud, Jessie Kennedy, Peter J. Barclay, "The Prometheus Database for Taxonomy", 12th International Conference on Scientific and Statistical Database Management (SSDBM 2000), Berlin, Germany, pp 250-252, 2000

Cedric Raguenaud, Jessie Kennedy, Peter J. Barclay, "The Prometheus Taxonomic Database", IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE 2000), Arlington Virginia, USA, pp 63-70, 2000

Cedric Raguenaud, Martin Graham, Jessie Kennedy, "Two Approaches to Representing Multiple Overlapping Classifications: a Comparison", 13th International Conference on Scientific and Statistical Database Management (SSDBM 2001), Fairfax, Virginia, USA, pp 239-244, 2001

Cedric Raguenaud, Martin R. Pullan, Mark F. Watson, Jessie B. Kennedy, Mark F. Newman, Peter J. Barclay, "Implementation of the Prometheus Taxonomic Model: a comparison of database models and query languages and an introduction to the Prometheus Object Oriented Model", to appear in Taxon 51, 2002

Cedric Raguenaud, Jessie Kennedy, "Multiple Overlapping Classifications: Issues and Solutions", 14th International Conference on Scientific and Statistical Database Management (SSDBM 2002), Edinburgh, Scotland, to appear, 2002

1 Introduction

Classification is a widespread concept that helps categorise, and therefore simplify knowledge in order to facilitate its manipulation. Classifications also provide a means to increase knowledge by representing relationships between classified things that might provide new insights into the data (e.g. discovering that two groups thought to be independent are in fact related in some way, deducing knowledge from relationships between groups such as pharmaceutical properties), allow automatic reasoning (e.g. propagation of attributes in computing models [Pirotte '97] [Díaz '90]), or support user interactions (e.g. searches).

For example, library catalogues are a kind of classification where books are placed into categories (e.g. "Fiction") in order to ease interaction and simplify tasks related to the books. These categories are not necessarily exclusive: a book may appear in several categories simultaneously (e.g. English writing, Crime, and in the authors' category). These categories overlap even more if independent libraries are put together, as librarians may have chosen alternative classification mechanisms. Medical classification mechanisms, such as the International Classification of Diseases (ICD), catalogue and relate diseases in order to make diagnosis, prevention, and cure possible. Diseases are put into categories according to four classification criteria (topological, etiological, operational, and ethical-political), which may lead to different classifications [Bowker '99]. Classification of living organisms also generates several classifications. For example, viruses have many origin stories (they could have been originally plasmids or transposons, they might have degenerated from primitive cells, they might have evolved from RNA polymers, or they might have evolved from viroids) therefore can be classified in different ways [Bowker '99]. Another example is plant taxonomy, where plant specimens are classified in order to describe the knowledge available about them and relate similar specimens or groups of specimens (*taxa*). In taxonomy, a single taxonomist may decide to classify a set of specimens differently overtime because of new discoveries or ideas. In addition, the integration of multiple sources makes the management of all classifications difficult.

These classifications, especially overlapping classifications, are therefore a fact of life and their manipulation a necessity of computing systems. This work endeavours to develop a mechanism to support these classifications through the study of plant taxonomy and its requirements. As will be shown, plant taxonomy is a very complex domain that poses several interesting challenges to the computing world. It has therefore been chosen as the application domain for the work presented here and is used to test the ideas proposed and the prototypes built. The following sections introduce the motivations for this work, starting with plant taxonomy, and the aims of this thesis.

1.1 Plant taxonomy

Taxonomy is a biological domain that uses classifications to categorise and understand the living world. The work of a taxonomist is to collect specimens, study them, classify them according to certain aspects, and name them. This classification work allows references to living things in other biological domains. Taxonomic classifications have two important aspects: they need to unambiguously identify specimens so that they can be referred to uniquely; and they can be redefined over time. Being able to identify specimens unambiguously is not only important for taxonomy, but for all fields of biology and more generally any domain that needs to refer to living things uniquely. For example, pharmaceutical research needs to be able to refer to organisms unambiguously so that experiments can be reproduced and checked. If specimens are not identified properly or if they are confused with others because of an inadequate classification system, all experimental results may be meaningless. When geneticists decode the genome of a specimen (or set of specimens), they need to know how the specimens they study relate and they need to make sure that they belong to the same group.

As taxonomic classifications are built partially on opinions (although substantiated by observation), it is impossible to describe a definitive classification for a set of specimens. Indeed, two taxonomists may consider different aspects of the specimens they observe as most important, and therefore they may build different classifications where these different aspects are highlighted. In addition, progress in technology and ideas mean that new aspects of specimens can be studied (e.g. DNA sequencing) or new ideas used to guide classification (e.g. changing geographical borders, the new availability of taxonomic information from previously closed countries such as Russia or China). Over time, this process generates several classifications of the same (or overlapping) specimen groups by different taxonomists. As in taxonomy, anything that has been validly published is considered valid forever, even though the publication may contain errors (misapplication of a name, mistakes in the observations), all the classifications created since the dawn of taxonomy are considered valid today and may be used to base new classifications on. This process generates data that taxonomists must take into account when they reclassify (*revisions*). Computers and databases therefore become more of a necessity in taxonomy in order to support taxonomists in their work.

The extent of the data that must be managed in order to represent a significant part of the known plant kingdom and in order to proceed to the revision of a plant group is enormous. Families that contain thousands of genera, and genera that contain hundreds of species are not uncommon. Currently, taxonomists do a lot of their work on paper, which has many practical limitations. For example the representation of the names appearing in certain plant groups during a revision can take several sheets of paper and hardly offers a suitable medium for taxonomists to have an overall idea of the group. If the descriptions associated with the groups to which the names are attached were to be represented, the amount of data would be overwhelming. Working on paper also requires that the taxonomist does all the work and therefore is able to remember all the data or a significant part of it so that comparisons

can be performed. It is clear that such amounts of data cannot be effectively manipulated by hand. The amount of data necessary to working taxonomists is not only important in the context of a single classification, but as they produce multiple concurrent classifications, the amount of data necessary to capture knowledge about the living world increases at a high rate.

Moreover, there is a need for official taxonomic repositories that could be used in judiciary actions (e.g. ban of some substances issued from the processing of specific plants) or biodiversity initiatives (e.g. species conservation). These repositories, although not required to contain detailed information about all aspects of plants, should allow unique identification or referral of organisms. As these repositories should be accessed by both specialised taxonomists and members of the public, the information should be exact (taxonomists looking for information about existing classifications want exact information) and consistent (members of the public are not able to judge the validity or the relevancy of the information they receive). The amount of data necessary for such projects is even more important than for a single revision or classification, and is undoubtedly beyond the abilities of a single person or group of persons. Plant taxonomy is therefore a domain in urgent need of suitable computing support.

1.2 Models of taxonomy

Many models of taxonomy have been proposed (e.g. IOPI [Berendsohn '97], CDEFD [Berendsohn '99]), some coupled with a database model (e.g. Taxon-object [Saarenmaa '95], HICLAS [Zhong '96], Pandora [Pankhurst '93]). Most of these models of taxonomy are based on a biased idea of taxonomy, namely that there is one single accepted way to look at the world and its living organisms. However, as indicated earlier, there is no single classification to describe the world, therefore these single view models of taxonomy are not suited for taxonomic work in a context beyond a single isolated classification or revision. Moreover, in ignoring the multiplicity of taxonomic classifications, they are unable to support taxonomists fully in their work. For example, such models are unable to provide taxonomists with information about specimens classified in various published classifications. They are also unable to support automatic processes such as finding *synonyms*, i.e. groups that share a certain number of specimens or taxonomic types, or allocating names. A few of these models (IOPI, HICLAS) recognise the importance of multiple classifications of specimens and propose an approach to capture them. However, even those models are limited in their application to taxonomic work: IOPI supports multiple classifications through the creation of potential taxa, which leads to a very large number of potential taxa; HICLAS captures multiple classifications by recording the life of a taxon (its movements from classification to classification), but by doing so only records the opinion of a taxonomist, not objective information.

1.3 Taxonomic databases

Many database models have been used to support these models of taxonomy (relational, object-oriented, and graph-based models). However, the majority of them rely on the relational model, which has well-known limitations to support complex modeling domains and complex applications (e.g. CAM/CAD). As a consequence, existing taxonomic databases do not support sufficiently the work of taxonomists who still have to rely on work on paper and slow research in libraries. For example, they are unable to implement the rules of taxonomic work (ICBN [Greuter '94]). Other models have been considered (e.g. graph-based [Zhong '96]) but these models are also limited in their ability to support complex application domains. Indeed, they are built on a very simple mathematically sound model that requires large amounts of data in order to describe sizable domains.

The representation of biological classifications in particular is difficult. A few approaches have been proposed (e.g. Materialization [Pirotte '94], Power types [Odell '94a]). These approaches represent classifications at two different levels: materialization uses the class/meta-class level, and power types use the instance level. The choice of level for the representation of the classifications depends on the data to be represented. Indeed, a class/meta-class approach implies that not many changes will occur once the classifications have been created. Changes would mean that schema evolution (e.g. [Bertino '93]) must be supported in the system, with the restrictions it brings. Working at instance level implies more flexibility, but can introduce a performance overhead, as many aspects of the data will not be fixed but be dynamically recalculated when necessary[1].

1.4 Motivation

This work is therefore motivated by the following facts:

- Taxonomists need databases that support their way of working, and the many taxonomic database systems created over time show this need (e.g. Pandora [Pankhurst '93], Alice [White '93], Taxon-Object [Saarenmaa '95], BG-BASE [Walter '93], Brahms [Filer '94], CDEFD [Berendsohn '99] (model only), IOPI [Berendsohn '97] (model only) and HICLAS [Zhong '96]).
- A major problem of existing taxonomic models and applications is their failure to support multiple overlapping classifications. This leads to inconsistencies, errors, and a biased view of taxonomy.
- Although supporting multiple overlapping classifications is a difficult task, it could be generalised in order to be embedded in a database system so that other application domains can exploit it[2]. This would make classification application development easier and more efficient.

[1] Whatever approach is chosen, the representation of multiple overlapping classifications is never tackled, and misunderstanding of taxonomic information lead the models to make these classifications impossible.
[2] Whether knowledge should be concentrated in the database or in the user application is debatable [Paton '99] and there is no clear limit between the two approaches. For this work, it is considered that the database should support as many features as possible, but should remain generic, i.e. the features should not be supported in a specific context only (e.g. plant taxonomy).

- No model to support multiple overlapping classifications exists in the literature, although this is not a plant taxonomy specific problem. These multiple overlapping classifications are a feature of many domains, therefore their investigation may provide techniques for a wide range of applications (e.g. in a library catalogue, books could be classified simultaneously in categories English, Fiction, Crime, in publisher categories, and author name categories).

1.5 Purpose of this work

This research work envisages the provision of generic features to support multiple overlapping (possible conflicting) classifications for plant taxonomy. This will be achieved through:

- The extraction of requirements and analysis of these requirements. As plant taxonomy is a domain that requires multiple classifications, it is used as an application domain in order to test the ideas presented here. To do so, this thesis first identifies the basic requirements of such a mechanism in as broad a context as possible, i.e. necessitating as little changes to an existing database and keeping in mind that other domains may use the same principles.
- Using these requirements, traditional database models are reviewed and their ability to support taxonomy discussed.
- It then endeavours to provide a solution to the problems highlighted in taxonomy, especially the multiple overlapping classifications problem. The resulting model should allow taxonomists to capture, compare, and contrast existing and new classifications.
- In order to provide a usable database system that supports these multiple classifications, this thesis also proposes a query language adapted to that specific problem (although not restricted to it).
- In addition, features such as integrity constraints and rules are investigated in order to support a wide range of taxonomic tasks that are identified as being able to be automated, and to support data integrity in general.

1.6 Structure of this thesis

This thesis is organised into 8 chapters, plus references and appendices. Chapter 2 explains taxonomy in sufficient detail to show what its peculiarities are, and motivates the rest of the work. It also reviews a new model of taxonomy elaborated within the Prometheus group [Pullan '00]. To conclude, chapter 2 lists a series of requirements, motivated both by taxonomic considerations and database considerations, which a taxonomic database should offer.

Chapter 3 and subsequent chapters present work developed for this thesis. Chapter 3 presents and discusses several approaches to biological classifications. These classifications are not to be assimilated with object-oriented database system classifications. These models have been proposed in the literature as good ways of representing biological classifications, but they do not consider the peculiarities of plant taxonomy fully. This chapter also presents the main database models available today (relational,

object-oriented, graph-based, and extended object-oriented), and studies their ability to support taxonomic applications and data. It shows that several of these models propose interesting features that could support some aspects of taxonomic information (e.g. graph manipulations, extensive programming language, constraints support). However, it also shows that there is no single model philosophy that supports all the requirements of taxonomic work.

Chapter 4 describes the database model developed during for this thesis work. The new model emphasises relationships as a primary modelling and implementation tool. These relationships are added to an ODMG-based database model. This chapter argues that relationships should replace references in order to allow semantic relationships in all circumstances. These relationships, or links, are also used to support the definition of multiple overlapping classifications according to the requirements specified in chapter 2. They also offer several features that provide a better development environment than most other models (e.g. semantics, constraints).

Chapter 5 presents the query language defined to support classifications (in particular plant taxonomy), and introduces the rules/constraint mechanism. OQL is extended in order to benefit from the new database structures and offer support to taxonomic applications. The extensions include: uniform treatment of relationships and objects, specific operators for manipulation of relationships (e.g. selective downcast), exploration and traversal of graphs, and extraction of (parameterised) graphs. The rules/constraint mechanism is inspired from several existing mechanisms and includes immediate and differed rules, several execution strategies, and automatic actions (e.g. transaction abortion). It also provides additional features specific to plant taxonomy such as interactive rules, relationship-centred rules, and rules with condition of applicability.

Chapter 6 presents the architecture and implementation of the prototype. The prototype is a framework that proposes classes that can be used or extended to fit user applications' needs. The architecture of the prototype is discussed in detail and justified. The chapter also shows how some of the requirements expressed in chapter 2 have been taken into account during the building of the prototype.

Chapter 7 evaluates the system described in chapters 4, 5, and 6. The evaluation is performed from two distinct viewpoints: the taxonomy viewpoint and the database viewpoint. The taxonomic evaluation is performed through the description of typical interactions between a taxonomic application and a taxonomic user and the database system. Through the specification and the implementation of a benchmark inspired by OO7 [Carey '93], the performance of system built is compared to the performance of its underlying storage system, and the cost of the features described in chapter 4 is calculated and analysed.

Finally, chapter 8 concludes the thesis and discusses the success of the work presented here. It also outlines further work, which includes performance improvements and distribution of the system over many localised taxonomic database systems.

2 Taxonomic work

The purpose of this work is the design of a database system that is able to handle multiple classifications of concepts. As an application example, plant taxonomy has been chosen, as the rules that govern it are complex, the revision of groups is common, therefore the rate of evolution of knowledge and data is rapid. The resulting classification information is hard to represent without the help of a suitable computing system. The problems encountered in plant taxonomy are not unique. They not only appear in other biological taxonomy areas, but also in other domains where classification is an important process (e.g. in libraries). The findings of this work are therefore applicable to similar problems in other domains.

This section describes how taxonomists work, what are the known models of taxonomy, and the model that represents more faithfully how taxonomists really work. This section also extracts the basic requirements of a suitable database model for this application domain in particular and for classifications in general.

2.1 Taxonomic working practices

Taxonomy is defined as "orderly classification of plants and animals according to their presumed natural relationships" [Webster '01]. The purpose of taxonomy is the creation of classifications of *specimens* (that can be biological specimens or other kind of specimens), and the attribution of names to these specimens. These classifications allow a better identification and understanding of what the specimens are, which allows the extraction of further knowledge by studying the relationships these specimens have with each other. In this section, the way taxonomic work is performed is presented and the rules that govern it are given. Although the descriptions focus on plant taxonomy (which is hereafter called *taxonomy* for simplification), they apply in part to zoological taxonomy and other taxonomic disciplines.

The processes involved in taxonomy are of two kinds: one is classification, and the other naming. The first one allows the description of classifications from which knowledge can be extracted by studying the relationships between different groups (e.g. medicinal properties, evolution history). The second one allows the creation of handles for specimens so that they can be referred to unambiguously. It is argued in [Pullan '00] that these processes are distinct in nature. It is shown why in this section. The classification process is first described, and then the naming process is presented.

2.1.1 Classification

The first step of proceeding to the creation of a classification is the collection of as much published material on the group to be studied as possible. This information can be found in previous published classifications of the same or similar groups, or from determinations on herbarium sheets. Determinations do not have classification value, as they are simply the application of a name by a taxonomist to a specimen on a herbarium sheet (plant specimen) without justification or publication. They only show a taxonomist's point of view and represent how the taxonomist thought that the plant specimen should be named. They can however provide information as to what a particular taxonomist thought when in the process of building a classification and are therefore useful information.

The next step is the collection of plant specimens. These specimens can be collected in the field, in herbaria, or from published monographs. Since taxonomy is a very old discipline and has grown over the centuries, taxonomists rarely classify entirely new specimens or plant groups. They usually have an idea of where they might come from and where to collect new specimens. This information can be found in previous classifications of a similar group (especially in the case of a revision of a specific taxonomic group). When all the specimens have been collected, a taxonomist starts the classification process.

During this process the taxonomist chooses a set of criteria to be used in order to distinguish the groups of specimens and the relations between these groups. For example, one taxonomist might choose as a discriminating property the shape or colour of leaves, the form of petals, or the place of origin of the specimens. These criteria are chosen arbitrarily, as there is no rule that dictates this process in the International Code of Botanical Nomenclature (ICBN [Greuter '94]). The choice is not totally random, as a taxonomist usually has an idea of the properties of the specimens under study from the collection of existing information in previous classifications and previous professional expertise. However, this choice is very important, as it underlines the whole process of creating a classification. Indeed, if a taxonomist chooses different sets of criteria, different classifications might be created.

During this phase of the work, the taxonomist arranges specimens into piles according to the set of criteria that were chosen. Each of these piles is then transformed into a group of specimens, or *taxon*. The set of specimens that are placed into a taxon is called the *circumscription* or *delimitation* of the taxon. There is no rule on the number of taxa or the number of specimens in each taxon. However, at least one specimen must be present in each group in order to build a specimen-based process. When all the piles are created, they act as surrogate for the specimens they contain. They are then grouped with other piles (classified) in order to form a hierarchic structure: the classification hierarchy. The process is repeated as many times as necessary for each additional level. Plant taxonomy classifications are therefore in essence sets of connected one-level classifications of specimens or specimen surrogates, where each level is classified by the next higher level.

Once classifications have been created, levels, or *ranks*, are assigned to each of the piles in the hierarchy. Ranks are the essential information that allows the application of the ICBN to a given classification. The ICBN distinguishes three types of ranks: primary ranks, secondary ranks, and additional sub ranks. Primary ranks contain *Regnum*, *Divisio* (also called *Phyllum*), *Classis*, *Ordo*, *Familia*, *Genus*, and *Species*. Secondary ranks include *Tribus*, *Sectio*, *Series*, *Varietas*, and *Forma*. Sub ranks are created by adding "sub" in front of primary or secondary names: *Subregnum*, *Subdivisio*, *Subclassis*, *Subordo*, *Subfamilia*, *Subtribus*, *Subgenus*, *Subsectio*, *Subseries*, *Subspecies*, *Subvarietas*, and *Subforma*. These sub ranks represent sub-divisions of the rank whose name "sub" has been added to. All these ranks and their relative position are shown in Figure 1. Only primary ranks are compulsory in classifications, and taxonomists are free to use any number of the secondary and sub ranks, as long as their order is always valid. For example, it is acceptable to work on a classification where only Regnum, Divisio, Ordo, Genus, Sectio, and Species have been selected (all primary ranks and one secondary rank). Ranks are ordered, i.e. each rank follows another rank in a very specific order. According to the ICBN, ranks specify the way taxa can be arranged, their properties, and later the method to derive their names. For example, the ICBN specifies that a taxon that is placed at rank Species must always be placed below a taxon that is placed at a rank between Genus inclusive and Species exclusive. Taxonomists usually select only a portion of the rank hierarchy in which to do their work because the amount of information the conjunction of all ranks would produce would be unmanageable. For example, a typical range would be between Genus and Species (this can include hundreds or thousands of groups) or Genus and Subspecies.

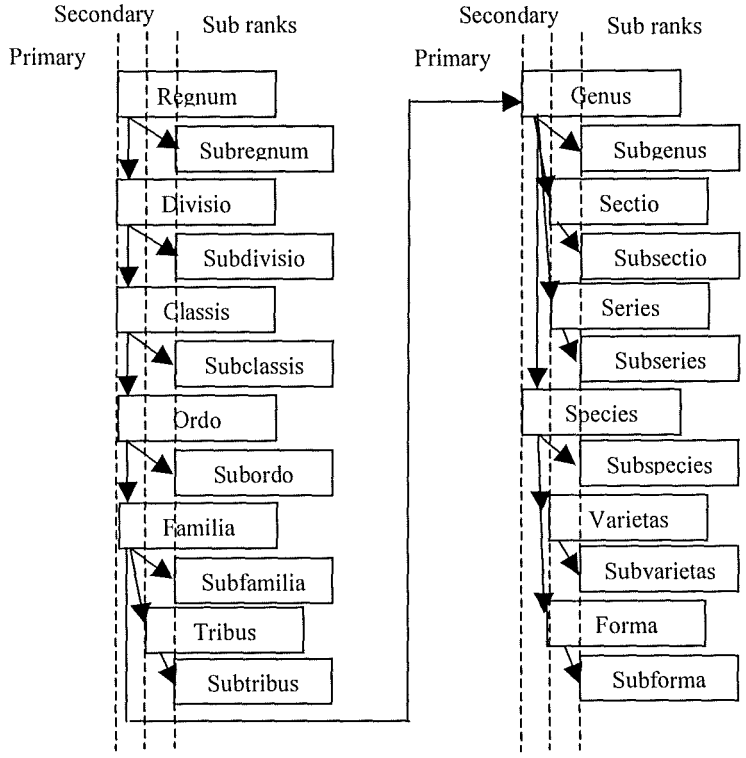


Figure 1: Hierarchy of ranks

When these classifications are published, additional information, such as author, publication, and ideally classification decisions (e.g. what characteristics of plants have been used to build the various levels of the classification), need to be added to the objects that represent classification nodes.

Once a classification has been built, names must be found for the various groups that have been defined.

2.1.2 Naming

Creation of names.

Names in taxonomy do not carry any taxonomic opinion, i.e. they do not represent any classification information. They are only the records that a name has been used and published at a specific rank by a taxonomist or that certain combinations of names have been used.

The choice of names is largely free. However, certain rules apply. They should be single worded except Genus names that can use a hyphen. Names at ranks between Series and Species (Species excluded) must start with a capital letter, whereas names at ranks below Species and at Species rank must start with a lowercase letter. In addition, the ending of names at ranks above Genus are pre-defined: names at family level must end with –aceae[3]. Names at rank sub-family must end with –oideae. Names at rank tribe must end with –eae. Names at rank sub-tribe must end with –inea.

Names form small hierarchies independent of classification and have an importance of their own. Two hierarchies can be distinguished: type hierarchies and placement hierarchies. Type hierarchies represent the fact that some names or specimens have been selected as the *taxonomic type* for other groups. For example, a Genus level name must have a Species level name as one of its taxonomic types, and a Species level name must have one or more specimens as its taxonomic type (Figure 2). There exist many different taxonomic types that convey slightly different semantics. For example, a holotype signifies that the taxonomic type was selected by the taxonomist that published the name; a lectotype signifies that the taxonomic type has been selected by a taxonomist that was not the taxonomist that published the name (the information might have been missing or the rules of nomenclature may have changed); a neotype signifies that the type specimen has been lost, therefore a taxonomist that is not the taxonomist that has published the name has selected a new taxonomic type (probably among the isotypes); a syntype is a taxonomic type that is a synonym of another taxonomic type, and an isotype is a taxonomic type that is equivalent to an existing taxonomic type (lecto, holo or neo). The rules that govern the use of these specimens vary slightly. For example, when a name is derived from a group of specimens, the holotype is always the taxonomic type to be used in priority, then the lectotype, then the neotype. Isotypes are not used for naming if they are not selected as lectotypes. Names can have many

[3] 8 exceptions: Palmae, Gramineae, Cruciferar, Leguminosae, Guttiferae, Umbelliferae, Labiatae, Compositae

taxonomic types, but they can only have one lectotype, or one holotype, or one neotype. They can however have as many isotypes as necessary.

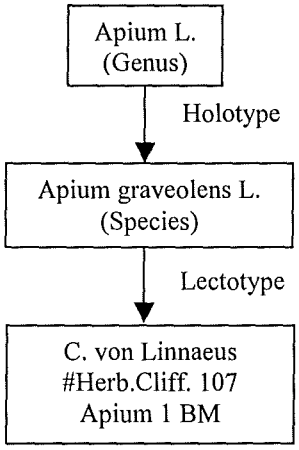


Figure 2: Simple taxonomic type hierarchy

The placement hierarchy is only used for nomenclatural completeness. For example, it is necessary for a Species name to be related to a name of rank Genus so that the binomial name can be derived accurately, e.g. the Species name graveolens can be paired with the Genus name Apium in order to form the complete Species name *Apium graveolens*. If no nomenclatural information is needed (e.g. for names at ranks above Genus which are not composed names), no placement relationship is used. It is important to note that this placement relationship only represents the use of a specific combination of names, not a classification statement (e.g. graveolens specimens are part of Apium).

In addition, names contain the name or abbreviation of the name of the taxonomist that has published it. For example, in Figure 2 the two names are followed by the letter L., which signifies that the taxonomist whose abbreviation is L. (Carlus von Linnaeus) has published the name. If a name is reused and placed into a new context (e.g. when a Species name is placed in a new Genus name), the name of the taxonomist that used the name first is placed between brackets. For example, if a taxonomist whose abbreviation is T. published Cyclospermum graveolens after Apium graveolens L. had been published, the new name would become Cyclospermum graveolens (L.)T.

Derivation of names.

Once the classification has been completed and ranks assigned, the taxonomist selects some specimens in each group. These specimens are called *type specimens*. A type specimen is a specimen that has been published as a taxonomic type (a typical representative) for a name. All names must have a taxonomic type (the process is called *typification*), that is used in order to derive names from sets of specimens. The oldest published taxonomic type specimen is selected from each group, and its name becomes the name of the group. If no type specimen was placed in a group, one specimen is elected as the taxonomic type of the group, and a new name is created and published. The process repeats with the taxa that contain the specimens. Indeed, taxa can also be the taxonomic type of other taxa of higher rank. For example, Genus taxa have a Species taxon as their taxonomic type. In that case, instead of

selecting and electing specimens, taxa or names are selected and elected as taxonomic types. In the case of a new use of a name (e.g. a Species name has been placed in a new Genus), a new name must be created and published. For example, if Apium inundatum L. exists and its taxonomic type is found in a classification taxon, but the classification taxon is placed under a taxon whose derived name is Heliosciadium, a new name Heliosciadium inundatum (L.) must be created and published so that groups in the classification can be named.

The assignation of names from classifications is both a top-down and bottom-up process. The selection of names starts from the highest level of the classification hierarchy. Each of the groups encountered is named down to the lowest level using taxonomic type information and the ICBN. This is particularly important for multinomial names (e.g. Species names and below), as the name of their higher taxon must be known before they are themselves named. Indeed, as explained before, the use of an existing name in a different context (e.g. moving a Species to a different Genus) requires the publication of a new name. But the selection of a name for each level involves the set of all specimens that have been used at any level below the group. For this, all groups that were placed in the taxon under study and all the groups placed in groups below that are examined by recursing down classification trees until the specimen level is encountered, which may vary from one branch of classification to another. Each of the specimens found is examined, and the taxonomic type hierarchy is traversed in a bottom-up fashion from specimens in order to find the names that have been published at the appropriate rank. When such a name is found, it becomes the name of the new taxon. If no such name is found, a new name must be published. The process then goes down a level, and each of the groups placed in the first taxon is named. In the case of multinomial names, special care must be taken to ensure that the combinations of names that are derived have already been validly published.

Figure 3 shows an example of derivation of names in a classification. Taxon 2 is first created by grouping two specimens together because they have common properties that are considered important by the taxonomist. Then, this taxon is put inside another taxon (Taxon 1), possibly along with other taxa. Taxon 2 is declared to be at rank Species, and Taxon 1 at rank Genus. In order to derive names, first Taxon 1 is named. All specimens described in one of its subgroups, at any level, are collected, and the taxonomic type specimens extracted. Since Taxon 2 contains two type specimens, there is no need to elect new taxonomic types. When examining the dates of publication of the names of these two type specimens, it can be seen that one is the taxonomic type of Apium repens (Jacq.)Lag. and the other one the taxonomic type of Heliosciadium nodiflorum (L.)W.D.J.Koch. These two names are placed in Genus names, but only Heliosciadium nodiflorum (L.)W.D.J.Koch. is the taxonomic type of another name. Therefore that name is selected and the traversing of the hierarchy of taxonomic types upwards continues. Heliosciadium W.D.J.Koch. is reached, which is a name placed at rank Genus. Since Taxon 1 is also placed at rank Genus and only one name at rank Genus was found, Taxon 1 becomes Heliosciadium W.D.J.Koch. If many names at rank Genus had been found, the oldest validly published one would have been chosen. Then Taxon 2 is considered. Taxon 2 is a taxon at rank Species, therefore the process applied to Taxon 1 is repeated but stopped at that rank. When examining the type

specimens, it can be seen that Taxon 2 can either be Apium repens (Jacq.)Lag. or Heliosciadium nodiflorum (L.)W.D.J.Koch. However, Apium repens (Jacq.)Lag. was published in 1821, whereas Heliosciadium nodiflorum (L.)W.D.J.Koch. was published in 1824. Taxon 2 must therefore be Apium repens (Jacq.)Lag. However, because Taxon 2 is a Species group, its name is a binomial name. But the combination of Heliosciadium (Taxon 1) and repens (the epithet name of the binomial Species name) (Taxon 2) has never been published. Therefore a new name needs to be published, Heliosciadium repens (Jacq.)Raguenaud., and the type specimen of Apium repens (Jacq.)Lag. is elected to be the taxonomic type of Heliosciadium repens (Jacq.)Raguenaud.

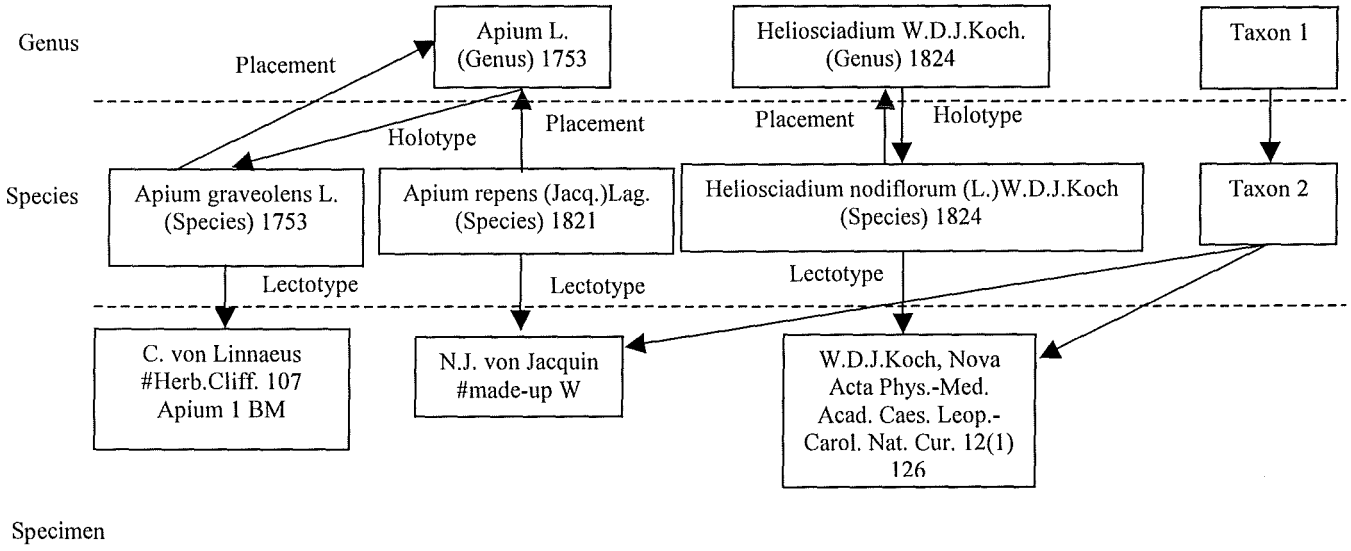


Figure 3: Derivation of names example

2.1.3 The multiple classification problem.

One of the main problems associated with taxonomy is the presence of multiple conflicting classifications. Because, as was said, taxonomists choose the set of criteria that will be used in the process of building a classification, many taxonomists may choose many different sets of criteria and see the same set of specimens through them. Even a single taxonomist might choose different sets to refer to the same group at two different points in time, for example because of advances in technology (e.g. the increasing use of DNA has generated many new classifications). Because these criteria are central to the process of classifying, choosing different sets of criteria often results in building different classifications. For example, if leaf shape is chosen as an important criterion, the resulting classification is likely to be different from a classification built on the assumption that place of collection is more important. As taxonomists do not need to relate to existing classifications and place the new work in the context of all previous classifications, the result is multiple overlapping classifications that can be conflicting.

The following example (Figure 4) shows a possible scenario that leads to the creation of multiple classifications of the same set of specimens. In order to make understanding easier, actual taxonomic information is not used, but geometric shapes are selected to illustrate the argument. The top left figure is the earliest classification and is based on a smallish set of specimens. The criterion used for this classification was the shape of the specimens, which resulted in a two-level hierarchy. The specimens are represented as shapes, the first level of the classification as a plain line oval that could be a Species rank group, and the highest level is represented by a dashed line oval that could be a Genus rank group. As this classification is the first classification, the taxonomist elects type specimens for each group and decides on a name (that needs to be published). The Squares group is *typified* by the white square because in the taxonomist's opinion this is the most typical example of square shapes, the Triangles group by the light grey triangle and the Ovals group by the black oval. Shapes in general are typified by the white square specimen because it is the oldest published type, hence the Shapes group's type is the Squares group.

Subsequently, (top right) a second taxonomist decides that an intermediate level in the classification would make things clearer (for example at rank Section). New groups are created that represent finer gatherings of specimens. As a result of the reordering of specimens, some groups do not have types, therefore the taxonomist chooses to elect type specimens in these groups (white rectangle and dark grey circle). The application of the ICBN results in the assignment of names to groups that contain already published types: Squares, Ovals, and Triangles that contain the specimens white square, black oval, and light grey triangle respectively. The groups that contain new type specimens receive new names: Rectangles and Circles for the groups that contain white rectangle and dark grey circle respectively. These Species level groups are gathered in higher level groups: 4 angled shapes, that contains Rectangles and Squares, 3 angled shapes, that contain Triangles, and Round shapes, that contain Ovals and Circles. The type specimen of these groups should be the oldest published type specimens from lower groups, therefore the type specimen of 4 angled shapes is white square, the type specimen of Triangles is the light grey triangle, and the type specimen of Round shapes is black oval. By extension, taxonomists say that the group Squares is the type of 4 angled shapes, Ovals is the type of Round shapes, and Triangles is the type of 3 angled shapes. The same rule applies to the highest level group whose type specimen is the white square, therefore its name remains Shapes.

A third taxonomist (bottom left) finds some new specimens (diamond shapes) and decides that shape is not an important characteristic after all and reclassifies the larger specimen set according to their brightness. This creates a two level classification with five groups (he ignores one particular shade, the mid-grey square, as there is only one instance of it and it must therefore be a mistake to include it in the classification). Co-incidentally each group contains an existing type specimen and therefore no new types need to be defined for the classification. In practice often several types will end up in one group, which then requires the oldest type specimen to be chosen as the type. Notice that now, after rearrangement of shapes, the group called Circles not only contains a circular shape, but also contains a triangle, a square, and an ovoid shape. Since the ICBN requires that the oldest type specimen represents

the group it belongs to, this is an expected, although unintuitive, result. Because the white square is the oldest published specimen, it remains the representative for all the shapes and the group called Squares is the type of all the shapes.

Finally a fourth taxonomist (bottom right) undertakes a revision, and reclassifies the specimens by shape again. But because new specimens have been discovered by taxonomist 3, and because taxonomist 4 is aware of taxonomist 2's work, the new classification contains 3 levels as taxonomist 2's classification, and new specimens, as in taxonomist 3's classification.

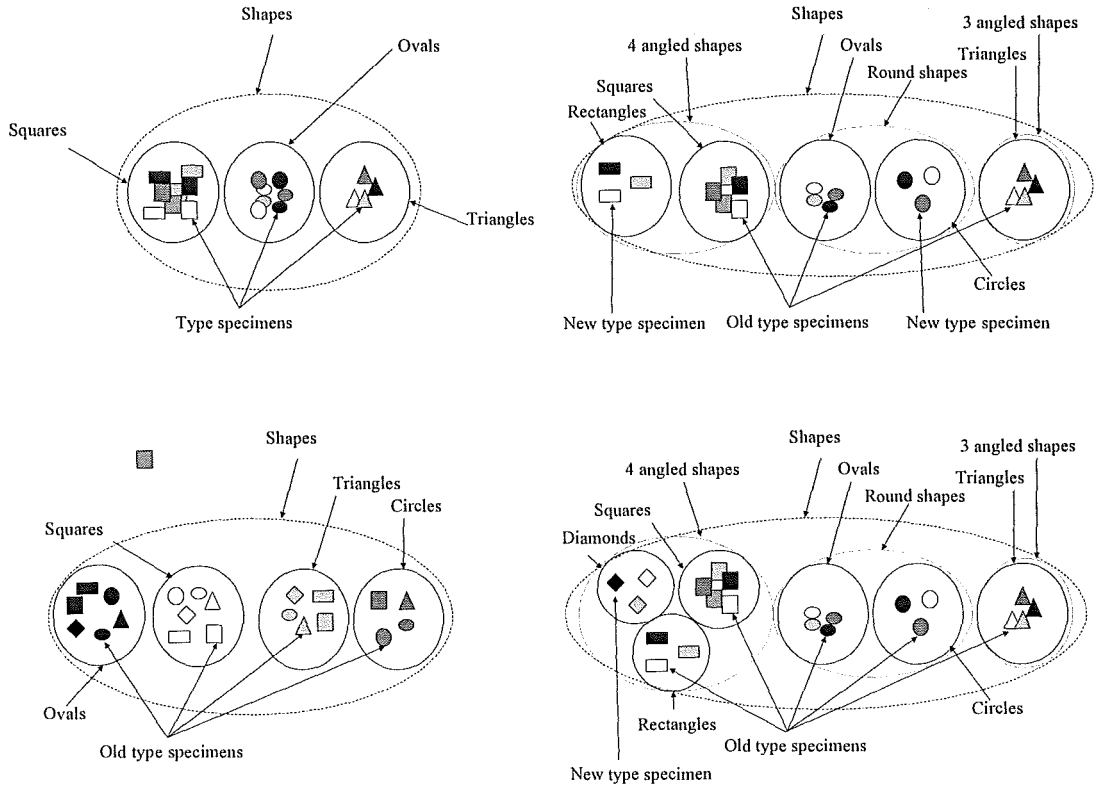


Figure 4: Multiple classifications

In the process of a revision, it is common that a taxonomist experiments with different representations of the groups that are being classified. It is also common that taxonomists want to compare newly created classifications to previously published classifications. The groups can be compared on the basis of the set of specimens they contain at any level. From these comparisons, synonyms can be discovered. Two taxa are synonyms if they contain overlapping sets of specimens. This overlap can be complete (full synonyms) or partial (*pro parte* synonyms). In addition, these synonyms can share the same taxonomic type (in which case they are *homotypical* synonyms) or not (in which case they are *heterotypical* synonyms).

The representation of multiple classifications is an important consequence of taxonomic work. As it is inevitable that many classifications will be created over time using overlapping sets of specimens, and given that all these classifications are equally valid and can be referred to in order to name a particular specimen, it is essential to be able to specify the classifications, or contexts, in which specimens are described. It is therefore unthinkable to create a single classification that would represent everything. Doing so would impair the representation of knowledge about existing species and would potentially lead to loss of information. This loss of information may lead to communication problems or even legal problems. For example, when a pharmaceutical company undertakes research on a specific substance and refers to it as a product of a specific plant using a classification scheme that is not aware of alternative means of classifying the same organism, there is no way for this company to be aware that the same plant has already been studied by a different company or even by another of its departments. It is also possible that two scientists working on the same specimens are not aware of that fact because they both use different classification schemes and are not aware of different denominations for their specimens. It has even been seen that research has been done on specimens as unrelated specimens, whereas they were in fact classified as belonging to the same species in another classification. That research has produced strange correlated results that have alerted the scientists.

It is also important that this process be specimen-based, as specimens, because they are physical entities that can be seen and studied in detail, are the only piece of objective information available after the process of classification has ended. Indeed, taxonomists are not required to describe their opinions and assumptions when they publish a classification. It is therefore impossible to be certain of what a taxonomist was thinking when building a classification. Therefore, specimens are used by other taxonomists to understand the meaning of specific groups, thereby providing a basis for objective comparison. A name-based process would create ambiguity and inconsistency. For example, Apium graveolens L. is a name that has been used by at least two groups of taxonomists (Tutin and Popper et al.). However, without specimen information, it is impossible to know whether these two names represent the same concept or if they have been used in a different manner, but the groups classified have inherited the same names. It can be seen in the example in Figure 4 that the same name can be used to refer to very different groups of specimens in different classifications.

2.2 Known models of taxonomy

From the description of taxonomy in the previous section, two important points can be identified:

1. The taxonomic process is divided into two distinct processes, namely the creation of classifications and the creation and derivation of names.
2. It is vital to be able to represent multiple classifications in order to capture accurately the state of the knowledge and the reality of the world.

Given these two important requirements, existing taxonomic models can be studied (e.g. Pandora [Pankhurst '93], Alice [White '93], Taxon-Object [Saarenmaa '95], BG-BASE [Walter '93], Brahms [Filer '94], CDEFD [Berendsohn '99] (model only), IOPI [Berendsohn '97] (model only) and HICLAS

[Zhong '96]) and their weaknesses discussed. It will be shown that no taxonomic model properly supports taxonomic work and taxonomic practices (i.e. ICBN), and that only two models are close to providing an adequate support for taxonomy (HICLAS, IOPI).

Regarding requirement (2), it can be shown that most of all known taxonomic models do not satisfy the criterion. Indeed, these taxonomic models or systems adopt a simplified view of taxonomy where only one classification is represented at a time. The previous section showed that this is not a realistic view. These models are all based on similar representations of taxonomy. All represent the main component of a taxonomic classification by a taxon that is placed below another taxon, and above a series of other taxa. Each taxon can only have one parent, which means that they can only appear in one classification. In order to overcome this limitation, some of these models support the definition of synonyms, homonyms, and accepted names (e.g. Pandora). The disadvantage of this approach is that the user is forced into choosing one viewpoint from which to describe classifications, and make decisions regarding the status of a specific group (e.g. they must choose which group is the accepted group and which is a synonym).

In addition, they are also name-based systems, which implies that the meaning of taxa (their circumscription as a set of physical specimens, some or all of which must be type specimens) is not clearly defined. As a consequence of this lack of information, it is not possible to design an automatic mechanism for naming or checking the name of taxa. Therefore, regarding criterion (1), they provide insufficient support for taxonomic work (e.g. a revision). It is also impossible to have an objective view of the data, as names are not objective entities. Indeed, taxonomists may attach meanings to certain names, but their point of view may differ from that of other taxonomists.

Only two models attempt to capture multiple classifications: IOPI [Berendsohn '97] (model only) and HICLAS [Zhong '96]. However, these models take an approach that does not allow them to capture properly (e.g. HICLAS) and easily (e.g. IOPI and its numerous *potential taxa*) how taxonomists work and relate their respective work, or to compare effectively classifications.

The IOPI model is a name-based model, as the working group that created it consider that although a specimen based approach would be the best approach, it is impractical at the present time because:

- A substantial percentage of taxa exists which are well circumscribed and not beset with nomenclatural problems, a vast amount of information exists that is only linked to names.
- The priority at the time of the creation of the model was the generation of a nomenclatural checklist, which is a taxon-based approach.

Therefore, although the specimen-based approach is considered a good approach, the IOPI model concentrates on the representation of taxa. Provision is however given to allow the representation of specimens. Because, as was described in section 2.1, the definition of the meaning attached to a name varies from a taxonomist to the next, it is not possible to represent a single named group once and for all. The IOPI approach consists in defining *potential taxa*. A potential taxon is a taxon identified by its

name, but also by a publication that distinguishes between the distinct meanings given to the name. That potential taxon is attached to a plant name which represents the name of the taxon.

It appears that IOPI combines the concepts of nomenclatural status of a name (was it validly published?) and the status of the taxon in the classification (is the taxon synonymous with another?). This is because all status information is linked to the potential taxon. This mechanism does not allow the representation and comparison of concurrent classifications. Moreover, IOPI requires the user to make a statement about the status of the taxon rather than allowing the system to work out that information. With IOPI taxonomists must declare synonyms and specify an accepted name for each group of synonyms. Instead, synonyms could be discovered by a process that examines specimen lists and infers which taxa are synonymous. This would avoid the artificial creation and manipulation of accepted names.

From a classification point of view, the taxa defined in IOPI may be classified under many higher rank taxa. New potential taxa are created for each distinct classification and are placed in higher potential taxa. This provides the basic mechanism for multiple classification. It is not clear however how classifications are identified and managed. In addition, a name is still attached to this taxon, which means that the name can become incorrect in certain circumstances (e.g. when a specific Species epithet is moved to a new Generic taxon). It also means that the taxonomist creating the classification must decide what the name is, and there is no provision for a system generation of names, or automatic reorganisation of classifications in order to assign the correct names. Automatic derivation of names would in addition be made impossible in most cases, as IOPI considers the representation of specimens impractical, therefore generally ignores them. Moreover, not enough information is stored in relation to the classification process, so when taxa are classified under taxa which are also multiply classified, the concept of multiple hierarchies does not appear and the different classifications are mixed together without any means to differentiate them clearly. The IOPI approach does not allow easy classification comparisons because the different taxa defined have no common reference point.

HICLAS uses a different approach to the multiple classification problem. HICLAS is based on the concept of *taxon-view*. A taxon-view is a quadruple of the following elements: taxon name, author/authority, year, and publication number. HICLAS defines two types of trees: the classification tree, where taxon-views are stored, and operational trees. These two trees are orthogonal. HICLAS represents classification hierarchies as parallel trees without classification interaction. These trees may only be related because an operational tree may link them. For example, when synonymous taxa (from a name point of view) are used, a primary taxon-view is created, and equivalent views (secondary views) are related to it. These operational trees represent the life of a taxon view by recording the operations that have been applied to it: *origination* (when it is the first of its kind), *more* (the movement of one taxon from one classification to another), *merge* (the creation of a secondary taxon by combining two existing taxa), *partition* (the division of a taxon into two distinct taxa), *promotion* (the move of a taxon from a rank to a higher rank), *demotion* (the mode of a taxon from a rank to a lower

rank), and *recognition* (the use of a taxon in a different classification with the same parent/child information). For example, the classification example in Figure 3 that lead to the creation of the new name Heliosciadium repens (Jacq.)Raguenaud. would have been represented in HICLAS as the movement of the taxon repens (the epithet name of the binomial Species name) from Apium to Heliosciadium by authority Raguenaud (Figure 5).

This approach has a clear drawback: because taxon-views in HICLAS represent both naming and classification information, the movement of repens from Apium to Heliosciadium implies that the two repens taxon-views represent the same things. However, according to the ICBN, the fact that a name appears in two taxa of higher rank (they seem to move from one to another) only implies that the two versions of the name use the same type (specimen or taxon), not that they represent the same concepts. For example, in Figure 4, taxonomist 2 and 3 have used the same names because they have used the same type specimens, but their concepts are clearly very different in terms of specimens (e.g. the white circle specimen will be called Circle in taxonomist 2's classification whereas it will be called Square in taxonomist 3's because of the precedence of types).

The comparison of taxa in HICLAS can be performed in two ways:

- The comparison can be made on the basis of names, i.e. two taxon-views may or may not have the same name, whether they are or are not in the same classification
- The comparison can be made on the basis of operational trees, where the history of two taxa can be compared and relationships between them inferred.

Although the representation of the life-cycle of taxon-views is interesting as it shows the evolution of ideas, because it is based on a confusion of the concepts of names and classifications, the information that can be extracted is not valuable. The evolution of classification in Figure 4 shows why: the use of names to record the history of concepts implies that names are understood objectively whereas their meaning in terms of specimens varies. Therefore, when a taxonomist records that one taxon has been moved from a group to another, the assumption is made that the two taxa, identified by their name are the same. But it was shown that this assumption is meaningless.

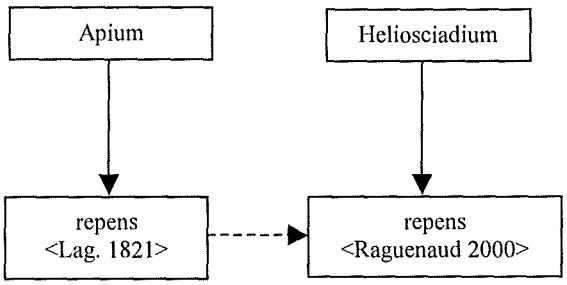


Figure 5: HICLAS example

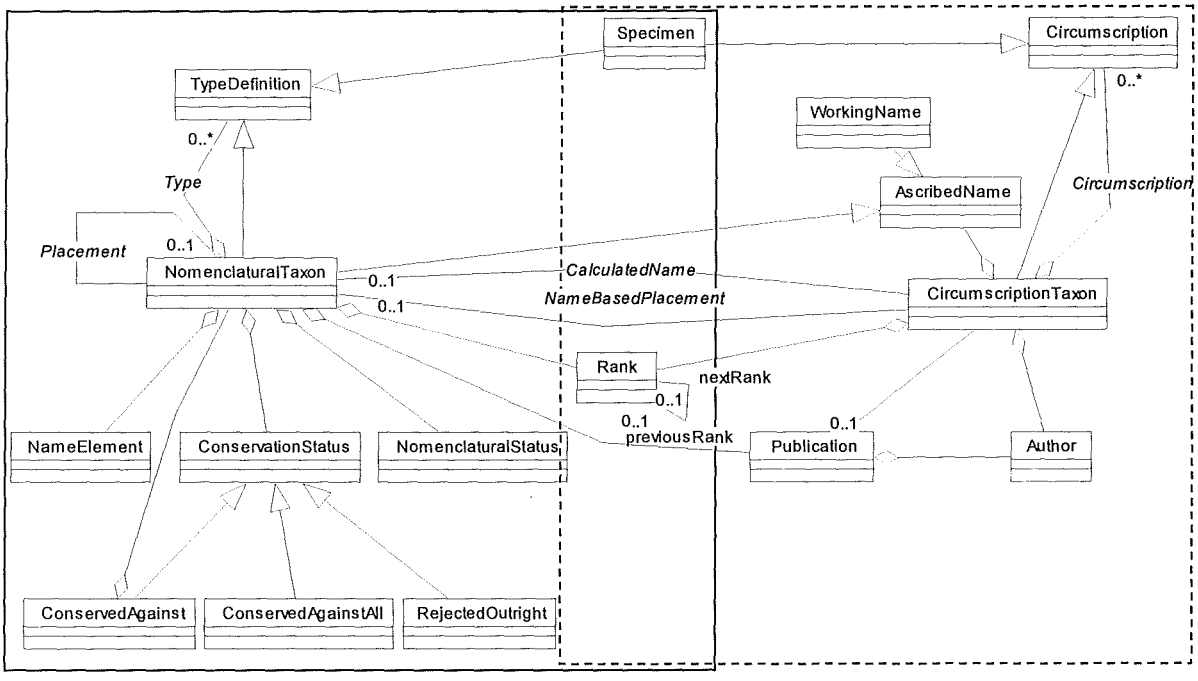
2.3 Accurate taxonomic model

Sections 2.1 and 2.2 have shown that taxonomy is a complex domain to represent. They have also shown that existing taxonomic models do not appropriately capture the data used in taxonomy and the processes applied to it. The Prometheus project (http://www.prometheusdb.org) has therefore been set up and has undertaken the definition of a new model of taxonomy that would faithfully capture the data and the processes associated with plant taxonomy.

In order to capture multiple classifications and support taxonomists' work, three features appear to be necessary in the taxonomic model:

- Names must be separated from the classification so that many taxa may share the same name or a taxon may have different names
- Names must be derived automatically so that the name of a taxon can be generated according to a classification and the ICBN. This implies a separation between the processes of classifying and naming, and the use of type specimens.
- The classification hierarchy must be able to handle multiple specimen classifications so that a specimen may be classified under many different taxonomic groups

Figure 6 shows the Prometheus conceptual model of taxonomy. As it is only a conceptual model, information is missing or is not described in detail. In this model, nomenclatural and classification information have been separated in order to emphasise the distinction between the two processes, and in order to allow the representation of multiple overlapping classifications.



Nomenclatural side Classification side

Figure 6: Taxonomic model (from [Pullan '00])

The name side of taxonomy is captured by *Nomenclatural Taxa* (*NT*). NTs are complex names that not only are a name, but also contain all necessary information to uniquely identify them. An NT is a unique combination of many pieces of information such as the place of publication of the name, the type of the name, its rank, its nomenclatural status (whether it was validly published or not), and its rejection/conservation status. The classification side of taxonomy is captured by *Circumscription taxa* (*CT*). A CT, is a composite object constituted of a publication, an author, possibly ascribed and calculated names (NTs or WorkingName objects), and of a set of circumscription items that can be specimens or other CTs. The common elements between the two sides of the diagram are specimens and ranks. Indeed, the ICBN specifies that specimens (type specimens) should be used in order to derive names from classification information. Therefore, the crossing from classification to names is performed through them. Ranks are also common elements because taxa on both sides should be defined at the same level. For example, a CT at rank Species should only receive as its name an NT defined at the same rank.

In the context of a revision, when a classification is represented, CTs are created and sets of specimens or other taxa are attached to them through the circumscription relationship. This creates a hierarchy of CTs whose order follows the order of ranks (or a subset thereof). CTs should not be named before the end of the classification process because it would influence taxonomists and force them into making decisions about their specimens as a result of their understanding of names. Working names are therefore used in order to represent the CTs without nomenclatural information. Once the complete classification has been created, names can be derived from the available data using type specimens, as explained in section 2.1. Calculated names (NTs) are thereafter assigned to CTs.

In the context of representation of existing classifications (i.e. historical data), CTs are represented with name information because this information is already available. Therefore, working names are not used to serve as handles for CTs, and ascribed names are specified to represent published names. For example, in the example in Figure 3, once the names have been derived from the classification data, they are published and become unchangeable. However, because of the presence of historical data or because of mistakes (misspelling, selection or publication of the wrong names), published names are not always the names that should be derived in the current state of knowledge or according to the current rules of nomenclature. In order to represent that published name, taxa may have therefore an additional name called the ascribed name.

Unlike most models of taxonomy, the Prometheus model does not represent accepted names and synonymous names. Indeed, the representation of such synonyms is the result of the representation of a single classification from which the world is seen. In Prometheus, because the model represents all classifications without priority, synonyms can be detected in two ways:

- Name-based synonyms are located by comparing the names attached to CTs. This is the approach taken by other taxonomic models. In this approach, two groups that have the same name are said synonym.

- Specimen-based synonyms can be inferred from the description of the circumscription of CTs. This allows a more reliable detection of synonyms, as previously undetected synonyms can be located. For example, a single specimen overlap between two groups appearing in different classifications may indicate a misplaced specimen or confusion in the groups.

Regarding the limitation raised in the IOPI model about incomplete nomenclatural information, which is often promoted as the reason why a specimen-based approach to taxonomy is not practical, the Prometheus model forces taxonomists to typify (through lectotypification) existing names that have not been typified by their author. As typification is now the norm in taxonomy, it is sensible to ask taxonomists, that use names published before typification was the norm, to elect types for their use of the names (as lectotypes for example). This way of working with types is supported. It is also sensible because when these names have been published, although no type was elected, representatives of the names were generally given as representatives. These representatives can now become taxonomic types without changing the intent of the creator of the names (as some of the specimens that were chosen as representatives become types).

Note that this model mainly applies to new classifications. Historical classifications can be represented as current classifications when type information, and possibly the whole specimen information, is available to the taxonomist. In this case, taxa are defined as sets of specimens and hierarchies of taxa form classifications. These classifications are as objective as possible and reliable.

When specimen information is not available, for example for old classifications, the taxonomic database should allow the representation of classifications based purely on taxa (no specimen information is available). This approach however has an important limitation: the derivation of names, according to the ICBN, is based on type specimens. If this information is not available (e.g. if a classification was published before 1751), the system could not derive names automatically or check that some of the actions taken by the user are valid (some actions would still be automatically checked, e.g. the respect of the rank hierarchy).

The Prometheus taxonomic model is therefore the first accurate model of taxonomy.

2.4 Requirements

The description of the process of taxonomy of section 2.1 can be analysed in order to extract the inherent properties of the data and processes that a database must support in order to provide the necessary features for a taxonomic database specifically and classification applications in general. This section describes these properties and draws the consequences for the database system. The requirements are of two natures: taxonomy-motivated requirements and database-motivated requirements.

2.4.1 Taxonomy-motivated requirements

Taxonomy related requirements arise from the inherent structure of taxonomic data and the nature of the processes applied to it.

1. **Tree/graph structure**: the most obvious property of taxonomic data is that it appears to be constituted of a set of hierarchies (directed connected trees). This set of hierarchies, when interconnected via specimens forms a directed graph. Therefore, a database system that would be used to support taxonomy would need to support the definition of graphs. In order to support the process of a revision, it is also necessary to be able to copy classifications so that they can be extended to form new versions or form the basis of a new view of a plant group, therefore be able to see graphs as an entity and manipulate that entity as a whole.
2. **Directed graphs**: the fact that these graphs are directed is very important, as the levels in the hierarchies are controlled by the rank hierarchy that are standardised and ordered. In addition, these ranks are not compulsory, therefore the hierarchies should not be fixed. For example one classification may use the ranks Genus, then Sub-Genus, then Species. But as ranks between Genus and Species may be ignored, another classification may use only Genus and then Species. It was also said that the elements of these hierarchies are ordered, as groups can be seen as nested within others. The direction of graphs is therefore vital information, although the depth of these graphs may vary.
3. **Multiple classifications**: because taxonomists consider different criteria the most important criteria on the basis of which they build their classifications (e.g. because of opinions, changes in technology), the same set of specimens can be classified in multiple ways over time. Since these different classifications are equally valid, classifications in taxonomy form multiple overlapping classifications. It is therefore important that any system used to model taxonomic data is able to capture multiple overlapping hierarchies.
4. **Traceability**: as a consequence of multiple classification of specimens, traceability is fundamental. Traceability allows the explanation, in the data, of the motivation for a particular classification. For example, a taxonomist should be able to explain why a particular *taxon* has been placed in another.
5. **Composite objects**: plant names are stored in the form of Nomenclatural Taxa (NTs), i.e. names with their types and publication data. These NTs are unique combinations of their various constituents (according to [Pullan '00], publication, author, type, placement, epithet, and rank) and are only meaningful as the set of these constituents. Therefore complex entities must be manipulated so that complex concepts such as NTs are meaningfully used. This implies a clear distinction between the internal representation of a concept and its relations to other entities in the system. In addition, the representation of composite objects is necessary to provide full generic support for classification structures. Indeed, classifications are not always built on a part-of relationship. They can be built for example on functionality relationships (objects in the classification hierarchy use other objects and are used by other objects), or containment/membership (that is not necessarily a part-of semantics [Odell '94b]). These

classifications therefore need to make a clear distinction between what is perceived to be part of the objects that are classified (their definition as aggregation of values and other objects), and what is considered to be outside of the objects, including the classification relationship.

6. **Population-based classifications**: As presented in section 2.1, plant taxonomy classifications are population-based classifications, i.e. they categorise populations of *specimens* in a hierarchy of concepts. The concepts that are used for describing the categories are entities that have a life and an importance in themselves, they are not entities that only classify other objects, as it is often the case in other classification problems. These classification concepts are therefore objects/instances (e.g. names with their publication, authority, taxonomic type information, and their rank or level) that a taxonomist manipulates and publishes, even if no classification exists. These classification concepts are very volatile and can be redefined (republished with a new taxonomic type for example) or moved in classifications during the process of a revision.
7. **Rôles**: Specimens, when they are used in publications for nomenclatural purposes, become type specimens. NTs may also be used as types. The fact that a specimen has become a type specimen in the context of an NT changes its behaviour. It can from that moment be used for deriving names from classifications. However, being a type is neither a property of an NT nor a property of a specimen (especially since specimens can be promoted to or demoted from the status of type). Entities may therefore acquire rôles and these rôles may alter the behaviour of the entities or of the system towards these entities. In object-oriented modelling (Figure 6), the representation of the relationship between these objects is essential and is information in itself.
8. **Rules/constraints**: in addition, the way taxa are named varies according to the ranks studied (see ICBN). For example, Species names and below are multinomial names, whereas above the Species rank names are monomial. These rules may also change over time. It is therefore important that a taxonomic system offers flexible ways to represent these rules and support their definition fully.
9. **Recursive behaviour**: finally, the extraction of the specimens that are to be considered for the calculation of names to be attached to defined circumscriptions involves the exploration of the classification hierarchies, whatever the ranks and the number of ranks used in each classification hierarchy, until specimens are found. The set of all specimens is then studied and type specimens extracted. From this list of type specimens, potential NTs are selected (by traversing nomenclatural hierarchies in a bottom-up fashion), and the oldest validly published NT is selected as the name of the circumscription CT. The exploration of classification and nomenclature hierarchies must be recursive and flexible. The recursive behaviour of the system is also necessary in order to compare classifications. Indeed, as was presented in section 2.1, the only objective fixed points of classifications are specimens. So in order to compare two taxa, whatever their ranks, it is necessary to recurse down classifications until all specimens related to these taxa are collected.

Requirements 1, 3, 5, 6, 8, and 9 can be generalised to any classification mechanism that faces the multiple classification problem. Indeed, these mechanisms all require the ability to describe trees/graphs in order to represent classifications. They also require the ability to capture multiple concurrent classifications and composite objects if they need to classify complex information.

Requirement 8 is a general requirement for many databases, but particularly for databases that need to enforce a code of practice. Requirement 9 is also necessary in order to manipulate and explore the classifications. Requirements 2, 4, and 7 may not always be needed by classification mechanisms. Directionality may only be important if this is inherent to the data as it is in taxonomy. However, directionality does not impair mechanisms that have no use for it. Requirements 4 and 7 may not be necessary for other domains.

2.4.2 Database-motivated requirements

Database-motivated requirements are requirements that are inherent to a classification database system but are independent from taxonomy itself.

10. **Integration with existing system**: specialist databases already exist, and it would be a plus if the system designed by this work could be integrated with as little changes to an existing system as possible.
11. **Generic classifications**: unlike other classifications, plant taxonomy classifications are not is-a classifications. Indeed, the concepts used as classifiers are abstract names (*taxa*), e.g. *Apium*, and it would be wrong to say that *graveolens* at rank *Species* is-a *Apium* at rank *Genus* because it has been placed in *Apium*. Although ranks are central to the process of classifying, it would also be wrong to say that a *Species* is-a *Genus*. Plant taxonomy classifications are rather placement classifications, i.e. *graveolens* has been placed in *Apium* (and therefore becomes *Apium graveolens* after application of the ICBN) for some reason that the taxonomist can describe. It is therefore important that the model supports any kind of classification semantics.
12. **Orthogonality of classification and classified information**: in order to support the processes associated with taxonomy, the classification and the objects classified should be clearly distinct and identifiable. For example, some operations manipulate classifications (e.g. for reorganisation), and they would be impossible if the classification information could not be clearly and generically identified. This also allows the classification of any kind of data.

Requirements 10, 11, and 12 are general requirements that apply to any classification mechanism. Indeed, integration with an existing system is always a plus if such a system already contain a lot of information. In addition, the genericity and orthogonality of classifications make such mechanisms more suitable to other application domains and should never hamper classification mechanisms.

2.5 Conclusion

This section has described the purpose and the workings of plant taxonomy. Plant taxonomy is a challenging domain, as the processes involved in the various activities connected with taxonomic work (e.g. representation of existing knowledge, revisions, extraction of new knowledge by the study of

available information) are not trivial. They involve the collection of information on existing classifications, the application of the ICBN, and the derivation of names based on provided data according to complex rules.

Most models of taxonomy are inappropriate and few offer the basic features necessary (although they are limited, e.g. multiple classification in IOPI) to represent taxonomy. All of existing models take a simplistic or flawed view of taxonomy, often reducing it to the representation of a single subjective point of view on the diversity of the world's organisms. The resulting models are therefore limited in the services they can offer taxonomists (e.g. the support of the process of revision requires the ability to be able to experiment with classification groups and their expected names), and ambiguous or non-rigorous in their representation of the world (a specimen might not have only one name).

A model of taxonomy representing exactly how taxonomy works has been explained. That model emphasises the distinction between the two main processes that constitute taxonomy (especially in the case of a revision): the process of classification, and the process of naming. These two processes and their data are therefore represented as distinct hierarchies whose connection points are specimens. This approach allows an objective view of taxonomy where taxonomists are not forced into making decisions when they can only guess the meaning of the data they handle (e.g. on the meaning of a particular name according to different people). It therefore provides the basis for a better representation and support of taxonomy based on objective observation and available data instead of judgements and opinions. In essence, it provides a specimen-based view of multiple overlapping classifications.

The analysis of that model allows the extraction of the requirements that a computing system built to support taxonomic work ought to fulfil. These requirements include a graph-based aspect (to represent interconnected hierarchies), a recursive aspect (to explore and extract information from the graphs), the ability to model complex data (e.g. composite objects), and a rule-based aspect (for the enforcement of the ICBN).

Database technology can thereafter be applied or developed to automate the common processes and move the responsibility to ensure that the data is correct from the user to the computer. However, as the next section will show, existing database technology suffers from clear limitations that can only be overcome by the definition of an extended database model.

3 Classifications and existing database systems

This chapter reviews classification mechanisms in computing and existing database in relation to classifications.

3.1 Classification mechanisms

Classifications in computing are often understood as object/class classifications where object-oriented classes are organised into an inheritance hierarchy according to their content and their inheritance relationships to other classes. These classifications are generally straightforward (although they might allow exceptions [Borgida '89] or contradiction [Borgida '88]) especially in the sense that they do not change too much over time. As was explained in section 2, although taxonomic classifications do not change once they have been published, they are likely to change when revisions are attempted within the revision process. They are also based on opinions, substantiated by observation of characters for example, not on properties or relationships with other groups (unlike inheritance for example). It is therefore clear that one must make a distinction between what is a computing classification (class hierarchy) and a taxonomic classification.

Some classification mechanisms have been described in the literature for biological classifications. For example the *Materialization* relationship [Pirotte '94] and power types [Odell '94a]. These computing classification mechanisms appear at two levels: modelling (how to conceptually represent a classification) and implementation (how can these mechanisms be handled in a computing system). The distinction between these two levels is not always clear, as some implementation models implement modelling classification mechanisms as a feature of the implementation (e.g. materialization). Others are modelling mechanisms (e.g. power types). These are of interest for this work, as a classification mechanism is necessary to handle taxonomic information.

This section discusses their ability to represent taxonomic classifications when the full complexity of taxonomy is considered.

3.1.1 Materialization

The materialisation, or classification pattern, combines classes and metaclasses to form classification concepts. Each classification class is "two faceted", i.e. it is comprised of an instance of a classification metaclass and a subclass of a more general class. These concepts therefore become classified by the meta-class. Figure 7 shows such a classification: A Car is a kind of vehicle (it inherits from Vehicle) but has a fixed licence class. The licence class is an instance of the meta-class V_type and is attached to the Car class. In practice, the instance aspect of classification classes act as class constants. The

concrete class has only one abstract meta-class, and each meta-class can have many concrete representations.

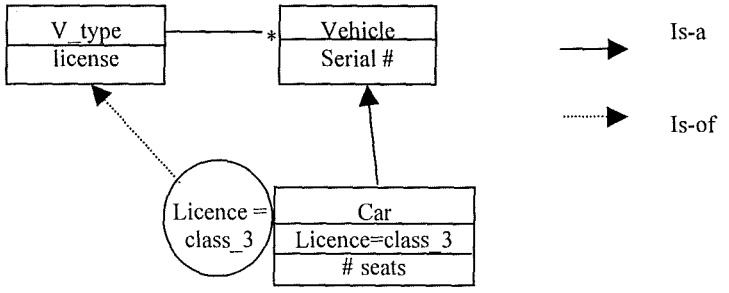


Figure 7: Materialization example

This approach, although powerful to describe some sorts of classifications, has serious limitations when plant taxonomy is considered (even though biological classifications are cited as an example). The classifications created using *materialization* are relatively static classifications, as they involve only the creation of classes as classifying entities (e.g. Car). In taxonomy, the concepts that are classified and that classify are instances, not classes. In addition, moving the classification process to the class level would generate many problems due to the dynamic aspect of plant taxonomy. The result would be a large number of classes (dozens of thousands), and enormous reclassification problems (e.g. class/instance migration, *materialization* modifications with knock on effects on existing classes and classifications). In addition, *materialization* does not provide support for the multiple reclassification on the same concepts (*taxa* or *specimens*) over time. First of all, the concrete classes have only one abstract class, which makes it impossible to classify classes in distinct manners: subclasses of the concrete class have always an instance of a single meta-class. Secondly, if this restriction were removed, there would be no way to trace the reasons for the distinct classifications. Finally, [Pirotte '94] shows that population-based and category-based classifications are different matters although they are related. As was said earlier, plant taxonomy classifications are population-based classifications, therefore *materialization* is not suitable.

3.1.2 Power types

Another example of classification mechanism is power types [Odell '94a]. Power types allow the definition of classifications of instances into other classifying instances. Figure 8 shows an example of power type where instances of Trees (real world trees) can be classified into instances of Tree species. In fact, instances of Tree species are subtypes of Tree. Unlike *materialization*, power types are an instance-based mechanism. This supports a dynamic approach that *materialization* makes cumbersome by manipulating classes.

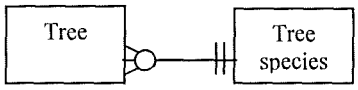


Figure 8: Power types

However, some problems regarding plant taxonomy ought to be noticed. First, objects are classified in only one category. This means that it would be impossible for a plant to specimen be a member of two species (in different classifications). Although it is true in the context of a single classification, it is not in the context of multiple classifications. Secondly, this pattern does not support the definition of multiple classifications, as it is not possible to identify distinct classifications (except by embedding classification information in classifying object, which would mean that they can only be part of a single classification). Thirdly, no traceability mechanism is provided, therefore it would be impossible to know how and why a particular specimen or classifying group has been placed where it is.

3.1.3 Conclusion

The level at which classifications should be managed is debatable. Some approaches insist on class or meta-class level classifications (e.g. [Pirotte '94]), others on instance-level classifications (e.g. [Odell '94a]). The difference is significant: the implementation of one or the other may have an impact on the performance of the overall system, or could more or less easily be integrated with a query language (e.g. query languages are generally less able to query classes or meta-classes than instances).

In addition, database practicalities should be considered. As was said in section 2, taxonomy is a discipline that always changes, either in the context of a single revision, or in the broader context of all revisions. It is a well-known fact that schema evolution is not an easy task. (e.g. [Bertino '93]). Changing the definition of classes or their relationships with other classes requires following many rules, possibly re-computing classification hierarchies, and this sometimes is simply impossible. The Materialization approach in this context is likely to generate many problems.

3.2 Database systems

This section reviews existing database models (relational, object-oriented, graph based, and extended object-oriented) and their query languages, and studies their ability to satisfy the requirements for classification mechanisms and databases highlighted in section 2.4. A more complete overview of database systems is presented to the interested reader in [Raguenaud '01].

3.2.1 Relational systems

Most taxonomic databases use an underlying relational [Codd '70] storage system (e.g. Pandora [Pankhurst '93], Alice [White '93], BG-BASE [Walter '93], Brahms [Filer '94]). Relational systems are therefore a sensible starting point to review how classification databases could be implemented. This section reviews the support these systems offer regarding the requirements expressed in section 2.4.

1. **Tree/graph structure**: as was said in section 2, hierarchies are at the centre of taxonomic data. It is vital that a database used for taxonomy is able to store and manipulate hierarchies. The classical relational model [Codd '70] is a flat model that does not easily represent this concept of hierarchy. Indeed, because everything is represented as tables and all the treatments occur at the table level, it is not possible to describe accurately structures that span many tables or represent recursive structures such as trees. However, graph structures can be represented with tables if nodes and edges are both stored in different tables. Although this representation may be sufficient in some cases, it is not satisfying regarding taxonomy because these graph structures are not understood by the system as specific structures. Indeed all the database management system handles are indistinct tables. Extended (Third Manifesto [Darwen '95], which does not appear to have been implemented) and object-relational (e.g. Postgres [Stonebraker '90], Oracle [Oracle '01]) models can support nested relations by providing extensible types, allowing a simple form of hierarchy representation. For example, one of the attributes of a relation can be the relation itself or another relation. However this representation is still very simple and would not support for example weighted graphs, as relations would be nested but the nesting would not convey information. Therefore only the simplest form of trees/graphs could be captured. Finally, as all these options are too simple to accurately represent trees/graphs, they lead to complex processes/manipulations and possibly integrity constraint problems. These complex processes would have a negative impact on the efficiency of the overall system.
2. **Directed graphs**: directionality is necessary to represent the way data is articulated, i.e. which entities are below which and in which direction information flows. If hierarchies are modelled as explained above, direction can only be represented by the choice of names in relations that play the rôle of edge. However, this does not clearly describe the direction of relationships as it relies on user understanding of the names of the attributes of a relation. The system itself has no concept of edge or direction and tables are manipulated irrespective of their semantics (e.g. edge or node). Although this may not be a problem for the representation of data, it may influence the way queries and applications are written, as the user or user applications need to have knowledge of the mechanism that handles directionality and need to emulate them. This would increase the cost of developing applications and the risk of inconsistencies.
3. **Multiple classifications**: since the relational model does not support the definition of hierarchical structures as a feature of the system (they have to be managed by the user application), it cannot manipulate overlapping hierarchies. However, one feature offered by most relational systems,

views [Cannan '92], may be of interest for the representation of multiple classifications. Views would allow the filtering of relations according to specific criteria in order to present a partial view of the information to the user, e.g. a single or several classifications at a time. The idea is seductive but practical problems arise: the definition of views may be very complex, as a single classification contains a high number of different concepts (e.g. names, taxa, types, publications, authors). The selection of all these concepts would require an important number of queries (at least one for each table of interest), which would not only be hard to express (new relations may need to be created), but would also have a negative impact on the performance of the system. Combined with the difficulty to capture trees/graphs, it appears that multiple classifications in the context of a relational model are not practical.

4. **Traceability**: traceability is an important feature that allows better understanding of author motivations. Depending on the approach chosen for representing trees, traceability may be supported by relational models. For example, if relations are used to represent edges (instead of nodes), then these edges can contain information that can capture classification motivation, therefore traceability. However, embedding information mainly on edges has the disadvantage of not being able to represent all information. For example, the representation of a relation that captures the concept of person requires many attributes and placing them on edge relations may not be possible, e.g. would that mean that an edge to another concept would contain the concept on the edge, which would at least require a nested relational model?
5. **Composite objects**: composite objects allow the definition of clear boundaries for objects, which permits better understanding and integrity constraints. Because the structure of tables is simple in classical relational models and the only way to describe data is the creation of simple tables, a lot of the semantics of the data is lost in the process. For example, although some Entity-Relationship notations offer inheritance as a basic feature, it is impossible to represent in the database. Complex mechanisms must be used in order to map the design to the database. These mechanisms are not interpreted by the system itself, and the user application must be designed to handle such information. For example using foreign keys between a table and another to represent inheritance and interpret this link in the user application, or embed all the attributes that all the classes in a hierarchy can have in one table and differentiate the nature of the entities through flags can only be understood and handled by the user application. Extended relational models deal more efficiently with this problem by offering inheritance (single in the case of the Third Manifesto). Other design semantics such as the definition of aggregation relationships are lost in the implementation, which means that composite objects cannot be properly represented. This creates an impedance mismatch between these two steps of the application development and makes programming and maintenance harder.
6. **Population-based classifications**: population-based classifications use members of the classified population as categories, it is the case in taxonomy. Population-based classifications requires the ability to manipulate concepts at any level of the classification. The fact that the relational model is a very simple model would allow this, as any node in a classification would be represented by a relation and treated as any other relation, any participant in a classification could be a relation.

7. **Rôles**: rôles give objects the ability to be multi-faceted, as some objects are in taxonomy (e.g. specimens). Relational databases do not support mechanisms that would allow the acquisition of rôles *per se*. However, it would be possible to create multiple views on a relation that would display different attributes according to the rôle intended for each view. For example, a view storing all specimens that are referred to by another table as a type could describe a type specimen view. However, this technique would not easily support the changes in behaviour that must occur when an entity changes rôle. This is because the various views would not be mutually exclusive, and it would be left to the user application to manage the various behaviours and spot when a particular entity has switched rôle. In addition, more instances and more manipulations of instances may lead to referential integrity problems (e.g. if a foreign key refers to a specimen in the Specimens relation and this specimen is promoted to type specimen and moved to the TypeSpecimens relation, would it be removed from the Specimens relation? How would all the foreign keys be updated?).
8. **Rules/constraints**: rules and constraints are necessary in general in databases, but particularly in taxonomy where a code (the ICBN) must be enforced. Relational databases support a powerful and well-adapted constraint mechanism (e.g. SQL92 [Cannan '92]). However, this mechanism is tailored to the structures that are definable in the database, therefore suffer from limitations due to the fact that it is designed to manipulate simple structures, not graphs or complex objects. This mechanism may therefore not be able to capture the rules necessary to fully implement the ICBN.
9. **Recursive behaviour**: recursive behaviour is necessary to manipulate recursive or graph-based structures like taxonomic data. SQL also has limitations regarding the operations that can be attempted on the data stored. Indeed, because the operations are centred on tables, operations are "localised". In standard SQL, it is not possible for example to describe a recursive exploration of the database because this involves the description of an unlimited or unknown number of joins. It is therefore impossible to extract information such as the set of all nodes that appear at any depth below one particular node. The only solution, unsatisfactory, is the explicit description of all the joins necessary to traverse the recursive structure. This solution would make the writing of queries harder (the depth needs to be known in advance). In most cases, these queries would be replaced by programs that would allow reuse and depth independence, but would make the evaluation of such queries less efficient (programs would not be as optimised as queries). Proprietary and research extensions exist that attempt to overcome some of these problems. For example, the Oracle relational/object-relational database offers an limited operator that allows the repetitive traversing of a set of tables, thereby providing a means to emulate recursive behaviour. Other research systems, e.g. SQL* [Koymen '93], Resql [Wood '90], provide mechanisms that allow the description of recursive statements via DATALOG-like rules to describe graph traversals or via path expressions respectively. However, these extensions are not standardised, not common, and often very limited (SQL* only supports recursive rules in insert statements, does not allow nested recursive statement, and does not provide depth control operators).
10. **Integration with existing system**: integration means that the addition of new mechanisms are made as smoothly as possible, i.e. without changing the way existing applications work. Relational

database models would not represent classification information accurately, as classifications would necessarily be of a generic type of relation (i.e. classification relations would not be recognised by the system as a type of relation that needs to be treated in a specific way). However, these relations would be smoothly integrated in the database schema as these relations would comply with the underlying relational model.

11. **Generic classifications**: generic classifications are necessary to support unforeseen applications or non-taxonomic applications. Relational models propose relations as the basic entity. These relations do not represent is-a or is-of relationships, therefore relational models would be unable to capture is-a or is-of classifications. All classifications in a relational database would be generic classifications. This can be acceptable in the case of taxonomy because taxonomic classifications are not is-a nor is-of classifications, however, it may be a severe restriction for other kinds of classifications.
12. **Orthogonality of classification and classified information**: orthogonality provides independence between classified data and classification mechanism. This is an advantage for supporting new applications and classifying existing taxonomy information. Because relational models offer relations as the only piece of information, it would be hard to represent classified entities distinctly from classification information. The only distinction that could be done would be on the basis of names, not semantics. For example, user applications would need to know that the relation named X is part of the classification information, whereas the relation named Y is part of the classified information.

This section has shown that the relational data model and its extensions, because of their simplicity, would allow the integration of new information smoothly. However, that same simplicity of model renders relational systems unsuitable for classification applications. One limitation is the impossibility to represent appropriately trees/graphs, and in particular complex graphs such as weighted graphs where information is necessary on edges. Any possible solution would either result in overly complicated user models that would make user applications more complex and querying harder, or user models that would not capture all the available information. In addition, high level concepts such as composite objects would not be managed in a satisfactory manner, and in all cases semantics would be lost in the implementation in the relational data model. This section has also said that views are an interesting mechanism that would provide a means to emulate some of the required features (e.g. multiple views on a taxonomic database to extract individual classifications, and simple rôles). However, the simplicity of the model would limit the ability of such approaches to support fully these features (e.g. integrity problems may arise with migration of rows between tables to emulate rôles). Finally, the query languages proposed for these models suffer from limitations regarding their interpretation of semantics (e.g. the non-existence of semantic relationships in the model) and regarding recursive structures (either because recursion does not exist at all, or exists in a limited manner).

3.2.2 Object-oriented systems

Object-oriented systems have been less often used to support taxonomic applications. However they offer higher-level features that may be of help to design classification systems in general, and taxonomic classification systems in particular. They have been designed to overcome some of the shortcomings of relational systems listed in section 3.2.1 (e.g. [Fishman '89] [Atkinson '90] [Bertino '93] [Brady '97]).

Over the years, many approaches have been proposed in order to design an object-oriented system: the design of a system whose persistent layer is a relational DB [Keller '93] [Keller '94] [Hohenstein '96] (in particular through the usage of object-oriented views [Delobel '95] [Fahl '97] [Vermeer '95] [Barsalou '91]), the design of mediators that automatically map objects to relations (used today in object-oriented applications that use ODBC), and the design of a pure object-oriented system [Fong '97] [Joseph '91] [Fishman '89] [Maier '90] [Ishikawa '93] [Peters '93b] [Scholl '93] [Hori '95] [Cattell '97] [Weiser '89] [Leung '93] [Kim '89] ['92] [Bretl '89] (with semi-automatic migration of relational data [Ramanathan '97] or not [Fong '97]).

This section reviews the support these systems offer regarding the requirements expressed in section 2.4.

1. **Tree/graph structure**: object-oriented databases can support the definition of simple object graphs. Indeed, objects and their references form a graph. This approach has two drawbacks: first it uses references to link objects, but these references are embedded within objects, therefore there is no clear distinction between objects and classification. Second, only the simplest graphs can be represented, as edges show only the existence of a link between two nodes (as a reference), without further information. Weighted graphs are impossible. Although object-oriented databases can represent graphs of objects using references or using inheritance, their query language does not support manipulation as a graph. Embedding the necessary specific code in the application could provide support for this, however, this would also increase development and maintenance overheads. An alternative approach would be the representation of graph edges by normal objects that user applications would recognise as edges. This has the advantage of providing a means to capturing complex graphs such as cyclic or weighted graphs. However, this would be limited in the sense that edge objects would not be recognised as relationships or references by the database system. Therefore, the insertion of an additional level of indirection would make user applications more complex and may lead to integrity problems (updating an object edge is more complicated than updating a reference). In addition, writing queries would be made harder by the additional level of indirection and the necessity to select these edge objects explicitly if they are required in the query result.
2. **Directed graphs**: directionality allows the representation of the direction of information flows. In hierarchies, it can be the direction of the hierarchy. Directionality is inherently supported in object-

oriented databases by object references/pointers. However, in the case of complex relationships (in the ODMG sense), the direction of the relationship is lost because of the utilisation of inverse references/pointers to support the various cardinalities. One has therefore to choose between cardinality control and directionality, which is not desirable.

3. **Multiple classifications**: some object-oriented models propose a form of multiple classifications: classification of types by distinguishing types from classes [Irani '93] [Norrie '95] [Joseph '91] [Hori '95] [Borgida '88] [Cattell '97] (sometimes with exceptions [Borgida '88] or behaviour classes [Irani '93] [Peters '93a] [Ozsu '95]). This multiple classification mechanism is however too limited to be applied to taxonomy. First, it is an is-a classification mechanism, therefore other types of classifications could not be handled. Second, classification information is implicitly represented, therefore would limit the ability of the system to capture complex tree/graph information (e.g. weighted graphs). Third, these classifications are not generic graphs, as they cannot be cyclic. Fourth, changes to the classes may lead to important class reorganisation problems such as schema evolution ([Banerjee '87] [Crestana-Taube '96] [Bertino '93]). They would also lead to very large schemas that could become unmanageable (a schema for a flora may contain hundreds of thousands of groups). It is not clear how overlapping classifications can be represented in a different way in object-oriented databases. Indeed, as the representation of graphs rely solely on the definition of objects as nodes and references as edges, it is not possible to describe the specifics of the relationships that would capture the overlap. Additionally, this may limit the ability of the system to be dynamically reorganised (this would imply schema, which would introduce problems, limitations, and important overheads). View mechanisms allow the definition of different appearances for objects and classes [Schek '91] [Rundensteiner '92a] [Bellahsene '97] [Santos '94]. The view mechanism can maintain a global schema that contain all classifications, and extract individual classifications to present them to the user [Rundensteiner '92a] [Bellahsene '96] [Kim '95]. Additional classes (involved in the is-a overall classification) can be created in the process in order to provide a consistent overall database schema [Rundensteiner '94]. Views are more flexible than schema-based approaches, as they can be created with the query language [Deßloch '92] or a view definition language [Rundensteiner '92b], and some view mechanisms allow reorganisation and possibly automatic class integration in existing schemas and views [Rundensteiner '94]. In the case of an ongoing revision, the cost of creating and modifying views, even if they are materialised [Kuno '95a] [Kuno '98], may be too high to allow taxonomists to work. For example conflicts must be detected and resolved, if the view mechanism supports this feature.
4. **Traceability**: traceability is necessary in the context of some application domains (e.g. taxonomy) in order to understand the motivations behind other taxonomists' work. Whether one of the mechanisms described above is chosen or not, traceability is an unresolved issue in object-oriented classifications. Indeed, nothing allows the description of the motivation of choices or mechanisms that lead to the creation of particular classifications in both the schema-based approach and the view-based approach.

5. **Composite objects**: composite objects allow the clear representation of boundaries of objects, therefore their easier management and the enforcement of integrity constraints. Object-oriented databases provide the means to describe composite entities. This is achieved through the use of type casting, inheritance, and encapsulation. However, the description is limited by the available semantics. For example, in most object-oriented databases it is not possible to describe the semantics of the composite entities, i.e. what the relationship of the component elements is to the other elements in the object and to the composite object as a whole [OMG '99] [Bock '94] [Bock '98] [Rumbaugh '91] [Odell '94b] [Rumbaugh '88] [Rumbaugh '94] [Rumbaugh '95] [Kim '87] [Henderson-Sellers '97]. Notable exceptions are Orion [Kim '87] that uses constraints on objects to support a form of relationship semantics, and Bertino [Bertino '98] that offers an extension to ODMG's ODL that provides means to define some integrity constraints (dependence/independence, sharing/non sharing) associated with the references of the ODMG model. This in addition leads to improper class design where relationships that should be independent are embedded in object types [Norrie '93].
6. **Population-based classifications**: population-based classifications require the manipulation of generic entities as classifying objects, which is necessary in plant taxonomy. The schema-based approaches suggested earlier also have the disadvantage of leading to category-based classifications, where the classifying objects are categories that have little importance or information in themselves (classes), and the classified objects are assigned to one of more category. It was said in chapter 2 that the classifying objects, the taxa, have a rôle to play independently of classifications and are complex objects. Representing these objects as classes would lead to loss of information (once again because of the lack of semantic relationships, especially at the class level).
7. **Rôles**: rôles provide support for objects that may change their appearance depending on their context, as specimens do in taxonomy. Rôle acquisition is partially supported by object-oriented principles. Indeed, the fact that objects have a hidden state and behaviour allows them to behave differently depending on their internal state. This change of behaviour can also be the consequence of the state of external objects. More advanced rôle acquisition mechanisms exist [Gottlob '96] [Richardson '91] [Stein '89] [Wong '97] [Wieringa '94] [Albano '91] [Kuno '95b]. These mechanisms allow objects to look different depending on the view other objects have on them and allow the acquisition of properties at runtime. An important limitation of these rôle mechanisms regarding classifications is that they only act at class level, which would imply that the chosen classification mechanism must be devised to act at class level. However, it was said earlier (section 3.1.1) that this approach is too restrictive to be an option in the case of multiple classification mechanisms, especially in the context of plant taxonomy. In addition, a more practical problem is that most proposals restrict the applicability of their mechanism to very specific database models that exhibit peculiar features (e.g. Smalltalk and its dynamical late binding system). These features restrict the database systems that are available to implement such mechanisms, and may lead to a very inefficient database system (for example through the impossibility to type check queries in advance to allow optimisations and rewriting, which would be inefficient [Grumbach '99]).

8. **Rules/constraints**: constraints allow the enforcement of the ICBN and rules may help the automatic deduction of information (e.g. names). Constraints are well supported in object-oriented databases and deductive object-oriented databases (e.g. [Paton '99], [Gatziu '91], Ode [Gehani '91], [Chakravarthy '94], NAOS [Collet '94], EXACT [Díaz '91]). Most of these mechanisms use Event-Condition-Action (ECA) rules described as part of the definition of objects or not. Because some of these mechanisms integrate rules and programming language, rules are able to represent any statement and take any action (efficiency may be an issue).
9. **Recursive behaviour**: recursive behaviour is required for the manipulation of hierarchical data such as classifications. Most object-oriented query languages are not recursive (e.g. [Cluet '98] [Bussche '92] [Abiteboul '95] [Mitchell '93] [Peters '93b] [Leung '93] [Straube '90]). Only few support the definition of transitive closure (e.g. [Kifer '92]) or regular path expressions (e.g. [Christophides '96]), but this is a limited form of recursion as no complete patterns can be defined. In addition, it is not possible to define specific depths of traversal (e.g. optionality, which requires a depth of at least one).
10. **Integration with existing system**: integration emphasises the necessity to allow existing applications to exist, i.e. changes to the an existing system should be consistent and minimal. As object-oriented databases only support classifications at the schema level, the integration of additional mechanisms would require the representation of classifications at class level. Although this is not an approach that seems suitable for taxonomy (see requirements in section 2.4.2), this is the approach chosen by the only existing object-oriented taxonomic database (Object-taxon [Saarenmaa '95]). Therefore the integration of such mechanisms may not require much redesign.
11. **Generic classifications**: generic classifications are a necessity to support unforeseen classifications and support any kind of classification. Object-oriented models could only model three kinds of classifications: is-a (inheritance), is-of (aggregation), and generic classifications (reference). Is-a and is-of classifications are not appropriate to taxonomy, therefore only a generic kind of classifications could be described. This generic kind of classification would not be able to support fully taxonomic working practices as its mechanism would be too simplistic.
12. **Orthogonality of classification and classified information**: orthogonality allows the separation of data and classification information, which makes generic classifications more straightforward and allow classification of existing information. Because the only type of relation between two objects is the reference, there would be no clear distinction between the objects classified and their classification information, as this information would be embedded in their type. Therefore, the classified objects would need to be designed so as to be classified. They could however be wrapped in other objects, but these other objects would need to be specifically designed to be classified. Classification and classified objects could not be independent.

This section has shown that although object-oriented models have the ability to capture trees/graphs in different ways (inheritance hierarchies, objects and references), these approaches are limited in many ways (they are either schema-based or can only capture the simplest graphs). They also lack the ability to capture traceability and to express composite objects fully. These limitations make most object-

oriented models unsuitable to support the definition of multiple overlapping classifications, especially classification mechanisms that need to be orthogonal to the classified material. Some mechanisms provided by few of these systems present however interesting features. Views for example allow a limited representation of multiple classifications (based on schemas). Rôles are also a feature of some object-oriented systems, but their application appears limited to specific models and may have important overheads. Some object query languages also offer some support for classification work, as they provide regular path expressions that emulate recursivity.

3.2.3 Graph-based systems

Relatively recently, graph-based database models have been developed for new types of applications. Their motivations were mainly the ability to represent intuitively the data that was to be stored in the database and their ability to represent any kind of information, including for example relational schemas [Papakonstantinou '95] or semi-structured data [Abiteboul '97].

As graph models seem to be able to represent any kind of data, they may offer promising features for the support of classification mechanisms. Some of the graph models are presented here and their support for the several requirements of classification work identified in section 2.4 analysed.

1. **Tree/graph structure**: graph-based models are built on the mathematical concept of graph. In such models, everything is represented by sets of nodes and edges that represent interaction between nodes. Although the principles are the same, several approaches can be distinguished:
   - Node information bearing (TSIMMIS [Chawathe '94], Lore [McHugh '97])
   - Edge information bearing models (BDS95 [Buneman '95])
   - Node based models (GOOD [Gemis '93], Spider [Rodgers '97])
   - Edge-based models (BDS95)
   - Or both (PROGRES [Schürr '98], Telos [Mylopoulos '90], ConceptBase [Jarke '95], [Watters '90], [Tompa '89], Gram [Amann '92])
   - Flat (GOOD, Spider, Hygraph [Consens '94a], Gram)
   - Nested models (Hyperlog [Poulovassilis '98], [Zhuge '98])
   - Hypergraph ([Tompa '89], [Watters '90], [Catarci '95])
   - Schema-based (GOOD, Spider, Telos, PROGRES, ConceptBase, Gram)
   - Schema-less (TSIMMIS, Lore, BDS95) models (although they use representations of a schema derived from the data, e.g. dataguides [Goldman '97], representative objects [Nestorov '97] which can be automatically [Nestorov '97/12] [Ashish '97/12] or manually generated).

   The features proposed by each model enable it to offer support for different types of applications. For example, semi-structured data, which is based on a schema-less approach, is suitable to capture domains where the schema is very large or where the distinction between schema and data is blurred [Abiteboul '97]. Very flexible models such as Telos or ConceptBase are suitable to represent knowledge [Mylopoulos '90]. Graph-based models therefore inherently represent

hierarchies, which are a special case of a general graph (tree). By explicitly modelling nodes and edges of a graph, they allow the representation of any kind of graph, including weighted graphs or cyclic graphs. There are many different approaches to modelling graphs.

2. **Directed graphs**: direction is necessary to capture the direction of hierarchies (e.g. top to bottom). Most graph models offer directed graphs only.
3. **Multiple classifications**: graph models have not been used to tackle the problem of overlapping classifications. Therefore, none of the reviewed models offers mechanisms to represent concurrent graphs distinctly and clearly. The main limitation of these models is their lack of support for distinguishing concurrent graphs when they share nodes or edges. However, as for object-oriented models, views may offer a way to express alternative classifications. View mechanisms have been proposed in the context of graph-based databases [Zhuge '98] [Wood '90] and semi-structured databases [Suciu '96] [Abiteboul '98]. Graph views are generally made easier by the fragmented aspect of graph data where objects do not exist as such, but take the form of nodes related by edges. Graph views are essentially filtering mechanisms where sets of nodes and edges are selected. The restructuring problems that object-oriented views face do not appear. These proposals are all limited to specific aspects of graph databases: [Zhuge '98] only extracts views as sets of objects; [Wood '90] does not deal with updates and deletions; [Suciu '96] only works with join-free queries and insert statements. Only [Abiteboul '98] proposes a generic view mechanism that takes into account the specificity of graphs, and supports all operations. This view mechanism may allow the extraction of sub graphs from a general graph as classifications extracted from an overlapping larger classification.
4. **Traceability**: traceability is required in order to understand what motivated the choice of other users (e.g. other taxonomists) when they built their classifications. As multiple classifications are not supported, traceability is not possible. In particular, if views are used, then it is not possible to record the reason for the creation of views.
5. **Composite objects**: composite objects provides the means to describe the boundaries of objects and enforce integrity constraints. Graph-based models do not support clearly the distinction between aggregation and association relationships, and their fine nuances. Even when graph query languages (e.g. [Mendelzon '97]) show the necessity to distinguish between internal and external structures, these structures are never explicitly captured by the data models. The resulting models therefore sometimes appear to confuse multiplicity and semantics of relationships. Only a few provide a very limited and simplistic mechanism. For example, Hyperlog offers both references, that can be seen as associations, and nesting, that can be regarded as aggregation. Although this is not the purpose of the model, these two kinds of relationships could be used to model simple forms of semantics. They would be limited though, as nesting always implies containment, which is not a constant semantics of aggregation relationships. The model defined for WebOQL [Arocena '98] is also an exception where two kinds of relationships exist: internal and external (applied to web pages). The only model that offers extensive support for semantics is ConceptBase [Jarke '95], where rules/constraints can be attached to relationships in order to enforce their semantics. The

semantics that can be expressed are not exhaustive, as for example abstract semantics such as spatial or matter relationships cannot be captured, but are nonetheless interesting.

6. **Population-based classifications**: population-based classifications, of which taxonomy is an example, require the ability to use any object as a class. The inherent support for graph structures makes it possible to design a system that captures the volatility of classifications (taxonomic classifications in particular). Semi-structured data even offer the liberty of ignoring any kind of schema. It is therefore possible to describe population-based classifications where any participant may have importance in itself. Simply linking nodes together with edges would support this requirement.
7. **Rôles**: rôles provide support for objects that may appear differently or be treated differently by the system under certain circumstances (e.g. specimens in taxonomy). Rôles do not appear to have been studied in the context of graph models.
8. **Rules/constraints**: Rules and constraints are useful to enforce working practices (e.g. the ICBN) and deduce information (e.g. group names). As query languages, graph rules/constraints can be expressed using patterns (e.g. [Paredaens '93] [Münch '98]) and textual descriptions (declarative rules, e.g. [Jarke '93] [Mylopoulos '90], and for semi-structured data [Rousset '97] [Fernandez '99] [Calvanese '99]). These rule mechanisms offer varying degrees of complexity: some support recursive behaviour as queries [Fernandez '99], some do not [Münch '98]. Some allow complex checks [Jarke '93] [Fernandez '99], whereas others only allow simple node/edge existence/non-existence checks [Calvanese '99]. These varying abilities make some of these mechanisms unsuitable for classification rules. For example, if recursion is not supported, then it is impossible to test graph configuration beyond the simplest checks. If only a node/edge existence check is supported, then it may not be possible to express some constraints. Note that none of these mechanisms describes event management and constraint evaluation techniques. The lack of complex event detection may hamper the implementation of the ICBN, as some constraints may be hard to enforce (e.g. a change in the publication date of a specimen may lead to the necessity to reorganise whole classifications).
9. **Recursive behaviour**: recursive behaviour is required to manipulate and explore hierarchical structures such as classifications. Many approaches to querying graph structures have been proposed. Some of them are based on pattern matching (GOOD [Gemis '93], Spider [Rodgers '97], Hyperlog [Poulovassilis '98]), others are based on SQL/OQL-like languages extended to support the peculiarities of graphs ([Quass '95] [Buneman '96] [Abiteboul '96] [Arocena '98] [Mendelzon '97] [Fernandez '97b] [Abiteboul '97]), and very few use a recursive notation that exploits the underlying recursive graph model ([Fernandez '97a]). Their support for recursion takes two forms: the inherent recursive language developed for a recursive model [Fernandez '97a] and the use of regular path expressions to simulate recursivity on a non recursive model ([Arocena '98] [Mendelzon '98] [Papakonstantinou '95] [Fernandez '97b] [Abiteboul '97] [Abiteboul '96] [Zhuge '98] [Wood '90]). As many of these query languages were developed for semi-structured data, they are very flexible and powerful (e.g. they can rewrite the database graph, interpret incomplete

queries, or explore the database graph via structural recursion). These features would be of interest to a classification mechanism.

10. **Integration with existing system**: integration is motivated by the need to keep existing applications and information running after the addition of new mechanisms. As graph models support graph/tree representation inherently, the addition of classification mechanisms could be done in a well-integrated manner as long as the recipient system is also graph-based.
11. **Generic classifications**: generic classifications are classifications that may classify anything, e.g. non-taxonomic applications. Graph-based models generally only support one kind of relationship between nodes. Only a few support classes and is-a classifications (e.g. Hyperlog [Poulovassilis '98]). If relationships are classified (in the object-oriented sense), then it is possible to create classifications that are of the required type. However, if they are not, classifications become indistinguishable from any data in the system. The limitation of graph-based models is that they do not interpret the semantics of relationships. Therefore it is possible to describe is-a or is-of classifications, as well as generic classifications, but the system would not be able to interpret their meaning. For example is-a edges may be created in models that support the extension of edge types, but they would only be called is-a edges, and inheritance rules would not be enforced.
12. **Orthogonality of classification and classified information**: orthogonality, by separating the processes of information gathering and classification, provides en environment where it is more natural to create multiple overlapping classifications. As many graph-models are not node-oriented (as most semi-structured models), they do not offer any distinction between the classified objects and their classification status. In addition, as they do not support semantic relationships, they do not provide any mechanism to circumscribe the classified objects. The classification status of objects is therefore indistinguishable from their definition. Other models, that propose the description of nodes and edges, can represent the classification information and the objects, but once again as most do not support any kind of nuances in semantics, the definition of classified objects and their classification status cannot be distinguished (or only on the basis of names in user applications).

This section has shown that graph-based models offer valuable features for the support of classification mechanisms. The first is their inherent support for tree/graph structures. This allows them to capture naturally classifications. The second is the ability of several models to capture both node and edge information, which provides a means to represent complex graphs. Some, through view mechanisms, may also provide mechanisms that support the definition of simple overlapping classifications, but they are limited in their support for update/deletion operations, or their ability to extract complete graphs. This section has also shown that semi-structured systems in particular offer powerful query languages that support recursion and would be useful for classification mechanisms, and interesting rule/constraint mechanisms. However, this section has also shown that traceability and therefore the representation of multiple overlapping classifications are unresolved issues. In addition, the absence of semantics in the models makes the orthogonality of classification and classified data impossible. All

should be managed in an *ad hoc* manner by user applications. Graph models are certainly good candidates for a classification system, but they need to be extended to support all the required features.

3.2.4 Extended object-oriented systems

Extended object-oriented systems as considered here are specific kinds of object-oriented systems that emphasise relationships as well as objects (other object-oriented models de-emphasise relationships [Consens '94b]). This emphasis of relationships is justified by the fact that the traditional way of representing associations is unnatural as they require the implementation of references inside objects instead of real associations among objects [Albano '91] [Díaz '90] [Norrie '92]. Moreover, associations may have attributes of their own, and implementing them within the objects that participate in the relationship destroys the concept of association [Norrie '93] and loses the independence of the objects involved in the relationship [Albano '91]. Not only is this unnatural, but it may violate object-oriented principles, for example when constraints are necessary [Doherty '93]. These models therefore argue that there needs to be relationships as first-class concepts in object-oriented models.

The suitability of these models regarding the requirements expressed in section 2.4 is studied now:

1. **Tree/graph structure**: extended object-oriented models can represent graph structures. Their degree of support for graph structures varies: some support the definition of explicit but simple graphs (e.g. SORAC [Doherty '93]); others support the definition of more complex graphs such as weighted graphs (e.g. OMS [Norrie '93], GraphDB [Güting '94], ADAM [Díaz '90], Albano [Albano '91]). In all cases, these graphs explicitly represent both nodes (objects for GraphDB and SORAC, or unary collections for OMS and Albano) and edges (specific relationship objects for GraphDB and SORAC, or binary/n-ary collections for OMS and Albano). Although this is not the purpose of most of them, GraphDB has shown that they can be adapted to support generic graphs, therefore classifications. Note that GraphDB is also the only system that offers support for graph manipulation and exploration.
2. **Directed graphs**: directionality is necessary in classification structures to understand the flow of information (e.g. the direction from the most general to the most specific). Not all presented models support directed graphs (e.g. [Albano '91] represents undirected graphs). However most of them do. It is interesting to note that none of the models supports both directed and undirected graphs.
3. **Multiple classifications**: extended object-oriented models do not intrinsically support multiple classifications. They only support the definition of unspecialised graphs. Additional mechanisms, at the model and/or at the query language level, would need to be developed in order to support multiple overlapping classifications. Some models, e.g. OMS, provide a classification mechanism based on collections where things may appear in several collections simultaneously. The facts that the categories used in the classifications are objects that have an importance of their own and carry data (e.g. taxonomic type, publication) and these classes are classified that are surrogates for specimens are also classified (see chapter 2) make this approach unsuitable. Unlike object-oriented

and many graph-based models, no view mechanism is available for these approaches. In many cases, they are built from scratch in order to support uncommon features (e.g. GraphDB, SORAC, Albano) or are built on top of existing database systems that do not support views (e.g. ADAM). As a consequence, the representation of concurrent classifications as views of a single larger classification is impossible.

4. **Traceability**: traceability allows the tracking of decisions, thereby providing a tool to understand other users' work. Traceability is an unresolved issue. As the models presented do not support the definition of concurrent classifications, traceability is not natively supported. There is no built-in way to know why a classification has been created and why a specific object participates in it. However, some of these models support attributes on relationships (Albano, GraphDB, ADAM), therefore it can be imagined that these classification relationships also record the motivations for classifications.
5. **Composite objects**: composite objects are a necessity to record the boundaries of objects and facilitate enforcement of integrity constraints. Some of the models presented here offer some kind of support for the definition of the semantics of relationships. OMS [Norrie '93] and Albano [Albano '91] are collection based models. Special kinds of collections (binary in the case of OMS) represent associations. These associations are semantic associations that are defined independently of objects. In the case of OMS, the distinction between associations (binary collections) and aggregations (references embedded in objects) captures limited semantics of relationships. In the case of Albano, associations and aggregations are both defined as collections, and constraints enforce some of the possible semantics of relationships (e.g. constancy, dependency). Other models take a different approach and extend existing object-oriented models with specific structures that play the rôle of relationships (GraphDB [Güting '94], SORAC [Doherty '93], ADAM [Díaz '90], DSM [Shah '89]). These models do not provide semantic relationships, however, their relationships may in some circumstances carry constraints that could enforce such semantics (SORAC designs relationships specifically for constraints) or behaviour that may offer other semantics (DSM proposes the propagation of operations that forms of aggregations may require, ADAM propose attribute inheritance). It would therefore be possible to describe composite objects using several of these approaches. However, complete semantic description is not possible with most of them, therefore only very limited semantics would be available. Note that only OMS captures aggregations as references supports the concept of encapsulation. Others, because they represent all relationships as independent objects and do not provide specific mechanisms to emulate encapsulation, fail to provide any means to support this important facility.
6. **Population-based classifications**: population-based classifications, such as taxonomy, are classifications in which classes are objects that have their own importance and are involved in separate processes. As the proposed models are able to define any kind of graph, it is possible to capture graphs in which the classifying elements are not designed specifically to be part of a classification. The nodes in the classifications can therefore be any kind of object that can be manipulated on its own, irrespective of a classification.

7. **Rôles**: rôles allow objects such as specimens in taxonomy to change their behaviour or be treated in different ways by the system depending on the contents of the database. ADAM is the only documented approach that offers rôles. Rôles in ADAM are derived from the presence of relationships in which objects are involved. These relationships may carry attributes when these attributes could not belong to any one object involved in the relationship. These relationship attributes can be inherited by the objects targeted by the relationship and become new attributes of the object. For example, two objects representing people may become involved in a wedding relationship. The date and location of the wedding clearly are properties of the relationship, not of any one of the objects involved. Therefore these attributes would be defined on the wedding relationship. However, these attributes may of interest to the objects involved in the relationship, therefore may inherit them. This rôle mechanism is of particular interest for taxonomy classifications, as this situation happens to specimens that become taxonomic types: the fact that a specimen is involved in a taxonomic type relationship makes it a type specimen, and the behaviour of the system and other objects toward it change.
8. **Rules/constraints**: constraints and rules are useful to enforce working practices (e.g. the ICBN). Many proposed models support constraints. DSM only supports cardinality constraints. SORAC is a model specifically built to support constraints as relationships in order to avoid many well-known problems of constraints in object-oriented systems (e.g. duplication of constraints in all participants of a relationship, violation of encapsulation). Constraints in SORAC are expressed using the programming language, therefore can capture any constraint more or less straightforwardly (the programming language is computationally complete). OMS and Albano offer the most extensive specific constraint mechanisms among extended object-oriented models. They support for example constraints that enforce type relationships (e.g. totality of inheritance), and cardinality. Albano goes further and describes constraints such as dependency and constancy that are of clear use for semantic relationships. However, these approaches do not support deductive rules. This may prove a problem for the automatic derivation of taxon names for example.
9. **Recursive behaviour**: recursive behaviour is required to manipulate recursive data strctures such as classifications. Many of the reviewed models do not support any query language. Among those which do, two approaches to query languages exist in extended object-oriented systems: graph-based languages (GraphDB) and set-based/relational languages (OMS, Albano). The majority of the models presented here support some kind of recursion. This recursion can be simple through transitive closure or regular expressions (OMS provides the transitive closure on relations), or more complex through the definition of graph traversals (GraphDB provides a graph oriented language that allows graph rewriting and searches, e.g. shortest path between two nodes). Albano's model does not support any form of recursion.
10. **Integration with existing system**: integration with an existing system allows existing applications to continue working and provides the basis for a widespread use of the package that contains the new features. As the development of ADAM has shown, it is possible to extend an existing object-oriented model with relationships. This extension is useful when an existing taxonomic database is

to be extended with more powerful classification mechanisms. GraphDB has also shown that query languages can be extended in order to support fully graph oriented query languages.

11. **Generic classifications**: generic classifications are necessary to support certain kinds of classifications (e.g. non is-a and non is-of) and support unforeseen types of classifications. By defining different kinds of relationships and using them to capture classifications, it is possible to define classifications that are not is-a or is-of classifications. This can be done by creating new kinds of relationships, with their semantics, and linking objects together. However, for the models that do not support semantic relationships, is-of classifications are impossible.
12. **Orthogonality of classification and classified information**: orthogonality allows independence of information and classifications, which supports more naturally the multiple classification of information (e.g. taxonomic information). As any number of different kinds of relationships can be defined in some extended object-oriented systems (e.g. Albano), it is possible to clearly distinguish between the classifications and other entities in the system. Combined with semantic relationships, it is possible to extract composite objects from the classifications unambiguously.

This section has shown that by combining aspects of object-oriented models and graph models, extended object-oriented models attain a higher level of features of interest for the creation of multiple classification mechanisms. They all allow the description of explicit graphs, and some support the definition of more complex graph such as weighted graphs (e.g. GraphDB). Combined with the ability to extend relationships, they can represent orthogonal classifications. In addition, few models propose a simple level of relationship semantics that may allow the distinction of classification and classified information. In addition, some provide recursivity through graph exploration (GraphDB) or transitive closure (OMS). However, the models presented all have limitations that should be overcome in order to build an extensive classification mechanism: semantics of relationships should be extended in order to support more nuances that are necessary in modelling and programming; support for multiple concurrent classifications should be provided and integrated with the query language in order to provide generic ways to represent and query such classifications.

3.2.5 Conclusion

This chapter has presented in overview four of the most common database models: relational, object-oriented, graph-based (and semi-structured), and extended object-oriented. Each of them was compared with the list of classification requirements presented in section 2.4. This has lead to the following conclusions:

- The relational model is too simple to support complex application domains. In particular, classifications, that require both complex objects and recursive behaviour, prove very difficult to capture. In addition, graphs are not inherently supported by the relational model and the implementation of such a mechanism would require the implementation of most of the knowledge in the user application, which makes reuse and genericity very difficult.

- The object-oriented model has the advantage of being able to capture complex information in a more efficient manner. In addition, the view mechanisms are well developed and may offer a solution to the multiple classifications problem: using views, it is possible to describe concurrent subsets of the instance graph, each of which can represent a single classification. In addition, rôles have been investigated and viable solutions have been proposed, even though they require a specific underlying programming environment. However, most object-oriented query languages do not support recursivity and none of them supports the manipulation of classifications as a whole. This may limit the ability of a user application to support adequate functionality. In addition, the query language proposed does not deal with multiple simultaneous views. This limits the possible comparisons in the system.
- Graph-based models support inherently the graph/tree aspect of classifications. They also offer query languages that allow the specification of recursion (for semi-structured data). View mechanisms have been investigated and solutions exist, even though they are less complex and powerful than object-oriented views. Constraints and rules are supported by a few models, which allows the implementation of classifications where validity can be checked and information derived. Graph models seem good candidates for classification systems. However, rôles have not been investigated in these models and this may be a problem. In addition, the semantic level is insufficient to fully support complex objects and classifications, as only very few models support semantic relationships and none supports explicitly aggregation relationships with all their semantics.
- Extended object-oriented models borrow features from both graph-based and object-oriented models. Through the definition of relationships between objects, they can define graphs of arbitrary complexity. Their query languages also offer a degree of recursive behaviour that is supported by transitive closure or graph operators (one model, GraphDB, supports graphs as entities). Rôles have also been investigated by one model that supports the definition of rôles and attribute inheritance. In addition, these models support some constraints that could be useful for the definition of semantic relationships. On the other hand, these models are limited in their support for the semantics of relationships. They also lack the ability to define views or any mechanism that allows the definition of concurrent classifications.

As was shown, two approaches seem to be promising: graph-based models and extended-oriented models. These two kinds of models offer different advantages that could be useful for the representation and manipulation of classifications. However, it was also shown that none of these approaches offers full support for all the requirements of classifications in general and taxonomic classifications in particular. They would both need extension in order to become appropriate classification systems. These two approaches are tested in chapter 4.

3.3 Conclusion

Classifications have been studied in the context of biological data for some time. Some conceptual classification mechanisms have been described, but they always expose a biased view of biological classifications where definitive immutable classifications exist. It was shown in chapter 2 that this view is unreasonable as biological classifications, in particular taxonomic classifications, are dynamic knowledge representations that change with opinions and time. Therefore the classifications mechanisms presented do not support classification work properly.

Many database models and query languages have been proposed in the literature. Many of these models had specific use (e.g. financial operations for the relational model), but have been applied to new disciplines as these became computerised. Some of these models (relational, object-oriented, graph-based) have been applied to managing biological classifications. However, this chapter has shown that their support for the required features of classifications is insufficient. They all either lack modelling power and/or querying power.

More recent and powerful models have also been proposed (e.g. semi-structured and extended object-oriented models). These models also have also been designed for specific use (e.g. support for dynamic and unstructured data for semi-structured models). However, as some are more powerful in their ability to capture complex data (e.g. extended object-oriented models), they offer higher levels of support for biological classification mechanisms (e.g. explicit and semantic relationships).

Section 3.2 has presented two models that offer interesting features for the support of biological classifications: graph-based models, that inherently support the description of graphs and in some cases recursivity, and extended object-oriented models, whose ability to capture complex data may be of use for the description of multiple concurrent classifications. The second type of model, that subsumes the first, has been used as the basis for the building of a suitable model for plant taxonomy and is presented in chapters 4, 5 and 6.

4 Prometheus/ODMG

As was shown in section 2, taxonomic data is inherently hierarchical. It is therefore a straightforward and sensible choice to build a system that would support taxonomic work on a model that inherently supports such structures. A prototype was built for this thesis in order to test the feasibility of supporting multiple classifications mechanisms on a graph-based model [Raguenaud '00]. That prototype showed that a graph-based approach, although intuitive, didn't provide all the mechanisms required to support taxonomic data and working practices. This chapter presents the model developed for the second prototype built for this thesis, an object-oriented database system. The approach chosen is that of introduction of first-class relationships in ODMG.

As was presented in section 2.4, a classification database needs to satisfy specific requirements. The new model therefore needs to offer the ability to describe trees/graphs (including weighted graphs), direction of relationships (i.e. graph arcs), composite objects, recursion, rules, multiple classifications, traceability, and rôles.

In designing the model, the following additional criteria, which overlap with requirements in section 2.4, were considered especially important:

- The integration should be as generic as possible. Building a model on a particular object-oriented implementation model would restrict its ability to be used. Therefore the integration is made in a simple object-oriented environment common to the majority of object based database systems. ODMG has been chosen for that role because it is a (de facto) standard.
- The integration should be implemented on a commercially available database in order to make its application directly useful to others.
- The integration should be able to reuse existing data so as to extend an existing system with new features without loss of data or heavy treatment of existing datasets. One important aspect of the system built is its ability to be integrated with existing data without requiring many changes before becoming useful.
- The integration should not require the participation of existing objects and so avoids rewriting of existing data. Since existing data is to be reused, these data are not expected to be able to provide any level of service for this system.

There is debate in the database community about what is to be embedded in the database system, and what should be stored in user applications. Some argue that the database system should be as simple as possible and most of the knowledge stored in the user application. This way the system is easier to design and more generic. Others argue that as much knowledge should be pushed downward in order to offer more abstract and complex operations, and free the user application of most of the generic work, however complex. This work is based on the decision that, in order to offer a system that offers features that make the development of applications easier, more information should be pushed into the database

system. This way, it is possible to design and optimise the operations once and for all. It is also possible to enforce consistency in a more generic and natural way.

Existing systems such as GraphDB [Güting '94], SORAC [Doherty '93], and Albano [Albano '91] offer partial integration of an object-oriented environment with graph mechanisms and structures (e.g. GraphDB which does not support object-oriented semantics of relationships and offers a graph based query language, and SORAC that lack the definition of a query language) or use models tailored for the integration (e.g. Albano's collection based model [Albano '91]).

Prometheus provides an expressive environment by extending an object-oriented model with relationships whose semantics are clearly defined and by proposing a suitable query language. The graph mechanisms that are added improve the ability to model object-oriented systems by offering a more direct mapping from object modelling to object implementation. The resulting environment provides the user with multiple viewpoints in which the graph extensions can be ignored to work in a standard object-oriented environment, object-oriented features can be left aside to work in a graph environment, or both environments can be used at the same time in a smoothly more expressive integrated system.

Although the intention is to create a database with more expressive relationships, it is not to create a UML database. Such a database would be based on (mostly binary) relationships constituted of tree parts: the relationship itself, and both relationship ends. These relationship ends would contain all the information necessary for the interpretation of the relationship. For example, a relationship end can be of type association or aggregation; it supports multiplicity; it defines the behaviour of the relationship with attributes such as navigability or visibility. However, using such a definition, as has been done in non-database systems such as cOlOr [Barbier '01] would complicate the definition of relationships and above all make their querying far more complex than necessary. For example, since relationships in the cOlOr sense could be queried as normal objects because they may contain information or can participate in a query, the ends of the relationship would have to be expressed in each occurrence of the relationship. Relationships would therefore become complex path expressions themselves and augment the difficulty to express queries. In addition, the ability to use relationship ends instead of predefined relationships allows the definition of many configurations that are not meaningful. For example, with relationship ends, it is possible to express that a relationship has an aggregation end as both relationship ends, thereby implying that the whole-part relationship so defined is a whole-part relationship in both directions. Clearly, if an object is part of another, the other cannot be part of the first one. It is easy to restrict the definitions to what is meaningful in order to simplify the model. The attributes that can be defined on relationship ends are also unnecessarily duplicated in each end. For example, the *changeability* attribute is defined in both ends of a relationship, although non-changeability of one end implies non-changeability of the whole relationship, therefore of both ends. For these reasons, relationships have been defined as single entities with no replaceable parts or no optional parts (relationship ends), with a single set of attributes valid for the whole relationship.

This chapter contains the following sections: first, the notation used for the object-oriented model in the thesis is presented. Then the model itself and the relationships are defined with reference to ODMG, and instance synonyms, a peculiarity of the model motivated by taxonomic working practices, are introduced. A database example illustrating how the concepts presented here are used to build an application is given. Finally, a few remarks on the model and its workings are noted and the chapter concludes.

4.1 Notation

In this thesis, the following notation has been adopted for object-oriented class diagrams in the Prometheus sense:

A class is represented by a box. The upper part of the box contains the class name, and the lower part a list of atomic attributes (Figure 9). Although these attributes should be independent objects targeted by relationships in Prometheus, they are shown as part of a class to simplify diagrams, except when the aggregation relationship that references them has specific semantics. This is consistent withe definition of atomic attributes in Prometheus for compatibility reasons (see section 4.8). Note that types are not represented as they are part of the class definition and are nameless. This does not mean that types cannot be shared among classes.

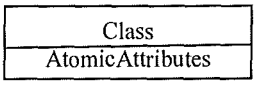


Figure 9: Diagram of a class

Relationships in Prometheus are represented by diamond shape boxes that contain the name of the relationship class and occasionally a list of atomic attributes (Figure 10). The incoming arc may start with a diamond touching the source class of the relationship if the relationship represents an aggregation. Cardinality is shown on each end of the arcs composing the relationship. The incoming and outgoing arcs always start at one corner of the diamond box. Relationships can reference other classes to become n-ary relationships. The link between the relationship and these classes is represented as a dotted arrow to show that it has different semantics than other relationships. These dotted arrows never start at one corner or the diamond box.

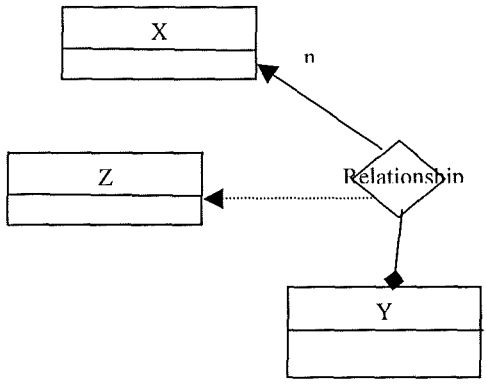


Figure 10: Diagram of a relationship class

Inheritance is represented by thick black arrows between classes or relationship classes that are involved in the inheritance relationship (Figure 11). The targeted class is the super-class. Note that when relationship classes are involved in an inheritance relationship, all classes should show the targeted classes, as through co-variance, these may be different between the super-class and its sub-classes.

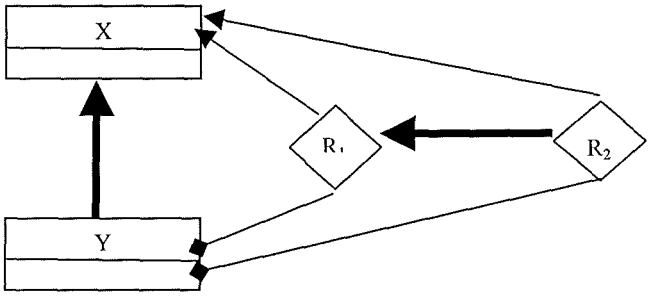


Figure 11: Diagram of inheritance

Exclusivity is shown by crossing incoming relationship arcs with a curved line (Figure 12). Exclusivity may appear between distinct relationship classes, or on a single relationship class, in which case it shows exclusivity among instances of a single class only.

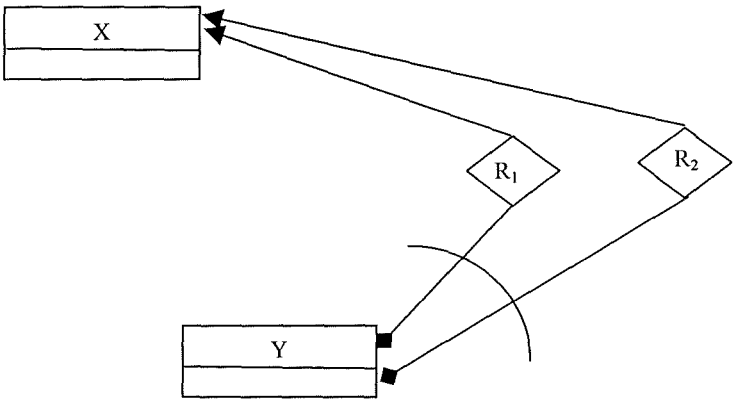


Figure 12: Diagram of exclusivity

Sharability is represented by a curved line crossing outgoing relationship arcs (Figure 13).

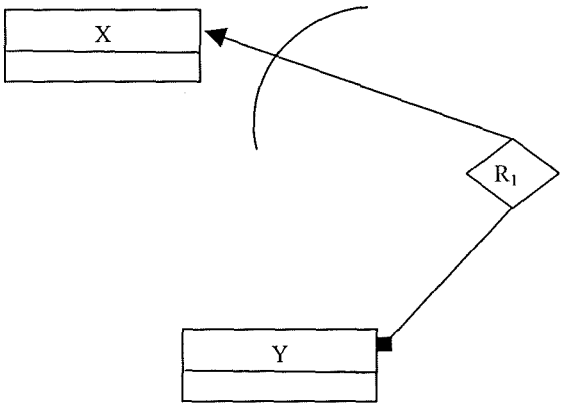


Figure 13: Diagram of sharability

4.2 The basic object-oriented model (ODMG)

In this section the model that was used to define the relationship extensions is described. More formal definitions can be found in [Alagic '99]. A built-in class is provided by ODMG: Object. That class can be used wherever a class is expected therefore can be seen as the root of all inheritance hierarchies. The interface and precise definition of this class is ignored in this document, as it is not necessary to understand the work described.

In ODMG, classes are defined as collections of methods and attributes. A method is defined by the type of objects it returns and the pairs (type, name) of its arguments:

> Given a database schema S, a method called m is defined by: C m (C_1 r_1, ..., C_n r_n), C is a class in S, i.e. C $\in$ S, C is the type returned by the method, m is the method name, r_n, $1 \leq i \leq n_i$, are the name of method arguments and C_n are argument types.

For example: int add (int number, int increment) is a method that adds the argument "increment" of type int to the argument "number" of type int and returns a value of type int.

In a similar manner, attributes are described as pairs (type, name):

> Given a database schema S, an attribute called a is defined by: (C, a), a is any valid attribute identifier, C $\in$ S (as ODMG does not distinguishes types and classes, classes are sub-types of types according to the ODMG meta-model).

For example: (String, name) represents an attribute called "name" of type String.

In addition, according to the ODMG meta-model, classes also contain references to their super-classes or super-interfaces, therefore they should be made explicit in class definitions. A class can therefore be defined as a name associated with three collections: an attribute collection, a method collection, a superclass/superinterface collection. A class can be temporarily defined as follows:

Given a database schema S, $C := ((C_a, a)_i, (C_m\ m\ (C_1\ r_1, \ldots, C_n\ r_n))_j, P_z),) \wedge S := S \cup \{C\}$

$0 \leq i \leq n_i \{(C_a, a)_i \in C \wedge (C_a)_i \in S\} \wedge 0 \leq j \leq n_j \{C_m\ m\ (C_1\ r_1, \ldots, R_n\ R_n)_j \in C \wedge (C_m)_j \in S \wedge (C_n)_j \in S\} \wedge 0 \leq z \leq n_z \{P_z \in S \wedge P_z \leq \text{Object} \wedge C \leq P_z \wedge \text{Object} \in P_z\}$

$C \leq P_z, \rightarrow$

$\forall (C_a, a) \in Pz \rightarrow (C_a, a) \in C$

$\wedge \forall (C_m\ m\ (C_1\ r_1, \ldots, R_n\ R_n) \in Pz \rightarrow (C_m\ m\ (C_1\ r_1, \ldots, R_n\ R_n) \in C.$

For example: Person((int, age), (String, name), int age(), {Object}) is a class representing people that contains two attributes, age and name, a method without argument returning the age of the object, and is a sub-class of Object, the super-class of all classes in ODMG (ODMG has a unique inheritance root).

The concept of constraints needs to be introduced, as these will be necessary for the expression of the semantics of relationships (section 4.3). A constraint is composed of an activation event, a condition of applicability, and a constraint (possibly an action). For simplicity, events are selected from a list of possible events (that can be primitive or composite, depending on the underlying system), and conditions and actions are seen as simple text strings (in reality they could be expressions from the constraint language or queries). Because Prometheus should be kept in the spirit of ODMG, and because ODMG does not describe constraints precisely, constraints are described as objects that are subclasses of a generic Constraint class, itself a subclass of Object. The structure of constraints is defined as follows (the String class is a built-in type in ODMG and Prometheus): a constraint is a class that contains at least three attributes: an event attribute that describes the event (simple or composite) to which the constraint will react, a condition of applicability attribute that will contain the condition under which the constraint makes sense, a condition attribute that contains the actual constraint to enforce, and possibly an action to take when the constraint is violated. The structure of constraints is therefore as follows (the String class is a built-in type in ODMG and Prometheus): The existence of a set of event descriptions E = { ... }, atomic or composite, that can be system dependent is assumed.

Given a database schema S, constraint := $((E, m_1)_i, (Cnd_1, m_2), (Cnd_2, m_3), (Act, m_4), P_z) \wedge S := S \cup \{\text{constraint}\} \wedge C := C \cup \{\text{constraint}\}$

$\text{Constraint} \leq \text{Object} \wedge Cnd_1, Cnd_2 \leq \text{String} \wedge \text{Act} \leq \text{String} \wedge 0 \leq z \leq n_z \{P_z \in S \wedge \text{constraint} \leq P_z \wedge \text{Object} \in P_z\}$

m_{1i}, m_2, m_3, and m_4 represent the names of the attributes that represent the activation events E_i $1 \leq i \leq n_i$, the condition of applicability (Cnd_1), the constraint (Cnd_2), and the action (Act) of the rule respectively.

For example: ((OnTransactionAbort, m_1), (), (String, m_2), (String, m_4), {Object}) represents a constraint fired when a transaction is aborted, and that does not have a condition of applicability.

Thereafter, creating a new constraint type is equivalent to sub-classing the Constraint class. The Constraint class also provides the interface necessary for its manipulation by the system and user applications. Constraints are attached to the classes on which they are defined because it simplifies understanding of the schema and keeps the application in an object-oriented and encapsulated form. For example, if a class Part has a constraint that its Price must be between 10 and 10000, it is sensible

to attach the constraint to that class instead of storing it elsewhere in the system. However, constraints that involve many objects are captured on the relationships that link these objects. Note also that the fact that constraints are generally defined on classes does not imply that they cannot be stored elsewhere in the system at runtime for optimisation purposes and cannot be created as independent objects.

In ODMG, classes may have extents. Extents are collections of instances, therefore they must be of type Collection or one of its subclasses (sets for example). ODMG does not specify how extents should be managed (by classes or as independent entities). The implementation of extents is left to the underlying database manager. Extents are therefore ignored from this definition of classes, but can still be managed by classes. An ODMG class in Prometheus is therefore defined as a set of attributes, a set of methods, and a set of constraints:

> Given a database schema S, Class := $((C_a, a)_i, (C_m\ m\ (C_1\ r_1, \ldots, C_n\ r_n))_k, X_l, P_z) \wedge S :=$ $S \cup \{Class\} \wedge C := C \cup \{Class\}$
>
> $Class \leq Object \wedge 1 \leq j \leq n_a \{C_a \in S\} \wedge 1 \leq l \leq n_X \{X_j \leq Constraint\} \wedge 1 \leq k \leq n_m \{(C_m\ m$ $(C_1\ r_1, \ldots, C_n\ r_n))_k \in Class \wedge (C_m)_k \in S \wedge (C_n)_k \in S\} \wedge 0 \leq z \leq n_z \{P_z \in S \wedge Class \leq P_z \wedge$ $Object \in P_z\}$

For example: Person((int, age), (String, name), int age(), {Object}) is a class representing people that contains two attributes, age and name, and a method without argument returning the age of the person.

Interfaces can also be defined as restricted classes with no state, or classes can be declared as a special type of interface with attributes (as in ODMG), or interfaces can be defined in a similar way to classes. In this thesis, no particular attention is given to interfaces, as they do not allow the complete definition of relationships (relationships require state). Therefore only classes are studied from this point forward.

4.3 Relationships

Relationship objects can now be defined by extending the model presented in the previous section.

As with constraints, changes to the ODMG model must be avoided, therefore relationship classes must be subclasses of the Object class. The skeleton of relationship classes is as follows: it contains as for any class a set of attributes, but some of these attributes are built-in. These attributes are necessary to locate the source and destination classes of the relationship class, its built-in attributes, and its constraints. A relationship class is therefore defined as: an origin attribute, a destination attribute, a set of user methods, and a set of user attributes. Thereafter, all relationship classes are subclasses of this Relationship class. The system can then identify them and treat them as necessary, without modification to ODMG. A relationship class is therefore defined as follows:

> Given a database schema S, Relationship := $((C_o, origin), (C_d, destination), (C_m\ m\ (C_1$ $r_1, \ldots, C_n\ r_n))_k, (C_{ab}, A)_b, (C_{aj}, A)_j, X_i, P_z), S := S \cup \{Relationship\}, C := C \cup$ $\{Relationship\}$

$0 \leq z \leq n_z \{P_z \in S \wedge \text{Relationship} \leq P_z \wedge \text{Class} \in P_z\} \wedge 1 \leq i \leq n_X \{X_i \leq \text{Constraint}\} \wedge 1 \leq k \leq n_m \{(C_m\ m\ (C_1\ r_1, \ldots, C_n\ r_n))_k \in \text{Relationship} \wedge (C_m)_k \in S \wedge (C_n)_k \in S\} \wedge 1 \leq b \leq n_b \{(C_{ab}, A)_b \in \text{Relationship} \wedge (C_{ab})_b \leq \text{Boolean}\} \wedge 1 \leq j \leq n_j \{(C_{aj}, A)_i \in \text{Relationship} \wedge (C_{aj})_j \leq \text{Boolean}\}$.

$(C_{ab}, A)_b$ represents a set of n_b built-in Boolean attributes whose signification follows in subsequent sections, and $(C_{aj}, A)_j$ represents a set of n_j user attributes with type $(C_{aj})_j$ respectively. C_o and C_d are origin and destination attributes respectively

For example: worksFor((Person, origin), (Company, destination), (), (Boolean, LifetimeDependent), (Float, salary), (), {Object}) represents a relationship class relating Person objects to Company objects, without methods, with a built-in attribute "LifetimeDependent", an attribute salary of type Float, and no constraint.

There are two kinds of relationships in Prometheus: aggregations and associations. Each kind of relationship is subject to limitations on the classes that can be used as its source or destination. Restrictions need to be applied on the classes that can be the source or the target of a relationship class. Indeed, since atomic node classes represent simple concepts, they cannot be the source of a relationship class (an atomic class does not have outgoing edges). Moreover, they cannot be the destination of an association, since an atomic class cannot make sense if it is not part of a larger concept (e.g. there cannot be associations to numbers, whereas numbers can be part of the representation of a Person concept, for example). This definition of aggregation is more general than some data modellers define it. The purpose of this general definition is to provide a system that allows the elimination of references as relationships. In Prometheus, all relationships should have semantics that make their meaning and behaviour clear. For example, if a Person object is created and a height of 1.80m is assigned to it, it should be clear whether this value can be changed during the life of the Person object. In OODBs that only support references to represent relationships, the semantics of the relationship must be implemented through methods, which makes them harder to check and understand. In Prometheus, the height attribute could be referenced through an aggregation relationship that could implement the changeable/non-changeable behaviour. Note however that this is not compulsory, and if no semantics are necessary, object references defined in types are possible.

Relationship classes are subdivided into 2 categories: single and multi-valued relationships. These relationships allow the definition of 1:1 and M:N relationships (1:N are a subtype of M:N) between classes. ODMG describes one-to-one (1:1), one-to-many (1:N), and many-to-many (M:N) relationships, but one-to-many relationships are a special case of many-to-many relationship. Furthermore, both these categories are divided into two further categories that support the semantics of association relationships and aggregation relationships. They are defined as subclasses of the relationship class defined earlier. In this way, it is easy to treat relationships as first class objects in the database and therefore benefit from object-oriented features (e.g. inheritance). Sub-classes of the basic relationship class are therefore created to handle some nuances:

- SARC: Single Association Relationship Class (whose instances constitute the set SARI, Single Association Relationship Instances).
- SGRC: Single aGgregation Relationship Class (whose instances constitute the set SGRI, Single aGgregation Relationship Instances).
- MARC: Multiple Association Relationship Class (whose instances constitute the set MARI, MultipleAssociation Relationship Instances).
- MGRC: Multiple aGgregation Relationship Class (whose instances constitute the set MGRI, Multiple aGgregation Relationship Instances).

ODMG, as it does not describe any semantics for relationships, does not make these distinctions in semantics. But it is necessary to introduce them in order to provide a means to extend the definition of relationship to a sensible and usable level. The way these subclasses are implemented depends on the underlying database system. The description of attributes and constraints can be acquired in subclasses of the basic relationship classes. For example, a subclass of Aggregation, Composition, can be created and it can acquire the lifetime dependency and non-sharing behaviours.

Figure 14 shows the basic taxonomy of classes in Prometheus.

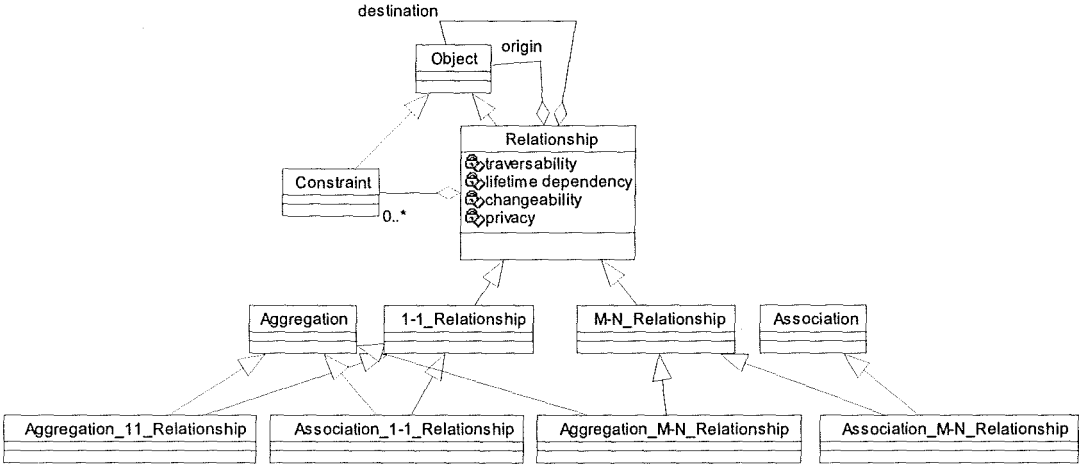


Figure 14: Meta model

Although only binary relationships are defined (as in other models such as GraphDB, OMS), they can be treated as n-ary relationships (as in Albano [Albano '91]) as they can carry attributes and these attributes can reference other objects. In this way, relationships link n objects. In fact, because the semantics of ODMG should not be changed, with regard to structure, there is no difference between relationship ends and other attributes. It is therefore easy to create n-ary relationships. This is one of the strengths of this approach. However, relationships always have a preferred source and destination (an orientation), and traversal direction that might be interpreted by the query engine and the database programming interface. This choice is motivated by the fact that many relationships are inherently orientated (e.g. an aggregation clearly goes from the whole to the part). Orientation in relationships simplifies their understanding, as they are usually thought of in one sense (e.g. the classical *works_for*

relationship in the Company-Employee example), and also supports the propagation of operations (e.g. deletion of objects that imply the deletion of dependent objects). See section 4.4 for more details.

Co-variance is important in this model as it allows the refinement of classes in subclasses. Co-variance means that methods/attributes can be redefined and their signature changed to more specific types. This is particularly useful when sub-classing relationships represents refinement of the relationship. Contra-variance is the opposite of co-variance; it means while the type of classes is specialised, the type of embedded attributes or method arguments is generalised [Meyer '97]. Many object-oriented models do not support this behaviour and rely on code to control the accepted types for a method or an attribute. However, this approach is confusing, as it is necessary to find the corresponding code and read it to understand the model. It is also problematic because it breaks the definition of the semantics of the model into two parts, the actual model and the code written in classes. This makes the model harder to define and even harder to read, understand, and maintain. However, it is necessary to be able to integrate these extensions with ODMG. Inheritance defined in ODMG only supports a no-variant redefinition of methods/attributes, i.e. when methods/attributes are redefined, their signature must remain exactly the same. Some models support an alternative approaches to redefinition: co-variance, contra-variance. Since Prometheus uses co-variance and since the ODMG model should not be changed in principle, this feature must therefore be emulated instead of inherently supported by the model, e.g. using additional constraints or code.

As can be seen, the definition of the structure of relationships only provides their basic structure. For example, there is no means to describe that a relationship has a cardinality of 1:2 or that it can be traversed in only one direction. These properties are managed by special attributes defined on the relationship or by constraints. The next two sections present these properties and constraints.

4.4 Semantics

In the literature, two main kinds of relationships are defined: aggregations and associations. Depending on the semantics given to these relationships and the approach chosen by a particular user, their names may vary (e.g. composition in UML [OMG '99] is an aggregation for which a lifetime dependency relationship is defined between the whole and its parts), but they can still be classified into these two categories (e.g. a composition is a form of aggregation). In this section, the different semantics given to these relationships are reviewed and then the properties inherent to each of them extracted. Secondly, their implementation in Prometheus is described as four basic features: attributes, constraints, attribute inheritance, and collections.

4.4.1 Aggregations

Aggregation relationships have been given very different semantics over time. These semantics vary from simple fuzzy aggregations (e.g. [Rumbaugh '91]) to complex semantics ([Odell '94b]). Within this range, many contradictory semantics have been given (e.g. [Odell '94b] vs. [Henderson-Sellers '97]).

Odell [Odell '94b] classifies aggregation relationships into six distinct categories depending on their context and their usage: component-integral object composition (parts bear a functional or structural relationships with the whole), material-object composition (similar to component-object composition but with an invariant behaviour), portion-object composition (the parts are of the same nature as the whole), place-area composition (similar to portion-object composition with invariability), member-bunch composition (no functional or structural relationship exists between the parts and the whole), member-partnership composition (similar to member-bunch composition with invariability). From this classification it can be seen that aggregation relationships have the following properties: configurational (functional or structural relationship between the whole and the parts), homeomeric (parts have the same nature as the whole), and invariant. Table 1 shows how the different relationships identified can be described with these properties:

	Configurational	Homeomeric	Invariant
Component-integral object	Yes	No	No
Material-object	Yes	No	Yes
Portion-object	Yes	Yes	No
Place-area	Yes	Yes	Yes
Member-bunch	No	No	No
Member-partnership	No	No	Yes

Table 1: Odell's relationships

This classification is debatable, as shown in [Henderson-Sellers '97]. Henderson-Sellers shows that the description of some of the categories is subject to interpretation. He therefore adds a feature-activity category, and changes the place-area to configurational/homeomeric/invariant and the portion-object category to non-configurational/homeomeric (invariance is undecided). This leads to the classification presented in Table 2.

	Configuration	Homeomeric	Invariance
Component-integral object	Yes	No	No
Material-object	Yes	No	Yes
Place-area	Yes	No	Yes
Feature-activity	Yes	No	Yes
Portion-object	No	Yes	No (or yes)
Member-bunch	No	No	No
Member-partnership	No	No	Yes

Table 2: Henderson-Sellers' relationships

However, it can be seen that some properties highlighted by Odell and Henderson-Sellers are only manageable on the user's side because they are purely abstract nuances. A computer cannot guess the functional or structural relationships between objects, as this information is not represented in the system. Likewise, the homeomeric aspect of some of the relationships is hard to manage, as there is not necessarily a relationship between objects of the same nature in the real world (e.g. Odell cites the example of a slice of bread as a part of a loaf of bread, two objects that could be distinct and independent in a computer representation). These properties can therefore be set aside because they are unreliable.

Some data modellers [Henderson-Sellers '97] insist that composite objects (the objects that play the rôle of whole) must exhibit at least one emergent property, i.e. a property that cannot be derived from the values of the aggregate objects. In addition, composite objects must exhibit at least one dependent property, i.e. a property that can be derived from the values of the aggregate objects. This object can be related to the *dominant class* in OMT [Rumbaugh '94], although the dominant object has no clear restriction on its properties. This property of composite objects is too great a restriction to be applicable in object models. Indeed, because computer models are simplifications of real world models, all properties of the actors of a composition are not necessarily represented and therefore emergent properties may be ignored in the model. For example, a boat is a composition of parts and clearly the boat has dependent properties (it floats as a consequence of having a hull), and it has emergent properties (e.g. it can carry passengers). However, these properties might exist in the mind of the modeller, but may not be represented in the system.

Bock and Odell also argue [Bock '98], extending Henderson-Sellers' ideas [Henderson-Sellers '97], that it is important that relationships are represented by first-class objects, as relationships can sometimes themselves be part of the composite object. Indeed, when two parts of a composite object are involved in a relationship that is vital to the working of the whole object, the relationship itself can become itself a part of the composition (a car is more than a collection of parts, and some parts must be related in very specific ways, e.g. the *powers* relationship that exists between an engine and the wheels of a car). Composite objects are also said to offer several services [Bock '94]: instantiation of parts (a composite object might only make sense with its parts), propagation of operations (when a composite object is modified or deleted, its parts might be affected), maintenance of connections between parts (when a part is removed from the whole), and enforcement of invariant compositions. These services can be seen as providing lifetime dependency, propagation of operations, and invariance.

Similarly, Pirotte [Pirotte '97] presents an implementation of a semantic model that focuses on the representation of relationships. It proposes, as previously introduced models, exclusiveness/sharing (both at a local and global level, i.e. within a composite or among composites), and dependence/independence. An interesting addition to common semantics is the ability to represent partially the semantics defined by Henderson-Sellers [Henderson-Sellers '97] and others. Their model offers three types of aggregation relationships: part relationship, part association, and part recursion.

Part relationship represents the aggregation other authors have described. Part association represents the association to objects which all share the same class. Part recursion, a special case of part association, represents the homeomeric relationship. These specialised relationships can be combined with transitivity in order to provide attribute derivation (as in ADAM [Díaz '90] to a certain extent).

Kolp [Kolp '97] proposes a model where different kinds of relationships (generic, aggregation, aggregated, derived, materialization) are first-class objects. These relationships support a set of properties (cardinality, dependency, and exclusiveness) that constitute their semantics. Although this model supports relationships and their semantics, the set of properties a relationship can exhibit is fixed, as it is part of the type of relationships. It is therefore impossible to extend them beyond the predefined properties (e.g. how can *encapsulation* be described?).

A different approach is taken by Blaha [Blaha '93]. Indeed, the author insists on the importance of different properties of the aggregation relationship: it should be transitive (in order to support the transitive closure), antisymetric (i.e. directed), and cardinality plays a central rôle. According to the author, two kinds of aggregation relationships exist: physical aggregation and catalogue aggregation. The main difference between these two aggregations is that catalogue aggregations are a higher-level description of a system, whereas the physical aggregation is the physical more precise description of the reality of the system. He implies that physical models are trees, whereas catalogue models are directed acyclic graphs (nodes may have multiple parents). As a tree restrict the number of parents of a node to one, it is necessary to be able to support sharability/exclusivity.

Other models propose a simpler view of the semantics of relationships: OMT has a less rigorous definition of the semantics of relationships. Only two kinds of aggregations exist: aggregations and compositions. Compositions are distinguished from aggregation by the fact that they imply non-sharing of the part objects. However, It can be argued that a room is composed of walls and doors, and that these can be shared between adjacent rooms. The distinction is not so simple. In UML, aggregation and composition (as a "strong aggregation") are defined. The distinction is made on the basis of invariability and lifetime dependency: a composition is invariant and implies destruction of parts when the whole is destroyed. UML, as OMT, seems to be more implementation oriented than other descriptions of the semantics of relationships, probably because of its programming language background. UML only insists on the very few properties that can be easily checked by a computer (invariance) and does not discuss abstract semantics (e.g. homeomeric).

Aggregation relationships should therefore be first-class concepts, should provide means to support lifetime dependency, invariance (or changeability), sharability, and propagation of operations, and should support the definition of attributes. It was also said that cardinality is a central problem since the usage of an aggregation relationship for different purposes requires different strict cardinalities. These properties and behaviours can be used in conjunction or disjunction in order to create richer semantics and provide relationships that implement various behaviours. For example, lifetime dependency and

invariance can be selected to form a composite object that can only make sense with its parts (or parts that can only make sense with a whole), and which cannot change during its life.

4.4.2 Associations

In comparison to aggregations, associations are given little semantics and have been less studied. This is due to the fact that associations are more general relationships, thereby bearing less precise meaning. Note however that some modelling methods offer the same semantics for aggregations and associations (e.g. UML). Although this can be sensible for some properties (e.g. invariant) it seems harder to justify for others (e.g. lifetime dependency, sharing).

One point that can be noted is that that all methods agree on the fact that aggregations are special cases of associations. But in parallel, associations are said to be inherently bi-directional, and aggregations uni-directional [Motschnig '99] (anti-symmetric and oriented). In addition, in OMT second generation [Rumbaugh '95] relationships may have an arrow next to their name to indicate how they should be read. This shows that although they can be traversed in two directions (as associations are defined as bi-directional), they must be read in one direction, therefore are directed. This shows that what is usually referred to as direction is in fact made of two distinct concepts: directionality of the relationship itself, and directionality of the operations (or attributes) and of the relationship traversal. These concepts can be called orientation (where the relationships goes from and to, how it should be read, and how the operations can be propagated) and navigation (how the relationship can be traversed).

4.4.3 Built-in attributes

The previous sections have reviewed existing analysis of semantic relationships. These analyses have highlighted many aspects of relationship semantics but have proved contradictory: for example Odell and Henderson-Sellers interpret similar concepts differently. However, it can be noted that these concepts use the same basic properties such as invariance or lifetime dependency. It also appears that some of the concepts used to describe semantics are not actual properties of the relationship, but instead are properties of the objects involved in the relationship. For example, Odell's portion-object composition where the parts are of the same nature as the whole is not a property of the relationship between the part and the whole, but rather meta-information about these objects. In addition, abstract nuances such as the homeomeric aspect of some relationships are unreliable and impossible to capture [Pirotte '97]. However, as in the model presented by Pirotte [Pirotte '97], it is possible to define subclasses of relationship classes that contain the homeomeric concept for example. However, unlike [Pirotte '97], this does not provide additional information to the database system (transitivity is inherent in Prometheus, as recursing down aggregation hierarchies is a feature of the query language, see section 5.1.1.3). It is therefore more useful to concentrate on basic behavioural properties of relationships and let the user gather them in order to create meaningful semantics.

Following this analysis of relationships a set of properties is selected:

- *navigation*
- *lifetime dependency*
- *changeability*
- *privacy*
- *cardinality*
- *exclusivity*
- *sharability*
- *encapsulation.*

Properties are managed by two distinct processes: built-in attributes and constraints. These properties represent behaviours of relationships that are omitted by ODMG and similar models and most of the extended object-oriented models that support relationships (e.g. OMS [Norrie '93], GraphDB [Güting '94], SORAC [Doherty '93]), but include the semantics described in [Bertino '98]. They participate in the definition of a relationship as well as its point of origin and its destination.

Navigation. Defining relationships as objects makes them equivalent to references with inverses, i.e. they can inherently be traversed in both directions. As was explained in section 4.4.2, orientation and navigation must be distinguished. Relationships have a preferred orientation (easier for understanding and for specifying propagation of operations), but can be traversed in either direction (navigation). However, it is sometimes necessary to restrict the allowed traversal of a particular relationship. For example, in ODMG, references without an inverse can only be traversed in one direction and are therefore relationships with only one direction of traversal. An attribute has therefore been defined, *navigation*, which describes the direction a relationship can be traversed. This attribute is then taken into account when objects request to follow a particular relationship in which they are involved (equivalent to following a reference in classical object-oriented programming languages); in this case the query language will only traverse the relationship if allowed.

Lifetime dependency. The *lifetime dependency* attribute is only applicable in the context of an aggregation relationship. The implication of lifetime dependency is that when the object that is the whole is destroyed, its parts are also destroyed (the concept of the destruction of objects is system dependent and might imply garbage collecting). Lifetime dependency does not imply non-sharing of objects. Objects can be used by two composite objects and still be shared. For example, the same part might be part of two composite objects (e.g. a door between two rooms). Particular to this model, two kinds of lifetime dependencies are distinguished: a strong one and a weaker version. The weak version states that whenever the whole part of the relationship is deleted, its subparts must also be deleted. On the contrary, strong lifetime dependency states that whenever a part is deleted, the whole, and therefore all the other parts must be deleted. A car without a body part is not a car. Strong lifetime dependency is usually used along with non-sharing (see this concept in section 4.4.4). Indeed, if sharing is allowed, a

part might be used in many part-whole relationships. Strong lifetime dependency would imply that deleting the part would delete the whole part of the relationship, in this case all whole objects using the part object. This behaviour may be required, but is not necessarily implied. In order to make sure that no unexpected behaviour occurs, non-sharing is useful.

Changeability. Other properties restricting or clarifying the status of a relationship commonly used in UML and other modelling methods are necessary. For example, *changeability* that describes if a particular relationship can be modified once it has been created. The changeable attribute specifies that once a relationship has been created, it cannot be changed. For example, it could be decided that when a customer buys a part, it cannot be returned and the relationship exists forever. Note that *changeability* is independent from *substituability* [Motschnig '99]. *Substituability* means that one object targeted by a relationship can be swapped for another without changing the relationship. Therefore if *substituability* is allowed, a relationship that is marked as *non-changeable* can target many different objects in its lifetime. In Prometheus, *subsituability* is not allowed, as a relationship is defined by the objects it targets. Therefore, if a relationship must target an object different from the one it has been created with, it must be deleted and another relationship must be created in its place.

Privacy is not an attribute whose existence is motivated by object modelling. On the contrary, it is motivated by object-oriented programming languages. It is generally ignored by semantic models (e.g. [Díaz '90], [Pirotte '97], [Kolp '97], [Shah '89], etc.). All object-oriented programming languages propose a mechanism that allows the distinction between private and public parts of an object. In many programming languages (e.g. Java [Sun '95] and C++ [Stroustrup '91]), a qualifier is used when each attribute is defined in order to specify whether it can be publicly accessed or not. ODMG uses interfaces to specify the public interface of an object and classes to specify the private interface of an object. Note that because of the definition of interfaces in ODMG, it means that methods can be public, but attributes are always private. In Prometheus, as explicit relationships are used for all relationships, unlike other approaches (e.g. OMS), attributes become objects external to their owner. In order to provide public/private specification of attributes, a *privacy* attribute is defined. This attribute can take several values (e.g. public, private, protected). When an object requests to follow a relationship from a given object, the system checks the privacy attribute of the relationship, and only allows the operation to be carried out if access rights are not violated. It is true that this approach has its limitations: in a database that manages extents, it is always possible to reach an object, whether it has been used as a private member or not. But this is not a consequence of the model described here, it is a consequence of extents.

Other properties can also be defined (e.g. *visibility* that is an extension of *privacy*), and while not currently described in Prometheus, the framework exists to do so.

4.4.4 Constraints

Some properties of relationships cannot be expressed simply using an attribute. For example *sharability* is a complex concept that necessitates the consideration of all relationships defined in the database in order to make sure that none refers to a particular object. Therefore, in addition to these user-defined constraints (whose detailed mechanism is explained in section 5.2), relationships in Prometheus contain a set of constraints that are used to augment their modelling ability and maintain more integrity depending on the meaning the user has given to them, i.e. what kind of relationships they are. In this section, the following built-in constraints are described: *cardinality*, *exclusivity*, *sharability*, and *encapsulation.*

Cardinality. There is often the need for precise cardinalities. These cardinalities are managed by constraints on relationship classes. These constraints check the number of relationships in the database obeying certain criteria. For example, for a 1:N relationship, there should be no other relationship of the same class as the one under study that has the same destination. More precise constraints can be expressed in a similar way (e.g. 1:6), except that instead of checking the non-existence of relationships, the number of relationship instances would be counted. Cardinality is also useful to offer support for optional/mandatory properties [Iivari '92]. Indeed, a cardinality of 0,1 specifies that a property is optional, whereas a cardinality of 1 (or more) specifies that a property must exist for the referenced object to be valid.
The general constraints are defined as follows and more specific constraints are derived from them:

$$\text{1:N relationship } r \in MARI \cup MGRI: \forall o \in NI \text{ such that } dest(r) = o \rightarrow \neg\exists r' \in RI \wedge r \neq r', \text{ such that } class(r') = class(r) \wedge dest(r') = o$$

$$\text{N:1 relationship } r \in MARI \cup MGRI: \forall o \in NI \text{ such that } origin(r) = o \rightarrow \neg\exists r' \in RI \wedge r \neq r', \text{ such that } class(r') = class(r) \wedge origin(r') = o$$

M:N relationships do not require any constraint.

Exclusivity. An important type of constraint on relationships is the *exclusivity* constraint (xor constraint in UML; or constraint in OMT). It can be said that the existence of one relationship implies the non-existence of another. Exclusivity means that there can exist only one of n relationships starting from one object. In the example in Figure 20, new subclasses of the LinkToType relationship can be added. Indeed, it was said in section 2 that taxonomists use different kinds of taxonomic types that mean different things (e.g. lectotype, isotype, holotype). It was also said that the existence of one kind of type for a name (e.g. holotype) implies the non existence of another (e.g. lectotype). This situation is illustrated in Figure 15.
The followinf is an example of exclusivity between relationships of class R and relationships of class R'. Both share the same class as origin (R and R' may be identical in order to create an exclusivity rule at instance level):

$$\forall r \in R \wedge \forall r' \in R' \wedge r' \neq r \rightarrow origin(r') \neq origin(r)$$

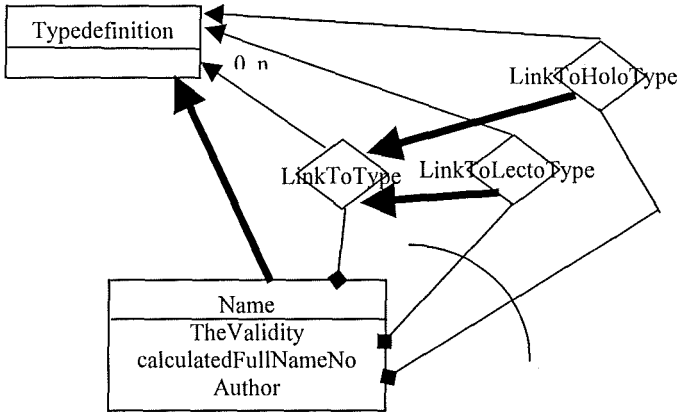


Figure 15: Exclusivity

Sharability. It is a property used mainly by aggregation relationships, although not restricted to them. In aggregations, non-sharing is often necessary, in particular when composition is represented (e.g. UML's composition). Sharability means that an object (e.g. a part) could only be referred to by only one other (e.g. a composite). In the example Figure 20, a different implementation of herbaria can be considered. Instead of representing herbaria as a string in the Specimen class, they could be independent objects that contain Specimen objects. (Figure 16). Then it can be said that each Specimen object belongs to only one herbarium. Indeed, it can only be in one place at a time.

The following is an example of an exclusivity constraint between a relationship r of class R and a relationship r' of class R'. R and R' may be identical in order to create a sharability rule at instance level:

$$\forall r' \in R' \wedge r' \neq r \rightarrow destination(r') \neq destination(r)$$

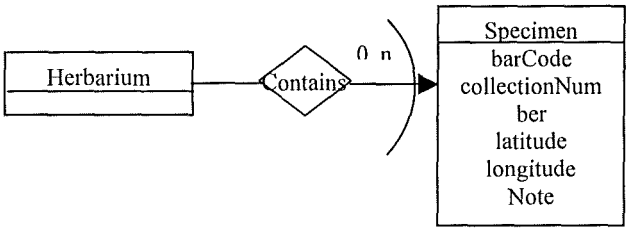


Figure 16: Sharability

Note that the previously defined *privacy* attribute does not imply non-sharability. Privacy only implies that it is not possible for external objects to know that a private object is referred to by an object via a relationship. This does not mean that the private object cannot be shared with other objects, privately or publicly.

Encapsulation. Encapsulation restricts the possible use of the part object of an aggregation relationship. For example, it can be defined that an aggregate object cannot be referred to by an association, as it would violate the encapsulation of parts, but can be referred to by an aggregation if sharability is allowed. Encapsulation is not the same as privacy. Indeed, privacy states that no object can find out the

objects targeted by a private relationship, whereas encapsulation states that those objects can or cannot be shared through specific relationships.

Example of encapsulation constraint on relationship $r \in MARI \cup MGRI$:

$$\forall o \in NI \text{ such that } dest(r) = o \rightarrow \neg\exists r' \in MARI \cup SARI, \text{ such that } dest(r') = o$$

By using the method described here, any of the relationships described in sections 4.4.1 and 4.4.2 can be created by grouping basic properties, given that unmanageable properties have been ignored (e.g. homeomeric).

4.4.5 Attribute inheritance

Attribute inheritance has been suggested as a property of aggregation relationships (Henderson-Sellers [Henderson-Sellers '97], Pirotte [Pirotte '97]) and implemented to a certain extent in ADAM [Díaz '90]. Attribute inheritance simply means that whole objects involved in a part-whole relationship inherit part of their properties (attributes and behaviour) from their parts. For example a boat, which is an aggregation of a hull and other parts, floats because it is composed of a hull and the hull floats. The "floats" attribute on the Hull object is therefore inherited by the Boat object through the isComposedOf relationship (Figure 17).

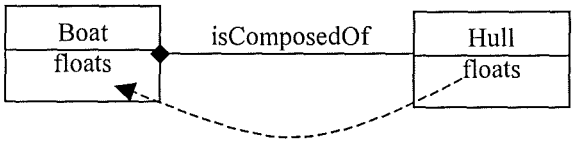


Figure 17: attribute inheritance in aggregation relationship

ADAM does not restrict attribute inheritance to aggregation relationships and allows any object to inherit attribute from any relationship they are involved in. The particularity of ADAM is that attributes are inherited from the relationship object, not from the objects involved in the relationship. For example, a Person object can inherit a salary attribute from the relationship that links it to a Company object (Figure 18).

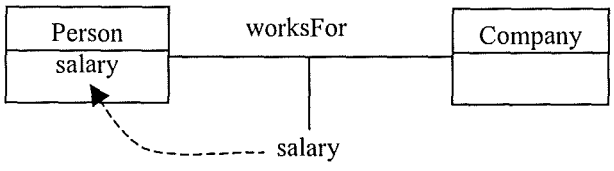


Figure 18: Attribute inheritance in ADAM

Attribute inheritance in the case of aggregation relationships can be restricted to certain kinds of aggregations. For example, in Odell's list of aggregation relationships (Table 1), member-bunch object composition and member-partnership object composition relationships can clearly not support attribute inheritance, as they do not convey any kind of structural or functional relationship between the part and

the whole (they are simply membership relationships). On the contrary, although this is not a necessary feature (except for Henderson-Sellers), other kinds of aggregation relationships can imply attribute inheritance (e.g. component-integral object composition, material-object composition, portion-object composition, and place-area composition which all imply a structural of functional relationship between part and whole, therefore possibly an influence of one on the other). This distinction was overlooked by Pirotte [Pirotte '97].

4.4.6 Collections

Relationships in general can be seen as simple collections. For example, creating a 1:N relationships between objects of class A and objects of class B implies that objects of class A can contain a certain number of objects of class B, or more generally that objects of class A are related to a certain number of objects of class B (collections do not imply membership).

However, these collections are very simple and do not account for all possible types of collections. For example, in ODMG collections can be bags (no order, duplicates), sets (no order, no duplicates), lists (order, duplicates), and arrays (order, duplicates). Just implementing collections as relationships would lose semantics, and that is one thing Prometheus endeavoured to avoid. These semantics are usually ignored by semantic relationship models, but ordering of relationships is described in UML. However, the need to prohibit the duplication of objects in collection relationships is never discussed.

As was said in sections 4.4.3 and 4.4.4, relationships in Prometheus can implement behaviours via attributes and constraints. These behaviours are not restricted to the semantics of relationships such as lifetime dependency or sharability. Attributes and constraints can be used to simulate the behaviour of various collections:

Bag:
Bags imply no specific order and no interdiction of duplicates in the collection. This kind of collection is naturally managed by complex-cardinality relationships in Prometheus.

Set:
Sets imply no specific order, but prohibit the presence of duplicates in the collection. A constraint can be devised to enforce this as follows:
This restriction can be enforced by implementing the following *uniqueness* constraint on the collection relationship R:

$$\forall r \in R \wedge \forall r' \in R \wedge r \neq r', \text{ such that } origin(r) = origin(r') \rightarrow dest(r) \neq dest(r')$$

Note that this holds because there can be only one relationship class with a given name starting from a given node class.

List:

Lists do not imply prohibition of duplication, but require order. Unlike *uniqueness*, order cannot be implemented through constraints. Instead, attributes can be used. Ordered relationships therefore implement an *order* attribute of type Integer that allows the software to describe the position in the list of each relationship. Then sorting is possible on these relationships and ordered retrieval of objects supported by the system.

Array:

Arrays are similar to lists in their semantics, therefore array relationships also implement the *order* attribute.

In addition, this approach has the advantage of clarifying the meaning of the collections that are represented. For example, in ODMG complex cardinality relationships are captured by collections and inverse references. Without reading the code that manages these collections, it is impossible to know what semantics they have or should have. On the contrary, in Prometheus, if the relationship that is used to capture the collections is an aggregation, then there is a part-of meaning to it. If it is an association; then it only shows that there exists a relation between the objects involved, but no part-of semantics. The representation of collections through collection objects loses this information, which makes it hard to implement certain behaviours.

4.4.7 Compatibility

All combinations of the attributes and constraints defined cannot always be applied simultaneously. For example, exclusivity and sharability cannot refer to the same pair of relationships. Table 3 shows these possible combinations of attributes. The system can make sure that basic properties are not used in an inconsistent way.

	Travers.	Lifetime dep.	Chang.	Privacy	Cardinal.	Exclusiv.	Shar.	Encap.
Navigation		✓	✓	✓	✓	✓	✓	✓
Lifetime dependendy	✓		×	✓	✓	✓	✓	✓
Changeability	✓	×		✓	✓	✓	✓	✓
Privacy	✓	✓	✓		✓	✓	✓	✓
Cardinality	✓	✓	✓	✓		✓	✓	✓
Exclusivity	✓	✓	✓	✓	✓		×	✓
Sharability	✓	✓	✓	✓	✓	×		✓
Encapsulation	✓	✓	✓	✓	✓	✓	✓	

Table 3: Allowed combinations of behaviours

4.5 Instance synonyms

A feature implemented in Prometheus that is not directly motivated by classification mechanisms in general is instance synonyms. Instance synonyms are structures that declare two things equivalent (nodes in the graph model, objects in the object-oriented model). Equivalence means that although these things are physically different in the database, they refer to the same real-world entity. But for many reasons, not design reasons rather historical and usage reasons, many forms of this real-world entity exist in the database. For example, misspelling may generate many copies of the same real-world entity. As these database entities are used in applications, it is not easy (or even sensible) to replace them by a single "right" entity. This could be for example because the different entities represent physical things that must be recorded, even though they are wrong (wrongly published names in taxonomy are in that situation). This situation is especially likely when various independent data sources are merged. Therefore, all copies of the real-world entity must be kept. However, if these different entities are simply kept in the database, it is hard (if not impossible) to query them and retrieve as much information as possible about them. As they are different, they are considered as distinct and unrelated entities by the query engine and only the explicitly specified entities are considered in the query evaluation. It is nevertheless possible to want to consider all existing forms of an entity in a single query. Enumerating these entities explicitly in a query may be complicated and is not generic (the user would have to know each of these entities). Therefore Prometheus provides instance synonyms.

Instance synonyms allow the definition of groups of objects that are not necessarily related by any available information (information in databases represent the application domain, not the real world, therefore information may be fragmentary and incomplete). Once a synonymy group is defined, it is possible to specify in queries that it must be taken into account. In that case, when the query engine encounters an object, it looks up the instance synonym groups the object belongs to, and instead of using the object in the query evaluation (e.g. path expression), it uses all members of the groups. In this way, it is possible to gather as much information as possible.

As an example, let us consider the schema presented in Figure 20. The database may contain two objects of type Author, one whose surname is Raguenaud and one whose misspelled surname is Ragenaud. These two objects have distinct OIDs, therefore are distinct objects. But for a human user, it seems possible that these objects refer to the same real-world entity. As these two objects come from different data sources (for example a staff list and a conference attendees' list), they are used in different applications and it is necessary to keep them separate (as they have been used for example in reimbursements). The first object, as it is the name that appears on the paper published at the conference is related to Publication objects. The second however, as it is a misspelled name, does not have publications, but has instead been used in conference arrangements in a separate application. If the two data sources are merged (e.g. in a financial database), inconsistencies may arise. For example,

Ragenaud has paid author fees for the conference, but does not appear as the author of a paper in the proceedings. Raguenaud appears as an author, but has not been to the conference. To overcome this problem the two instances may be joined by an instance synonym object and later used in queries (see section 5.1.1.1).

The definition of instance synonyms is not automatic. In some cases this could be automated by using secondary attributes to find similarities between objects, but this may not work in the general case. In Prometheus instance synonyms are created by users using their understanding of the data. Thereafter, they are managed automatically by the system.

Instance synonyms are noted as follows:

$I_1 \equiv I_2$, I_1 and I_2 are instance synonyms

This is not a rôle problem, as rôles are not taken into account in queries. All reviewed rôle proposals in chapter 3 ignore the query aspect of rôles and restrict their use to object structures.

4.6 Classifications

As discussed in chapter 3, the classification mechanisms proposed as suitable for managing biological classifications were shown to be unsuitable for representing the full complexity of taxonomic classifications. A new mechanism has therefore been devised for this work, which is described in this section. First, the technique for representing classifications is explained. This mechanism allows the representation of single and multiple overlapping classifications. Then it is shown how the representation of multiple classifications does not impair the ability of the system to retain single points of views.

4.6.1 Relationships as classifiers

An important feature of taxonomic data is that the objects should be independent from their classification. It would make no sense to design specimens for example so that they can be classified: specimens are simply specimens and should not be aware of their classification status (especially since they might not always appear in a classification). Mixing the description of real world objects with their ability to be classified would also increase their complexity and reduce reuse and maintainability. This independence of relationships between objects is already used to represent associations, as they should not be part of the object's types (e.g. [Norrie '93]). This feature is therefore essential to this new approach.

Two additional requirements are identified: the ability to store information in relationships, and the ability to describe the semantics of the relationships. The ability to store information in the

relationships is necessary to describe the classifications, especially in the case where classification information is meaningful and may affect the classified objects. The ability to describe the semantics of relationships (e.g. aggregation vs. association, sharing vs. non-sharing) is necessary to be able to distinguish what the classifying and classified entities are (identified as a set of aggregated objects), and what the classifications are (distinct from the definition, or type, of the objects classified). This is in particular necessary and true for non part-of classifications. Indeed, all classifications are not based on the physical decomposition of things. Some are more generally the relationship between classes or objects (they then become ontologies). Odell also declares that classification inclusion is not a member-bunch relationship. That relationship would imply spatial, temporal, or social connection, whereas the "classification relationship is based on the idea that a common concept applies to [the categorised objects]" [Odell '94b]. These classifications are not created using aggregation relationships, but more general association relationships. The distinction can further be enforced by creating a generic classification relationship of which all relationships used in classifications are a subclass. This way, even when classifications are part-of classifications, it is possible to distinguish the classification relationships from the relationships that participate in the definition of objects (e.g. node objects in a classification).

This new approach is to use relationships with the equivalent of weights (in weighted graphs). The first step toward a solution is the definition of a model that allows the description of graphs. Such systems exist: they are graph-based databases (e.g. Telos, ConceptBase, Progres, Hyperlog). However, these graphs are simple graphs that consist of nodes and edges (especially in the case of Hyperlog where attributes cannot be defined on edges). They do not support the definition of overlapping graphs that need to be unambiguously identified, as they do not support the definition of semantics for relationships.

The approach described here uses classes/objects as nodes, and relationship classes/relationship objects as edges in an OODB as they are more expressive. As the work is carried out in an object-oriented environment, these weights are not limited to simple integer values: they can be of any type defined in the system (including other edges). The definition of weights on relationships allows the description of the distinct trees with their overlaps.

Relationships effectively act as classifiers (or classifying mechanism). The action of creating such a relationship between two objects implies that these objects are classified. Furthermore, these relationships are the only objects in the system that are aware of the classifications and they contain all the necessary information to distinguish them from each other.

Figure 19 shows how the use of relationships as classifiers allows the description of multiple overlapping classifications. Each of these classifications represents a specific opinion, i.e. the context in which the specimens are classified. Figure 19 shows three distinct classifications: a dashed line classification, a thin line classification, and a thick line classification. In taxonomy, these distinct

classifications would have been published by distinct authors and the publication information would replace the type of arrow in this example. The leaf nodes in these classifications could be for example books or specimens. The other nodes can be book subjects or taxa that are used to classify the leaf nodes. It can be seen in the diagram that the classifications have elements in common: node 3 appears in the thin line classification and in the dashed line classification; node 4 appears in all three classifications. On the contrary, node 5 only appears in the thick line classification. Likewise, the leaf nodes can appear in one classification (node a), in two classifications (node b), or in all three (node e).

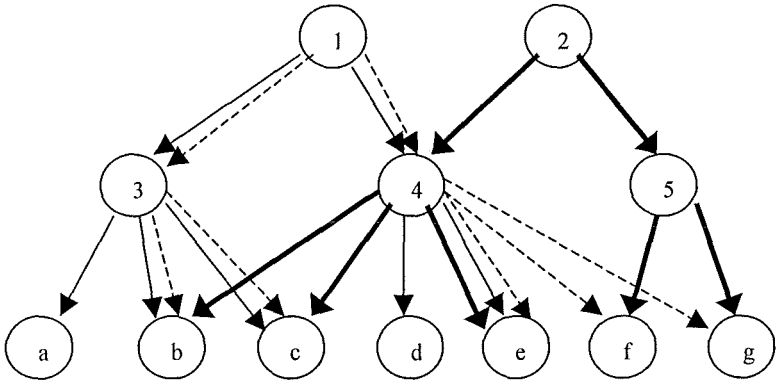


Figure 19: Multiple classification example

4.6.2 Classifying in context

This new approach allows the generic classification of entities by context. By "context", one can understand "anything that uniquely identifies a view". In plant taxonomy, this can be a taxonomist, a publication, or a combination of both. For example, one taxonomist's view on the world is a context and in that context a set of specimens is classified in a certain way. Concurrently, another taxonomist's view of the world represents another context where the same specimens (or any other set of specimens) are classified differently. The overall graph that is stored in the database represents a view of taxonomy out of context, or within all contexts concurrently. This view, although it is the most complete because it contains all existing information, does not suit taxonomy work, as taxonomists tend to work in one particular context or in relation to a limited set of contexts for comparison purposes. By representing classification information on the hierarchies that constitute that graph, Prometheus captures single contexts that can be extracted as necessary.

Because the distinct hierarchies created in different contexts overlap (in terms of specimens and names), the representation of all contexts in a single graph makes possible the comparison of classifications defined in different contexts and provides the ability to switch between contexts in order to gain knowledge. Indeed, by following relationships with specific values (e.g. publication information), it is possible to follow a path of a specific graph. But by switching between these values, it is possible to compare and navigate within and amongst classifications. For example in Figure 19, it is possible to compare nodes 3 and 4 and thereby to realise that they have some leaf nodes in common.

This can give new insight into the data (e.g. in plant taxonomy when two groups partially contain the same specimens, they are partial synonyms). It is also possible to contrast the different meanings of node 4 according to the different classifications: it contains nodes d and e in the thin line classification, nodes e, f, and g in the dashed line classification, and nodes b, c, and e in the thick line classification.

4.7 Example

Now that the basic structures of the model have been described, a sample schema can be defined and used in subsequent sections to support examples (Figure 20). As was said in the previous sections, the Prometheus model supports both "normal" or node classes, and relationship classes. These relationship classes are used to represent any relationship that may exist between objects, complex or atomic, in the system. Their advantage is that they allow the representation of graph structures without designing the objects that are part of the graphs (nodes and leaves) so that they can be part of that graph.

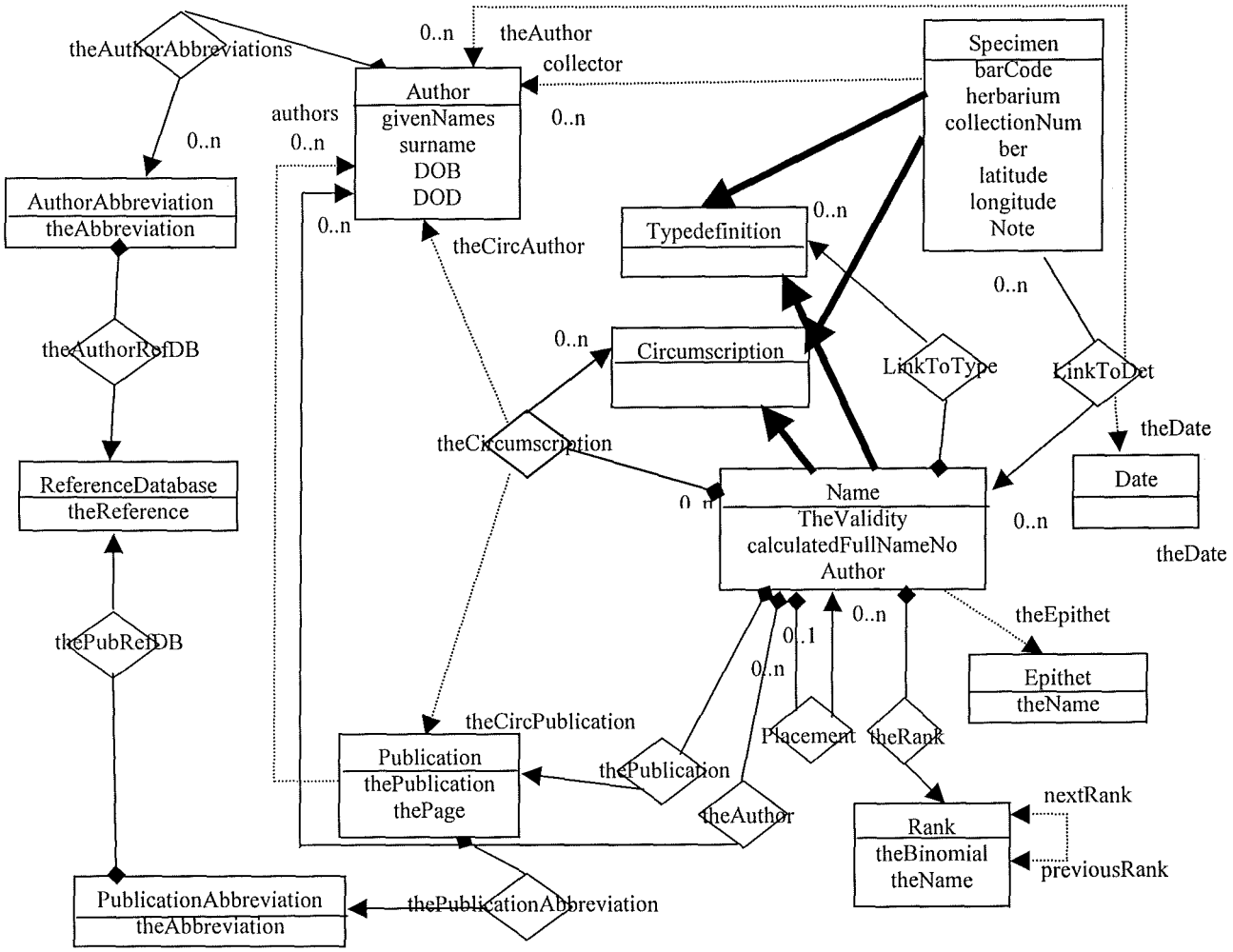


Figure 20: Sample schema

The diagram voluntarily uses most possible semantics.

The important sections of the schema in Figure 20 are the following:

- The hierarchy of names (or NT in Figure 6) is constituted of the Name, Specimen, and TypeDefinition classes, along with the LinkToType relationship class. Each Name may have many taxonomic types that it targets through a LinkToType relationship. Taxonomic types can be either Specimens or other Names. Note that inheritance and polymorphism are used in order to provide genericity.

  All the objects essential to the definition of a Name from a taxonomic point of view are related to that name via aggregation relationships. This makes possible the automatic handling or extraction of Name concepts and clearly delimits its borders.
- The classification hierarchies are constituted of the Name, Specimen, and Circumscription classes, along with the theCircumscription relationship class. The relationship class starts from a Name and targets the other Names or the Specimens that have been placed in it. Each theCircumscription relationship is parameterised by its "context", i.e. the Publication in which it appears and the Author that created it. Therefore, a single Name object (or any kind of object in other application domains) may have many circumscriptions, represented by a set of theCircumscription relationships, that can be distinguished using their context, i.e. a Publication and an Author in Figure 20.

Note that this approach is different from the approach described in section 2.3. This is due to two important ideas:

- The approach presented here represents the end-result of the process of classifying. Indeed, instead of representing how taxonomists work and supporting all the processes necessary (e.g. dynamic naming of taxa), it represents the data as it is when names have been derived and applied to classification groups (CTs). This allows better performance, as there is no need to recalculate the names each time one is required.
- The approach presented in section 2.3 is very specific to plant taxonomy, as the ability to create such classifications relies on the fact that the classified entities can be divided into two parts: a classification part (CT) and a naming part (NT). The classification part can be duplicated in order to represent single classifications. On the other hand, the naming part, which must be unique, is removed from the classification so that there is never any overlap between two classifications of the same concepts. This approach is not generic and is likely to be inappropriate in most domains necessitating classifications.

The approach described in section 2.3 is clearly limiting: if the entities to be classified cannot be divided into a classification part and a unique naming part, it is not possible to classify them. For this work, another approach has been chosen. In the current approach, classifications can classify any entity, whether they can be divided into classification and naming parts or not. Therefore, the classifications have elements in common and overlap at many levels (nodes or leaves), as was shown in Figure 19.

4.8 Notes on this model

This section discusses the following questions:

- If relationships are offered and can replace references, why have references been defined in the model?
- If relationships behave like simple collections, why have collections been included in the model?
- How can attribute inheritance be handled?

4.8.1 The reference problem

It was shown that the Prometheus model proposes simultaneously classical references and explicit relationships. Some database models proposed in the literature (e.g. OMS) that support relationships take this approach and have limited the use of relationships to associations, thereby requiring that references (i.e. all that is not an explicit relationship) are aggregations (as they are part of the type of objects). However, it was said earlier that relationships should have semantics whenever possible so that implementation of design models is straightforward and verifiable, and relationships are always unambiguous. Representing aggregations as references does not support the definition of clear semantics for these aggregations. These models are therefore unsuitable in their representation of design models (for lack of expressiveness or simplistic features) and force the user to change their semantics in order to allow their implementation.

References are however provided by Prometheus for the following reasons:

- All object-oriented database models propose references as a main feature and it would be hard to lose the habit of managing applications without them. Indeed, even modelling methods (e.g. UML) that should not be implementation dependent propose structures that represent references (e.g. complex attributes in classes).
- References are defined in these models and removing them from the models would change their semantics, something that should be avoided.
- Atomic objects are rarely targeted by aggregation relationships and some modellers do not support the idea of aggregating atomic objects. Instead, they are defined in the type of the objects that they are attributes of.
- The support for references alongside relationships is necessary for a compatibility problem: many application schemas exist and it would be a waste of time to force user to redesign them in order to use this new database model. One of the goals of Prometheus was to be integrated smoothly into an existing database model.

Instead of changing years' long programming habits and because Prometheus has been described to be compatible with an existing object-oriented model, it continues to provide references in types. These references should however be used within guidelines: no object should be targeted by a reference if it is

not an atomic object (e.g. text string, integer). References to Atomic values can be made through both explicit relationships (when semantics are necessary, e.g. non-changeability) and object references (when the type-based approach is chosen and when there is no need for specific semantics). In addition, Prometheus treats references like aggregations whose semantics are undefined (no lifetime dependency for example). Therefore, as will be explained in section 5.1.1, the operations provided by Prometheus that work on aggregations include references. Eventually, references should be abandoned because they are imprecise and ambiguous.

4.8.2 Collections

ODMG describes collections as a basic feature offered to database implementers. However, relationship-based models such as GraphDB or ConceptBase do not offer these collections. The lack of this feature can be understood because relationships that have complex cardinalities (e.g. 1:N, M:N) behave as simple collections. In Prometheus, collections are offered through relationships that can implement the behaviour of different collections via constraints (section 4.4.6). Their relationships not only implement the behaviour of collections, but can clarify the semantics of these collections, which ODMG, as with most other object-oriented database models, do not.

However, once again, the existence of a large amount of programs built with collections and the fact that ODMG explicitly describes collections makes it hard to ignore them as independent objects. Therefore, as for references, collections are offered in Prometheus as a backward compatibility feature, but their use is discouraged. Later, as Prometheus itself makes no use of collections to support its model and the features of that model, collections could be removed from the Prometheus model and only relationships left, without loss of semantics or flexibility.

Note that collections may still exist at the system level. For example in order to manage collections of instances as class extents, as query collection results, or as inheritance class lists that are not part of the multiple classification mechanism in Prometheus.

4.8.3 Attribute inheritance

Attribute inheritance (see section 4.4.5) is a feature that can improve object modelling (more information is captured in the model, therefore more checks can be performed and more exact information captured), support automatic inference of properties (a boat floats because it has a hull), and provide a mechanism for rôle acquisition (a person object inherits a salary from a relationship with a company, making it an employee object).

However, practical considerations must not be ignored. Attribute inheritance requires a fully dynamic programming or design environment. It requires for example, that an object could acquire attributes at

runtime because of the relationships it is involved in. This requirement imposes important restrictions on the languages that can be used to implement the models. Indeed, very few languages offer that degree of flexibility. Most systems requiring dynamic attribute redefinition (e.g. [Kuno '95b], [Gottlob '96]) have been implemented in Smalltalk and the features they use (modification of the message sending mechanism) are not available in most other programming languages.

Therefore, although its importance is noted, attribute inheritance would not be implemented in Prometheus if it didn't not use Smalltalk or Lisp as its programming language.

4.9 Conclusion

This work was motivated by the necessity to develop a database system to support the working practices of botanical taxonomists, an application where the storage and manipulation of complex overlapping graph structures is central. This chapter has shown how support for multiple overlapping classifications can be achieved. The proposed method is the provision of first-class relationships that capture all classification information and fulfil the requirements expressed in section 2.4.

Relationships have been integrated to an ODMG-based object-oriented model. This chapter has argued that the proper representation of relationships in a database environment is important for many reasons. It allows differentiation in the semantics of relationships (e.g. aggregation vs. association), saves the user maintaining pointers with inverse pointers, offers the ability to embed information in relationships, supports the definition of constraints, and supports the definition of complex classifications. Most object-oriented database systems do not support relationships, and a few of them support them partially. A new model has therefore been defined and described.

It could be argued that designing and developing a new database and query language was unnecessary and that existing systems could have sufficed. For example a standard OODB could have been used to build a one-off taxonomic database application where the necessary data structures were defined and methods written on these data structures for manipulation. However, in taking this approach where the semantics of the application was entirely hard-wired into the application code:

- The system would have been hard to develop because of the complexity of the concepts
- It would have meant mixing the semantics of the structures and the semantics of the user domain, thereby preventing the reuse of the generic concepts.
- The final system would have been inflexible and would not have been able to cope easily with changes or unanticipated uses without the requirement for further code to be developed.
- It would have been hard to validate the implementation against the design as the mapping from object model to implementation is not direct and the implementation of some concepts in code could possibly be scattered over many classes or modules.
- It would have ruled out the possibility of query optimisation.

- It would have put the burden on the application developer, as the concepts would need to be implemented more than once.

Another approach that could have been taken would have been to extend the database model artificially using standard classes with user defined semantics. For example, relationships could have been replaced by normal objects that could capture additional attributes. However, it would have been harder to write queries in a query language that does not handle relationships, and some would have been impossible. The query language would have had to be emulated using methods on objects, possibly called from the main query language. For example, recursion would have had to be managed by methods on the objects that are to be recursively used. This would have once again created a very specialised environment which would be hard to maintain, evolve, and use.

As discussed earlier, some database systems provide similar support for relationships, however they do not offer all the structures and mechanisms needed. Table 4 compares the features specifically offered by the Prometheus model to the other systems.

	ODMG	SORAC	OMS	GraphDB	ADAM	AGO91	POOM
Relationships as first-class objects	×	✓	✓	✓	✓	✓	✓
Single association	×	×	✓	×	✓	✓	✓
Multiple association	×	×	✓	×	✓	✓	✓
Single aggregation	×	×	✓	×	✓	✓	✓
Multiple aggregation	×	×	✓	×	✓	✓	✓
Attributes on relationships	×	×	✓	×	✓	✓	✓
Subclassing of relationships	×	×	**in part**	✓	✓	✓	✓
Constraints on relationships	×	✓	**in part**	×	✓	✓	✓

Table 4: Comparative table

By providing relationships as first-class objects and representing their meaning (that can be extended by the user), Prometheus offers an implementation environment which better reflects the modelling constructs used in designing complex applications. Implementation is facilitated by the more direct mapping between an object model and the implemented models.

Prometheus was designed to be simple, but provides a framework that can be easily extended to capture most possible nuances of the concepts offered. For example, containers (collections) do not represent all collection types available in ODMG (e.g. sets, bags, and arrays). But as containers are represented as objects, they can easily be subtyped and refined in order to support the definition of all collections. Prometheus does not capture all the possible semantics attached to relationships. However it provides a framework in which relationships can be extended or refined (through inheritance) in order to capture

more specific meaning. Since relationships are based on the standard representation of objects (but are treated as special objects), they support behaviour.

This chapter has also explained that relationships can be used to represent multiple overlapping classifications in a generic way. Relationships can indeed capture classification information independently of the classified objects, which allows the integration of the mechanism with existing databases/database models, and the ability to represent classifications where the classified objects are not designed to be classified. This section has also shown that this approach allows the clear distinction between classifications and classified objects through the combination of semantic relationships and classification relationships.

The achievements of the extended object-oriented can be compared with the taxonomic requirements expressed in section 2.4.

1. **Tree/graph structure**: the ability to use objects and relationships as first-class objects makes the representation of hierarchies (or graphs) a straightforward operation. Relationships represent edges and non relationship objects represent nodes. The representation of edges by first-class objects allows the storage of complex graphs (e.g. weighted graphs).
2. **Directed graphs**: the relationships defined in Prometheus are directed, i.e. they always have a preferred direction, even when they represent n-ary relationships.
3. **Multiple classifications**: multiple classification of entities is supported by the description of the classification information (e.g. publication, author) in relationships created between the classified elements. The selection of relationships with specific values provides a means to extract specific trees/graphs from the overall database graph. Each of these graphs represents a classification in taxonomy.
4. **Traceability**: traceability is supported in the same way multiple classifications are supported: classifying relationships can contain the reasons for particular classifications, e.g. character description information or the history of taxa as in HICLAS.
5. **Composite objects**: Prometheus supports semantically rich relationships that allow the description of composite objects, whatever the semantics of these relationships are. The semantics can be parameterised of redefined to fit the user's needs. Composite objects are also well integrated with the query language that can treat then and extract them as a single entity.
6. **Population-based classifications**: through the definition of classifications using relationships, it is possible to define classifications that involve objects that have not been designed to be classified. Prometheus therefore provides a generic way to define classifications. These classifications are population based classifications instead of category based classifications.
7. **Rôles**: rôles can be acquired through the involvement of objects in particular relationships. For example, a specimen involved in a taxonomic type relationship becomes a type specimen. If necessary, the objects that acquire rôles can acquire information through the

encapsulation of these relationships via methods (in a similar way to ADAM [Díaz '90], but not via the migration of attributes from relationships to objects).

10. **Integration with existing system**: as classifications can be represented by relationships, and since Prometheus supports the definition of first-class relationships, the integration of classification mechanisms in Prometheus is straightforward. In addition, the integration of a relationship mechanism in an existing database system allows the representation and manipulation of trees/graphs in a system that did not previously provide this facility. It would be for example possible to integrate Prometheus with a taxonomic database that does not support multiple classifications in order to add that feature.

11. **Generic classifications**: through the specialisation of the relationships used to describe classifications, any kind of classification can be defined, including non is-a or is-of classifications. Specific relationships can be used to represent special types of classifications. If necessary, it is also possible to capture is-of classifications.

12. **Orthogonality of classification and classified information**: because Prometheus supports the definition of composite objects, it is possible to make a clear distinction between the classification information and the classified objects. Indeed, classified objects would be represented as composite objects, whereas classifications would be represented by another kind of relationships. In addition, as was shown in section 4.6, all classification relationships can be created as a subclass of a specific generic classification relationship so that even in the cases classifications are part-of relationships, it is always possible to distinguish classification relationships from other relationships.

5 Queries and rules

As well as a new database model, multiple overlapping classifications in general and plant taxonomy in particular require an adequate query language and rules/constraints. The query language must allow the full exploitation of the constructs provided by the database model and provide operators that support the processes involved in the type of applications tackled by this work. The rules/constraints provide a means to enforce working practices and ensure that the data is correct. This chapter is therefore divided into two parts: the description of the query language, then the description of rules/constraints.

5.1 Queries

This section presents the query language offered by Prometheus. It is based on OQL (as the model is based on ODMG) and is extended with several operators that make working with explicit relationships and classifications possible.

Two aspects of the query language must be presented: the syntax and the semantics of the language. The next section describes these aspects. Then comments are made regarding several aspects of the query language.

5.1.1 Syntax and semantics

Because this document presents a model that extends classical object-oriented models with explicit relationships, it is necessary to extend the query language so that it can manage these relationships (POOL). The basis that has been chosen for extension is OQL. This choice is motivated by the fact that it is a well-defined language, that it is a standard, and because the system proposed has an ODMG implementation.

The query language is based on a subset of OQL (mostly select queries) with extensions to manipulate relationships and structures such as graphs defined using them. This document only shows the differences and improvements brought to OQL. First, this document shows how the differences in the model affect the query language, then it defines the new operators required for manipulating relationships, and finally it describes how graph manipulations are supported.

5.1.1.1 Model

Collections:

Navigation of attributes in OQL is performed using the dot notation. In OQL, the dot operator can be applied only to individual objects. Although ODMG has typed collections, which would allow their use

in collections, they cannot be used in selections and path expressions. This makes the expression of some queries harder. It has been shown (e.g. [Barclay '94]) that collection traversal is a practical feature in object-oriented query languages. This collection traversal has been implemented in Prometheus.

For example:

(Q1) `select p from Publication p where p.authors.surname = "Raguenaud"`

Here `authors` is a collection of Author objects defined in the class Publication (instead of a relationship). The query simply traverses the collection by considering each member in turn and continuing the evaluation of the query from there.

At each step of query evaluation:

> Given an object o ∈ Object, op(o.collection) := ∀ x ∈ o.collection → op(x), i.e. when an operator (e.g. the dot operator) is applied to a collection belonging to an object o, the operator is applied to all elements of that collection.

Note however, that this collection traversal only affects models that have not been implemented according to the guidelines expressed in section 4.8. Indeed, as Prometheus offers relationships with constraints that emulate the behaviour of collections, object-oriented collections should be a rare feature of Prometheus schemas. Collection relationships, as they are relationships, can be traversed naturally in path expressions. The traversal of object-oriented collections is in part offered to harmonise the creation of path expressions.

In the case of bindings where no support for parameterised polymorphism is offered (e.g. Java), using collections in path expressions could be incorrect from a type point of view. Indeed, the only type a collection may contain in Java is Object. Therefore, only messages that the Object interface or class can understand can be sent to objects contained in the collection. For these bindings, another operator that is defined (downcast operator, later described) can be used in conjunction with smart collections.

Instance synonyms:

The keyword "using instance synonyms" can be used in POOL queries to specify that all instance synonyms must be evaluated in queries. (Q2) shows such a query. It returns all publications published by Ragenaud (which is a misspelled name). Without the `using instance synonyms` keyword, no publication would be returned. But with the keyword, the query engine selects all Author objects that belong in the same groups as Raguenaud (the real author of the publication) and uses them in the evaluation of the query. Therefore, Ragenaud is seen as the author of the publication and a satisfactory answer is returned.

(Q2) `select * from Publication p where p.authors.surname = "Ragenaud" using instance synonyms`

At each step of query evaluation:

> $\forall$ op, if o1 $\equiv$ o2[4], op(o1) $\rightarrow$ op(o2), i.e. at each object encountered during query evaluation, if using instance synonyms is specified, all operators applied to an object o1 are applied to all its synonyms.

Note that the use of instance synonyms may have a significant impact on query evaluation. Indeed, they add objects to the search space, therefore slow down the query evaluation. They also require access to additional objects (the instance synonym structures and the synonyms themselves), therefore consume more processor time and increase the number of disk accesses, and makes optimisation harder.

5.1.1.2 Relationships

Relationship objects:

As relationships are classes, they can be used in queries as any other class (selection, projection, and from clause). Therefore, if attributes of a relationship object are to be queried, a relationship class name in a query can be followed by attributes of the relationship or attributes of the class that is targeted by the relationship. When a class A is related to a class B by a relationship R and x is an attribute name, the interpretation of the path A.R.x is: if x is defined in R, then A.R.x refers to the attribute x of R. If x is not defined on R, A.R.x refers to the attribute x of B. However, relationship classes contain the predefined attributes origin and destination, which can be used to specify that A.R.x refers to x in B, which is noted A.R.destination.x. In essence, the dot operator acts as an implicit join.

The dot operator in POOL differs from the dot operator in OQL and the ODMG model. Two cases must be distinguished: 1 argument and 2 arguments.

First, the situation where there is only one argument is presented. This case is divided into two possible situations: the argument can be a property defined on the object under study, or it can be the name of a class. If the argument is the name of a method defined on the object under study:

> Given an object o which is the current object and m the name of a property defined on class(o) (or inherited by class(o)), o.m := o_m, o_m of type C_m, where C_m is the type returned by the property (attribute or method) m.

If the argument is a relationship class name:

> Given an object o which is the current object in query evaluation and a relationship class rc from the relationship class set RC, o.rc = X := $\{\exists\ r \mid class(r) = rc \wedge rc \in RC\} \wedge (r.origin = o \vee r.destination = o) \rightarrow X := X \cup \{r\}$.

Second, the situation where there are at least two arguments is presented. Once again, two cases must be distinguished: the case where the first argument is a method name of the object under study, and the case where the first argument is a relationship class name.

[4] $\equiv$ was presented in section 4.5 and means that the two objects are equivalent.

When the first argument is a method name:

> Given an object o which is the current object and two properties names m_1 and m_2, $o.m_1.m_2 := (o.m_1).m_2$. In other words, the problem is simplified to the first case proposed above.

When the first argument is a relationship class name:

> Given an object o which is the current object, a relationship class rc from RC, and m a method name or attribute name: $o.rc = X := \{\exists\ r \mid class(r) = rc \wedge rc \in RC\} \wedge (r.origin = o \vee r.destination = o) \rightarrow X := X \cup \{r\}$.

Then if the second argument, m is an attribute of the relationship object returned by the evaluation of the first dot operation, then dot operator is applied with one argument to the relationship object.

> $\forall\ x \in X$, $x.m := o_m$, o_m of type C_m, where C_m is the type returned by the property (attribute or method) m.

Otherwise, it is assumed that an implicit traversal of relationship was specified, and the relationship as required is traversed and the second argument is applied to the object targeted by the relationship:

> $\forall\ x \in X \wedge x.origin = z_1 \wedge x.destination = z_2 \rightarrow x.m := z_1.m \cup z_2.m$.

Aggregate objects:

In Prometheus, the projection clause is extended so that it can explicitly request composite objects to be returned, i.e. an object and all the objects it contains through an aggregation relationship. Shallow and deep aggregates are expressible, i.e. it is possible to extract not only one level of aggregation, but to the terminal components of a complex object. The concept is similar to the one of deep and shallow equality in some object-oriented databases [Khoshafian '86]. For example, POOL can extract a Name and its components easily:

(Q3) `select shallow aggregate n from Name n`

The result of such a query is a Name object along with its subparts as designated by aggregation relationships to the first level only (e.g. Publication, Author, Type).

The aggregate operator returns collections of objects that participate in the definition of the composite object o. The result of a query involving the aggregate operator is therefore a collection of collections. The deep aggregate operator is similarly defined by recursive application of the aggregate operator on itself until no more results are found. Note that the type of the collection containing each of the aggregate objects is *Object*. This is due to the fact that the type of the objects that are potentially returned by the operator is unknown.

The formal definition of `aggregate` is as follows:

> Given an object o, $aggregate(o) = X := \forall\ R \leq Aggregation \wedge \forall\ r \in R \wedge r.origin = o \rightarrow X := X \cup \{r.destination\}$
> $\wedge\ \forall\ m \in class(o) \rightarrow X := X \cup o.m$.

The use of the aggregate operator is compatible with clauses such as *order by*. As the result is a collection of collections, this operator can be applied to collections, providing that they support ordering (for example by extending an Ordered interface). *Distinct* remains applicable, as it is applicable to collections (on the basis of identifiers for object type collections, on the basis of their content for literal collections). *Group by* does not apply because collections cannot be grouped according to a particular criterion; this is not particular to Prometheus. Prometheus in itself does not change the typing of such queries.

Note that as was explained in section 4.8.1, references are included in the result of the extraction of an aggregate object, as references are seen as simple aggregation in Prometheus.

Selective downcast operator:

OQL provides a cast operator which allows the restriction of the type of an object to one of its subclasses. However, it can only perform this modification if the type of the returned object is known in advance and conforms to the type specified. When a schema contains a recursive structures (e.g. part-explosion schemas), a composite pattern [Gamma '94], or more generally, when inheritance is used in the schema, this cast operator may be inadequate. In the taxonomic schema of Figure 20, this situation appears in the taxonomic type hierarchy and in classification hierarchies.

The subclass of the composite class or superclass is chosen and only use instances of this class in the query using the downcast operator noted by square brackets after an attribute. The downcast operator does not generate errors when instances of another type are encountered; they are simply ignored, distinguishing it from the traditional cast operator. For example:

```
(Q4)  select n from Name n where
   n.theCircumscription[Specimen].barCode = "E000001"
```

This query finds Name objects that contain Specimens whose bar code is E000001 and is type safe. It shows how it is possible to query a link (reference, container, and relationship) and select the returned type.

The definition of the selective downcast operator is as follows:

Given a current object o, [ ](o, B) := if class(o) = B → [ ](o, B) = x.

It is possible to express an equivalent of these queries in OQL with queries such as:

```
(Q5)  select n from Name n, Specimen s where s.barCode = "E000001"
   and s in n.theCircumscription
```

However, this notation breaks path expressions and when the schema makes heavy use of inheritance, queries become hard to understand. In addition, it requires that theCircumscription be implemented as a collection instead of a relationship.

Although this problem occurs often in the composite pattern (for which this solution was designed), it is a general problem when subclasses are chosen in path expressions. It also occurs when path expressions can be recursive. In this situation, it is impossible to specify recursive statements involving a join, as it would mean transforming the whole selection into a recursive statement. The problem also occurs in bindings where parameterised polymorphism does not exist (e.g. the Java binding). In that situation, all objects in a collection are of type *Object*, which leads to the inability to query collections accurately. For example, the following query is not possible in OQL if theCircumscription is a collection:

(Q6)
```
select n from Name n, p.theCircumscription a where a.barCode =
"E000001"
```

The variable a is of class *Object* in Java because n.theCircumscription is of type collection, therefore the attribute is not defined in its type. In Prometheus, the following statement is valid and equivalent to *(Q5)*:

(Q7)
```
select n from name n, n.theCircumscription[Specimen] a where
a.barCode = "E000001"
```

Note that this operator is not dependent on parameterised polymorphism. Indeed, it was designed to simplify queries that need to control inheritance, not to force casts that might then fail. Therefore, although it behaves in a similar way to cast operators, it does not raise exceptions or report errors when it encounters objects that are not of the expected type. As explained above, in this case these objects are discarded. Therefore, although it can be used to make queries possible in such environments (e.g. Java), it does not act as a real cast operator. In fact, a static cast operator would be impossible in Java because there would be no type information on the collection on which to base the type checking of path expressions involving the collection. However, it provides the necessary type cast when the type of the collection is consistently maintained.

5.1.1.3 Graphs

Orientation and navigation:

It is sometimes useful to be able to specify the direction relationships should be traversed. If no direction is specified, traversal of relationships is bi-directional. However, if the direction of the relationship is important, this can be specified using the keywords "origin" or "destination". The direction that can be taken is moderated by the definition of the relationship, i.e. a relationship can only be traversed from origin to destination if the traversal property of the relationship allows it. Note that the use of "origin" and "destination" is only necessary on recursive relationships in order to choose the direction of navigation. In unambiguous cases the navigation is automatically chosen by the system.

For example, it can be necessary to find all the Names that contain another Name called "graveolens" at the next level:

(Q8)
```
select n from Name n where
n.theCircumscription.destination[Name].theEpithet.theName =
"graveolens"
```

Conversely, Names that are contained in circumscriptions of Name "Apium" can be found:

```
(Q9)  select n from Name n where
   n.theCircumscription.origin.theEpithet.theName = "Apium"
```

In OQL, relationships are represented by references to objects with their inverses. It is therefore inherent in OQL to be able to choose how to traverse relationships.

Transitive closure:

When objects form graphs (a graph of parts where component parts are shared for example), it is sometimes desirable to be able to simply specify the repetition of segments of paths; which requires support for transitive closure. OQL is unable to represent recursive queries. POOL supports operators which allow the building of simple regular expressions over path segments. It also allows the expression of recursion. The operators supported are: ? for optionality, + for one of more repetitions, and * for zero or more repetitions.

"?", which specifies an optional path

$? := o_1.a_1. \ldots .a_i.(a_j. \ldots .a_k)?.a_l. \ldots .a_m = \alpha$, $i>0$, $j>j$, $k>j$, $l>k$, $m>l$, o_1 is an instance, $a_x \in$ Relationship, $x < m$

$\leftrightarrow o_1.a_1. \ldots .a_i.a_l. \ldots .a_m = \alpha$

$\wedge o_1.a_1. \ldots .a_i.a_j. \ldots .a_k.a_l. \ldots .a_m = \alpha$

"+", which specifies the repetition of a path strictly once or more

$+ := o_1.a_1. \ldots .a_i.(a_j. \ldots .a_k)+.a_l. \ldots .a_m = \alpha$, $i>0$, $j>j$, $k>j$, $l>k$, $m>l$, o_1 is an instance, $a_x \in$ Relationship, $x < m$

$\leftrightarrow o_1.a_1. \ldots .a_i.a_j. \ldots .a_k.a_l. \ldots .a_m = \alpha$

$\wedge o_1.a_1. \ldots .a_i.a_j. \ldots .a_k.(a_j. \ldots .a_k)+.a_l. \ldots .a_m = \alpha$

"*", which specifies the repetition of a path between 0 and n times.

$* := o_1.a_1. \ldots .a_i.(a_j. \ldots .a_k)*.a_l. \ldots .a_m = \alpha$, $i>0$, $j>j$, $k>j$, $l>k$, $m>l$, o_1 is an instance, $a_x \in$ Relationship, $x < m$

$\leftrightarrow o_1.a_1. \ldots .a_i.a_l. \ldots .a_m = \alpha$

$\wedge o_1.a_1. \ldots .a_i.a_j. \ldots .a_k.a_l. \ldots .a_m = \alpha$

$\wedge o_1.a_1. \ldots .a_i.a_j. \ldots .a_k.(a_j. \ldots .a_k)*.a_l. \ldots .a_m = \alpha$

For example, to find the components that contain graveolens at any level:

```
(Q10) select n from Name n where
   n.(theCircumscription.destination)*.Name = "graveolens"
```

Here a relationship is repeated a number of times and the direction of traversal is specified. More complex patterns can be defined in a similar way. By omitting "destination" or "origin", both descendants and ancestors are retrieved (if the schema permits it from a navigation point of view).

Extracting subgraphs:

An important feature of Prometheus is the extraction of complete sub-graphs, which can later be used as complex objects, either to be displayed to a user, or to be transformed into views. However, because multiple logical graphs might satisfy the search criterion, many subgraphs can be extracted at once. For example, complex parts might be of interest and the graphs they form extracted. This is achieved by the `follow` clause in POOL, which explicitly states that all members of the returned set must be related through a specified relationship class.

`follow` returns a collection of graphs formed by extracting subgraphs from the result of the query where elements of each subgraph are related by the specified relationship. As `follow` acts as a filter on the result of the query, the relationship to be followed must be specified in the `select` clause. Because the data is not restricted to trees and allows the sharing of objects (e.g. in overlapping graphs), it is possible for elements to appear in more than one subgraph isolated by a `follow` clause. Such a query can be expressed in POOL as follows:

```
(Q11) select n, bag<n.theCircumscription> from Name n follow
   theCircumscription
```

This result would consist of many collections, one for each graph that is identified by a "context".

More formally:

$$c \text{ is a collection, follow}(c) := \{C_i\} \wedge \forall\, r_1 \in C_i, r_2 \in C_i \wedge r_1, r_2 \in \text{Relationship} \wedge \text{context}(r_1) = \text{context}(r_2).$$

POOL first extracts all instances of the class supplied as an argument. Then it looks in already extracted sub-graphs to find whether the origin of the relationship in hand is already defined in one of them. If so, the relationship and its destination are added to the subgraph and other subgraphs extracted are considered. If the origin of the relationship is not defined in sub-graphs already extracted, a new one is created and the relationship, its origin and its destination are added to it, then evaluation continues for other subgraphs. The result of the evaluation of the `follow` clause is a collection of collections where each of the contained collections holds a connected graph (objects and relationships, therefore the nested collections contain instances of type *Object*). Relationships are also traversed in a consistent way. Because of the representation of logical graphs, relationships are only traversed if they have the same type value. Therefore, when a relationship is traversed, POOL records the attribute values of that relationship, and thereafter only traverses relationships without attributes or those with the same values.

`Follow` is similar in its behaviour to the *group by* operator in OQL in that it groups objects in a certain way and it changes the type of the structure returned by the query. It is an operator that is applied to the results of a query. Note that in order to function, `follow` needs to flatten the collection it receives as an argument.

Integrity of graphs in path expression evaluations

The representation of distinct graphs or hierarchies in Prometheus depends on the use of relationships that contain enough information to represent the scope of these hierarchies. When they are to be queried, it is often necessary to make sure that the evaluation of queries does not violate the limits of hierarchies, i.e. it is necessary to make sure that the paths that are to be traversed do not switch between hierarchies[5]. This is easily ensured by comparing the value of relationships at each step of the evaluation of queries. When a path is traversed, the next relationship encountered is only traversed if its type value is consistent with the previously traversed relationships. This is declared using the `in context` clause (formerly called `xlink` in some papers) without argument. For example:

```
(Q12) select n from Name n where
   n.(theCircumscription.destination)*.theEpithet.theName =
   "graveolens" in context
```

This query will find Name objects that contain the Name "graveolens" as in (Q10), but stay within the scope of one classification once it has traversed one of the theCircumscription relationships. For example it might traverse the theCircumscription relationship depending on its Publication and only continue to follow parts of the component created on that date. `in context` can be used with `follow` in order to extract sub-graphs defined by attributes on a relationship. In which case, `follow` returns subgraphs identified by values of attributes on relationships and no longer only by connection between objects.

Recursive joins and contexts

Recursivity was introduced in OQL as explained above. This recursive notation allows the repetitive traversal of paths or patterns. Since these patterns are likely to involve relationships, which are defined in the model as objects containing attributes, it must be possible to select the relationships that are to be traversed according to their attribute values (selective retrieval). In OQL queries, this is achieved by join statements in the selection clause. These joins result in tuples that can then be filtered through the projection. However, OQL, and indeed all other object-oriented query languages do not support a conjunction of explicit relationships with attributes and recursive behaviour. This conjunction makes the declaration of recursive traversal of relationships problematic. In the case of relationships, implicit joins already exist because traversing a relationship from an object to another is equivalent to creating two joins that find the relationship that links two objects. As these joins can be implicit, they are not shown to the user. Therefore, their presence in recursive statements is not a problem. However, when the joins are to be explicitly defined by the user, problems arise. The difficulty is to describe a join in structures that are recursive, thereby prohibiting the usual join notation. Using a classical join notation in recursive queries would break the sense of path in the queries, as they would have to be divided in many paths joined by a statement. Example:

```
(Q13) select c.origin from Name n, theCircimscription c where
   n.theEpithet.theName = "graveolens" and c.destination = n
```

This OQL query finds the Name objects that contain another Name called "graveolens" at the next level. As is shown, there is no longer the feeling of defining a path. In addition, it is impossible to represent a recursive statement by the definition of a recursive join. The solution adopted was to externalise the definition of recursive joins that involve relationships. If these joins are not defined as other joins, it is clearly possible to express them. The solution has two steps. First, aliases are introduced inside path expressions. These aliases, (as all other aliases/variables defined in OQL) are declared in the from clause of the query and used as attributes of objects. Example:

```
(Q14) select n from Name n, theCircumscription c where
   n.(c.destination[Name])*.theEpithet.theName = "graveolens"
```

This POOL query finds all Names having a sub-Name called "graveolens" at any depth as in ***(Q9)***, but the relationship theCirumscription is replaced by an alias (c in the example). This introduction of aliases in path expressions brings the problem of precedence in the evaluation of all statements in the selection clause. Indeed, a case exists where a part of the path expression can only be evaluated after another path expression has been evaluated, but that second path expression might also be dependent on the evaluation of the first one. This problem is similar to the precedence problem in joins.

The second step is the definition of the join criterion outside the selection clause. This is achieved by using the `in context` clause with a selection clause as argument. It selects the relationships that are to be considered for the rest of the evaluation of the query. In essence, this is the creation of a transient view on the overall database graph that then serves as the basis for the evaluation of the query. This also solves the precedence problem since `in context` selection clauses are evaluated first (and only once), and are kept independent from other path expressions by the syntactic checker. Example:

```
(Q15) select n from Name n, theCircumscription c where
   p.(c.destination[Name])*.theEpithet.theName = "graveolens" in
   context where c.theCircAuthor.surname = "Linnaeus"
```

This query finds all Names that contain another Name called "graveolens" at any depth as in ***(Q11)***, but only according to the relationship definition that is declared in the `in context` clause, i.e. that was published by Linnaeus. In fact, `in context` behaves as a selection in this case which describes that that the variable `c` is defined in a certain way in the context that is used (e.g. the author is Linnaeus).

The functioning of `in context` differs from other selections in two ways: firstly it selects relationships (hence its former name of `xlink`) and secondly the result of that selection can be used as variables in path expressions. Therefore this is the preferred notation for the selection of relationships. Another notation could have been to include the selection of relationships in the selection part of the main query as shown in (Q16). However, the use of variables in path expression in this context would be confusing and against OQL's understanding of variables.

[5] This feature is inherent in the semantics of the botanical taxonomy for which Prometheus is designed.

```
(Q16) select n from Name n, theCircumscription c where
   p.(c.destination[Name])*.theEpithet.theName = "graveolens" and
   c.theCircAuthor.surname = "Linnaeus"
```

The advantages of this notation is that is keeps the feeling of path which simplifies the comprehension and writing of queries, it allows the definition of recursive explicit joins, and it makes a clear difference between joins that concern normal objects and joins that concern relationships.

5.1.2 Notes on POOL

Five aspects of the query language proposed are discussed here: the limitation to select queries, the object creation/conservation approach, method calls, type checking, and set comparisons/sub-queries.

5.1.2.1 Select only queries

As it is apparent in previous sections, only "select" queries are supported in the current implementation of POOL. This restriction is due to the following facts:

- The focus of this work is on modelling of classification and taxonomic data and its retrieval. The implementation of update and delete operators was therefore not a priority.
- Select queries are an important part of the queries run on a database system, and it was felt that this aspect of the query language was a large enough area to experiment with new operators and new database structures.

These queries are sufficient to answer all the most common needs of taxonomists (see section 7.1). In addition, as OQL is a read-only query language where updates can only be performed through object method calls, strictly OQL-based query languages could not provide update operators. This may restrict however the ability of the query language to provide a powerful interface to the database, as any modification of that database can only be performed if they have been implemented on objects.

As was shown in [Güting '94], restructuring query language operators may be of interest to graph-based operations, for example for the creation of new circuits. OQL-based query languages may therefore be more restrictive than other approaches regarding the possibility of extensions to the query language.

5.1.2.2 Object conservation

As can be seen in section 5.1.1, POOL is an object preserving query language. The choice of this approach was motivated firstly by the fact that OQL until version 1.2 was an object preserving query language that only returned existing objects or set of objects or tuples. Later versions of OQL can

create instances of existing classes only. For this first prototype, the object conservation approach was chosen for simplicity.

Secondly, the creation of new objects brings many problems regarding class creation and integration, and object identity management [Kifer '92] [Kuno '95a] [Mitchell '91]. As one of the requirements of this work was its integration into an existing database system, the creation of new objects would have required the modification of the OID management mechanism in the underlying database. Although in theory possible, this was judged impractical.

5.1.2.3 Methods

It is sometimes important to be able to call methods in queries. For example in taxonomy, the name of a taxon is calculated on the fly in a function using the specimens (and type specimens) that are included in it. Whenever the name of the taxon is required, it can be computed by recursively selecting the specimens that were placed in it, then finding out what is the earliest name that applies to that set of specimens. However, in the current implementation of POOL, methods calls are not allowed. Although this is a restriction, two things should be noted:

- Method calls can be extremely expensive. For example, the method call described above would take a very long time because many graph traversals would have to be performed, many specimens extracted, and many comparison of names executed. In that case, it was found more reasonable to materialise the name attribute and re-compute it when necessary.
- Although method calls are not implemented, there is nothing in POOL that makes it impossible. The addition of method calls would be fairly straightforward.

5.1.2.4 POOL and type checking

As POOL is based on OQL, it is a strongly typed query language. In other words, before the query is evaluated, its syntax and its type correctness are checked and badly typed queries are rejected. This allows query optimisation [Grumbach '99] and can be performed automatically [Riedel '97] and formally [Alagic '99].

However, type checking in POOL is an option that can be disabled if necessary. When this is the case, no type checking is performed and the query is evaluated naively by following references and relationships are described in the query, regardless of whether these paths are correct. If a path is not correct, no exception is raised and the evaluation continues.

This feature is particularly useful in the context of semi-structured data. As the structure of the data is not always known or can be incomplete in these systems, type checking in queries is hard, if not impossible. Although Prometheus is currently implemented on top of an object-oriented database, future or alternative implementations could be built in other environments, e.g. graph-based or semi-

structured models. By allowing POOL to ignore type, it is provided it with the ability to be ported to new, very different systems.

5.1.2.5 Set comparisons/sub-queries (in)

OQL does not define the use of the `in` keyword in the SQL sense. Such an operator is defined, but only applies to variables and path expressions, not on sub-queries (according to BNF).

POOL provides such a feature, but restricts it to specific contexts. In order to use `in`, the sub-query must return OIDs or values only, i.e. it cannot select multiple fields or structures and must only select objects. The semantics of the operator is then a comparison of OIDs only (type is not taken into account, as two identical OIDs necessarily mean that the two references point to the same object).

This operator is useful for two reasons: it allows easy and intuitive specification of membership and non-membership (e.g. a specimen should appear in one classification but not in another one), and it allows the manipulation of graphs as entities (e.g. graphs can be extracted with sub-queries and compared via the result of the sub-queries, without the creation of specific structures as in [Güting '94]).

5.2 Rules/Constraints

In previous chapters, constraints have been described in an abstract manner, without presenting specifically how they are implemented. This section describes the Prometheus implementation of such constraints, and more generally of rules in Prometheus.

As was explained in section 4.4.4, the Prometheus model relies on constraints to capture more semantics in relationships. The list of constraints presented in previous sections is not limiting but rather only shows some of the basic built-in constraints offered in Prometheus. They have been chosen because they are common and of general use. However, since the semantics of relationships may vary (e.g. UML's taxonomy of aggregations [OMG '99] compared to Odell's [Odell '94b]), they are not definitive and can be changed in Prometheus to suit the user's needs. As can be seen, the constraints must necessarily satisfy many rôles: invariants, pre-conditions, and post-conditions. In addition, reaction to other events (e.g. committing of transactions, deletion of objects) may be of interest and are provided as well.

It has been presented that the types of constraints required for plant taxonomy are both intra- and interobject constraints. Intraobject constraints provide the ability to specify that the state of an object must obey certain rules. For example, in a database representing people, the Elderly class will have a constraint on the possible values of the age of the people represented by that class. Interobject constraints ensure that a particular object is in the right place in the database relative to other objects.

For example, in the case of relationships, an encapsulation constraint would check that all relationships that target the object that is targeted by the relationship are of type aggregation (and not association). It is therefore necessary to embed constraints both in objects (intraobject) and in relationships (interobject).

In the next sections, Prometheus' constraints are presented, then their execution strategy is explained.

5.2.1 Rules in Prometheus

Constraints are a special case of rules. Rules are traditionally defined as triples: they have an event part (E), that describes which events trigger the evaluation of the rule, a condition part (C), that checks whether an action must be taken, and an action part (A), that is the action to be taken in order to repair a problem. These rules are therefore referred to as ECA rules. Constraints can then be referred to as EC rules, since they have no action part (or a default transaction abortion). In Prometheus, the accent is put on constraints because the implementation of the ICBN is paramount to the ability to offer consistent structures and to derive information such as names from the available data. Since the ICBN only describes valid configurations of concepts, and not how to fix inconsistencies, rules can only report problems, not take action.

However, taxonomy constraints are context dependent, i.e. they need to check their own applicability. For example, a taxon at rank]Genus, Species][6] should be placed into another taxon at rank [Genus, Species[. Therefore taxa must have a rule that supports this constraint. This rule has obviously three parts: an event, a condition of applicability (is this taxon at rank]Genus, Species]?), and a condition (is this taxon placed into another taxon at rank [Genus, Species[?). In Prometheus; constraints therefore have an event part (E), a condition of applicability for the constraint (C), and a condition (C) that is the actual constraint. These rules can be referred to as ECC rules.

The object-oriented approach has been chosen to manage rules in Prometheus (e.g. as in ADAM [Díaz '91]). Instead of describing rules in external files or as lists of textual facts, Prometheus supports the definition of rules as objects. This approach has many advantages:

- It provides a uniform representation of schemas and rules.
- It allows the manipulation of rules as objects at runtime.
- It allows the manipulation of rules by other rules.
- It allows the description of rules that benefit from the computationally complete language offered by the system, as methods can be overridden and added.

[6] This concise notation represents ranges. A square bracket that opens to the outside of the range means that the element next to it is not included in the range. A square bracket that is closed toward the range means that the value next to it is included in the range. For example, [2,5[represents all values between 2 and 5, including 2, but excluding 5.]Genus, Species] means all ranks between Genus and Species, including Species but excluding Genus.

These rules can be embedded inside other objects and described at class level (as in Ode [Gehani '91]). If a rule is declared in a "normal" object, then it is an intraobject rule. If it is declared in a relationship object, it is an interobject rule. The definition of constraints inside objects has the advantage of providing a better readability of the schema, as all the information relating to a particular class is defined in that class. But because relationships are also defined as objects, they can carry interobject rules and avoid breaking encapsulation (e.g. SORAC [Doherty '93]). Note that although rules are embedded in objects, they can be extracted by the system if necessary and optimised independently from their owning objects.

5.2.1.1 Event

Rules in Prometheus can react to 6 events. Three of these events are related to the life of objects, and three to the life of the database. The events that are object-dependent are: OnCreate, OnChange, and OnDelete. These events are generated automatically by the database and rules can be attached to them. The three events that are dependent on the database state are OnTransactionBegin, OnTransactionCommit, and OnTransactionAbort. The choice of granularity is limited both by what is practical in commercial databases and what is necessary to support the type of applications to be managed (plant taxonomy). For example, events related to the occurrence of exceptions, based on time, or on external factors have not been defined. Exceptions are to be managed in the database application or by objects as they are executed since exceptions are results of their actions. Time is not a significant factor in taxonomy, and only very specialised applications require time awareness (temporal databases). Events outside the database are not important either, as plant taxonomy is a closed world that does not interact with the real world directly.

Events in Prometheus are only primitive events: they are the occurrence of a simple event in the database. Unlike systems like Ode [Gehani '91] or Jasmine [Ishikawa '93], Prometheus does not support fully the definition of composite events or user-defined events. The event part of the rule in Prometheus is captured by the class the rules belong to. For example, a rule that reacts to the creation of objects will be of class PrometheusOnCreateRule or one of its subclasses. A rule that reacts to the abortion of transactions will be of class PrometheusOnTransacrionAbortRule or one of its subclasses. This does not leave liberty of migrating rules between their possible rôles, but it would create too many problems to transform a rule into another and it would not be justified by the application domain.

Although Prometheus does not support composite events, since it is a full-featured object-oriented database, the system itself is constituted of objects that can be reached and queried. Therefore rules are represented by objects with a particular structure and a particular behaviour. In fact, they inherit from generic rule classes that implement the basic behaviour and structure for rules supported by Prometheus. Therefore, defining a rule that responds to the OnChange event is represented by subclassing the PrometheusOnChangeRule and extending its behaviour if necessary. It is then possible to create rules that respond to many distinct events by subclassing one or more basic rules. For example, in order to create a rule that responds to the OnChange and the TransactionCommit rules (but

not necessarily their conjunction), one can subclass the PrometheusOnChangeRule and the PrometheusOnTransactionCommitRule. This is equivalent to a composite event of the following form: OnChange *or* OnTransactionCommit. Other combinations of events such as *and* or *not* are not supported, therefore Prometheus does not claim to support fully composite rules. The implementation of more complex composite events was not undertaken for many reasons. Firstly, it was not necessary to satisfy the requirements on the application domain, and secondly they raise important efficiency and semantic problems [Paton '99].

The granularity of the event (whether it concerns all elements of a set, e.g. all instances of a class, or single objects) is not managed in the event part of the rule. This is managed by the rule itself, as can be seen in section 5.2.1.2.

The detection of events is problematic in Prometheus as queries can be complex, involve many objects and classes (path expressions, see section 5.2.1.2), and may be recursive (see section 5.2.3). Therefore, the association of a rule to an event on an object does not guarantee that all rules that may be affected by this event are detected. For example in taxonomy, an important rule is that a taxon placed at rank]Genus, Species] should be placed in another taxon at rank [Genus, Species[. Clearly, this constraint applies to taxa, therefore in Prometheus would be attached to taxa objects. However, events directly affecting taxa objects (creation/modification/deletion) are not the only events that should trigger the evaluation of the constraint. For example, the modification of the rank hierarchy, may invalidate this rule, as an object's rank may become above Genus. Constraints (ECC rules) are therefore declared with a list of entities that may have an effect on the validity of the constraint. These classes or objects are the entities that are monitored in the database that will trigger the evaluation of rules when they are subject to certain events. In essence, the rule attached to a particular class or object not only monitors that class or object, but a series of other classes. This allows a better use of the resources by minimising the opportunities when to evaluate rules.

In addition, as in Ode, rules can be declared as *once* rules or *perpetual* rules. Once rules are executed only once, then are deactivated for the object that contains them. These rules are rarely supported in databases and only Ode provides support for them. However, they are important in order to support a certain category of rules: pre-conditions and post-conditions related to object creation. Indeed, a pre-condition associated to the creation of an object (e.g. for a composite object the creation of its parts), must be executed only once in the life of the object. Therefore, once it has been executed, it is deactivated and no longer requires consideration by the rule manager. This reduces the number of rules to be inspected when events are triggered.

In Prometheus, there are no CA (Condition-Action) rules. All rules must be triggered by an event. Even though a rule can monitor the root (or the roots) of all inheritance hierarchies, therefore react to an event affecting any class, it has to be linked to a particular event (e.g. OnCreate). Although this choice is limiting in some aspects (some database systems choose to support CA rules, e.g. NAOS [Collet '94],

Sentinel [Chakravarthy '94]), it has proven sufficient for Prometheus' needs and was thought too expensive to implement. Prometheus however provides EA (Event-Action) rules, as when the condition is omitted, the action is evaluated after the event occurred.

5.2.1.2 Condition/Condition of applicability

The condition part of ECC constraints checks whether the conditions are appropriate to the evaluation of the action part of the rule. Constraints depend on configurations of objects in the database. Since these conditions are dynamic (e.g. the levels in the Ranks hierarchy), they cannot be resolved at compile time and must be checked each time a rule is triggered. This condition is referred to as condition of applicability. The condition of applicability is distinct from the constraint (section 5.2.1.3), as violation of the condition of applicability does not lead to a violation of the whole rule and action from the database manager (e.g. abortion of the transaction). It would not be possible either to combine condition of applicability and constraint, as violation of condition of applicability would lead to violation of constraint (e.g. *rank = "Familiy" and name like "%eae"*, would be violated if the rank is not Family, although it means that if the rank is not Family then the constraint does not apply).

Conditions must represent boolean expressions. If the expression resolves to false, then the condition is considered not activated or violated. Conditions are queries or PCL (Prometheus Constraint Language) declarations. This section focuses on query constraints, PCL constraints are studied in section 5.2.3. Figure 21 shows an example of such a condition. This condition specifies that the rule only applies to People objects that have children. A later action/condition could be that they are eligible for child benefits.

```
exist(select * from People p, p.children c where count(c) >0)
```

Figure 21: Example of constraint condition

Conditions specify the granularity of rules. Indeed, by default rules are defined for a class, therefore apply to all instances of the class. This is the consequence of defining constraints at design time in types/classes. However, at runtime, it is sometimes necessary to specify constraints on specific objects. For example, a particular object might gain access conditions at runtime and become accessible only for a set of other objects (e.g. private parts of a composite object). Or in taxonomy, a publication that does not follow the rules of the ICBN can imply that some elements of a classification need to respond to different rules than their siblings. Although the definition of granularity as part of the condition is not common, it is simple and yet powerful, as any kind of granularity can be described (e.g. a subset of the instances of a class, instances that obey a set of properties, an arbitrary list of instances, or single instances).

Because all constraints do not have a condition of applicability (they are absolute), the condition part of a rule can be ignored. In that case, instead of checking the condition, the rule manager executes the second condition (the proper constraint, EC rule) or the action (EA rule).

5.2.1.3 Action/Condition/Constraint

The action part of rules is called the action, or the second condition or constraint in Prometheus. It represents the action to be taken if the condition resolves to true (ECA rule), or the actual configuration that must not be violated (ECC rule or constraint). Rules are of two forms: rules that have no side effect on the state of the database (rules that only check the state of the database and allow an operation to take place or not), and rules that change the state of the database. Rules without side effects are constraints, whereas rules with side effects are active rules. Most of the rules in Prometheus are constraints. For constraints without side effects, the action part is another condition instead of an action. This is useful when constraints can only be evaluated in certain contexts and whose purpose is the enforcement of integrity. In this case, the condition of the constraint tests whether the constraint can be evaluated (whether the context is right), and the action part of the constraint performs the actual evaluation of the constraint if the condition was successful.

Constraints without side effects are limited in their expressive power to operations that cannot change the state of the database. For example, DELETE and INSERT operations are banned. This is due to the fact that integrity constraints such as invariants should not change the state of the database, but rather ensure that the database is in an acceptable state and report violations of these valid states. Most of Prometheus' rules are rules without side effects.

Rules with side effects enable the system to react to particular situations and take the proper actions to fix problems or carry particular actions. In Prometheus, all rules that involve modification of the state of the database, directly through the definition of the rule or indirectly through execution of code, are considered rules with side effects. These rules are managed like rules instead of calls to methods because Prometheus is implementation independent and the mechanisms for creating and deleting objects are different in different implementation environments (e.g. there is no real destructor in Java, even the finalization method is invoked too late in the process of destruction of an object). Rules with side effects are useful for example for the propagation of operations. In the case of lifetime dependency, the deletion of parts when the whole is destroyed can be managed by rules attached to the composite relationships. In this case, the direction of relationships is important.

It is important to note that rules with side effects introduce important conflicts. Indeed, since the execution of a rule can change the objects present in the database, it can invalidate another already triggered rule (that will not find the objects it requires). Rules with side effects therefore introduce the concept of priority. Each rule should be defined relative to all others so that when many rules with side effects are triggered, they are evaluated in a sensible order. These rules also introduce the need for database state history, as rules may have access to different states of the database when they are

executed. For example, it can be stated that rules are always executed in the context in which they have been triggered, knowing that this state may be out of date once other rules have been evaluated. In addition, since rules with side-effect can manipulate other rules (they are objects), the execution of rules introduces an important degree of unpredictability. It is indeed impossible to guess what the result of the execution of a set of rules might be without actually executing them. This approach introduces a great complexity in the definition of constraints (for example the need to know and understand all the constraints in a given system at once) and as Prometheus makes little use of such rules, it was not felt during the development that such complex mechanisms were needed. Prometheus only implements a naïve approach to the evaluation of these constraints: they are evaluated in the same order they were fired.

5.2.1.4 Types of rules

Constraints (rules without side effect) can be of different kinds: invariant, pre-condition, and post-condition. It was shown in section 4.4 that invariability is an important feature of semantically rich relationships. It was also said in section 4.2 that pre-conditions are necessary to the life of a database model (e.g. ensuring that a class name does not already exist when a new class is created). These different kinds of constraints are described here. In addition, a special kind of rule, relationship rules, is discussed in this section.

5.2.1.4.1 Invariants

As was said in section 4.4, invariance is a central feature of semantically rich relationships. This is in fact the one feature that can be automatically checked in many modelling approaches (e.g. Odell, Henderson-Sellers).

Invariants are always called when an object is created, changed, or deleted, whether it is the object containing the invariant itself or an object that has been declared as having an influence on the constraint (in the list of monitored classes). In Jasmine, invariants would be composite object created from two primitive events (OnCreate and OnChange). An invariant is the following combination of primitive events: OnCreate *or* OnChange.

5.2.1.4.2 Pre-conditions

In Prometheus, rules can also be explicitly triggered by the user. These rules are then not attached to a particular object but are independent rules that are directly sent to the rule scheduler by an action of the user. The rules declared as pre- or post-conditions must first be immediately checked. Since they need to check that the database or a particular object are in a consistent state before or after the execution of a method, they need to be evaluated as soon as the need appears.

The execution of pre-conditions must be triggered by the user by writing a call to the appropriate method at the beginning of a method code. This includes the constructor of objects. They are therefore not automatically managed by the system.

For example, in the case of multi-valued relationships, checking the number of relationships in one object before adding a new one may be important. If the cardinality is for example 5, before adding a new relationship the system must ensure that the cardinality is not already 5. If it is, then the new relationship cannot be added and an exception is raised.

5.2.1.4.3 Post-conditions

Post-conditions are executed at the end of a section of code. As with pre-conditions, they must be immediately checked, as any problem with a piece of code must be detected as soon as possible to avoid a knock on effect on other parts of the system. They can be called by the user in the application code at the end of a method.

In the above example, check that it is one more, after the relationship has been created, the system can check that the cardinality has been increased by one.

Pre- and post- conditions allow programming by contract [Meyer '97].

5.2.1.4.4 Relationship rules

These rules are necessary for two reasons: they refer to the way two objects are related more than to the objects themselves; they require breaking encapsulation. SORAC showed that such rules are in fact relationships rules instead of object rules that can be implemented with the monitor construct.

The creation of relationship rules has the advantage of not requiring any normal object in the system to violate the encapsulation of others and at the same time being aware of all the objects involved in the rule as they are targeted by the relationship. It can be argued that such rules, although they avoid breaking the encapsulation for non relationship objects, requires the relationships to break encapsulation of these objects. This is true only if relationships are seen as normal objects. However, they are special objects that are treated differently than other objects. It is therefore acceptable to allow them to violate encapsulation, as it is acceptable to allow a query language to violate encapsulation [Cluet '98].

A typical example of such rules are rules that govern the relative salaries of employees in a company. It can be decided that employees should always earn less than their manager. The schema representing

this situation would have two classes, Employee and Manager, and a relationship between them for example called Works_For. Embedding the salary rule in either object would require that object to know the internal state of the other. Embedding the rule in both objects would require both objects to break encapsulation and in addition would become harder to maintain. In fact, the salary rule is not a rule that belongs to either objects, but belong to the relationship between these objects. It is the fact that an employee works for a manager that makes the check on salaries necessary. Similarly in taxonomy, the rank of names requires that some names be placed below others at specific ranks. Embedding the rule in names would require these names to break the encapsulation of others. In addition, once again, it is the fact that these names are related that makes the rule necessary. Therefore creating a relationship rule allows the representation of the rule where it belongs.

5.2.2 Execution strategy

The execution strategy of rules is composed of two parts: the scheduling, or the way rules are executed, and the error handling, or the way the system reacts when constraints are violated. The next two sections present these two phases.

5.2.2.1 Scheduling

Constraints must intuitively be triggered when an object to which they are attached is created/modified/deleted. But in addition, the rules need to be re-evaluated when changes occur elsewhere in the database that could affect the validity of stored rules. For example, when a Rank object is changed, all queries depending on the value of a rank or its place in the rank hierarchy (i.e. most taxonomic queries) are potentially affected. Therefore, rules also contain a list of all types on which they are dependent and that will trigger their re-evaluation. This list can be provided by the user or automatically generated by the system. When an instance of these types is created, deleted, or modified, the rule manager searches all the rules to find out which ones are dependent on this kind of object and triggers their evaluation. Constraints are re-evaluated when any instance of a type is created/modified/deleted because it is very hard (if not impossible) to tell which instance might be reached by which query before actually executing the query. In term of performance, this implies that many queries are evaluated unnecessarily, but it is hard to avoid.

Furthermore, it is necessary to define different kinds of rule execution strategies: *immediate* effect constraints, and *deferred* effect constraints. These two modes of evaluation refer to the timing between the triggering of a rule (the E part) and the evaluation of the condition (first C part). In Prometheus, this is the only controllable timing. Other systems also propose timing between condition (C) and action (A) or even the execution of rules in separate transactions.

Immediate effect constraints are constraints that are evaluated at the time they are triggered and can immediately invalidate the transaction or the attempted operation (e.g. hard constraints in Ode). These

constraints are often necessary in order to check the internal state of a concept (i.e. its aggregation relationships) because they define the meaning of the concept. They are also useful for controlling situations that cannot occur, even temporarily. For example, in taxonomy, a rank Species can never be the (direct or indirect) parent of a rank Genus.

Deferred effect constraints are only evaluated, as the current transaction is to be committed. These constraints are the equivalent of relationship rules in SORAC or soft constraints in Ode. This distinction is necessary because of the sequential aspect of database manipulations. Indeed, complex structures may be the subject of a constraint that needs to check the entire structure instead of each element of the structure. For example, let us imagine a database where a Person node must be linked to another Person node with a spouse arc, and the spouse node linked back to the first node with a similar edge (spouse is a bi-directional relationship). If an integrity constraint checks that the two spouse relationships are always opposite, during the update of arcs, the integrity rule may be violated. However, this may be only a temporary situation and the state of the database may be correct at the end of the transaction. The system therefore needs to wait until all changes are made before the validity of such constraints is checked.

The system uses a constraint manager that stores the triggered constraints in a queue when they are fired by an action in the database. When an event that might trigger constraints occurs in the database, the constraint manager is called and finds out what are the constraints attached to the triggering object. In a first phase, only immediate effect constraints are evaluated. There is no defined order in which constraints are tested. Only the order in which they were described and then triggered is followed.

Unlike constraints, rules may change the state of the system while they are evaluated. They may therefore affect the way the rule scheduler behaves and how the evaluated rules behave. For example, it is possible to imagine a rule whose action is to change other rules because that rule means that the system has switched to another mode that invalidates rules. That rule can therefore delete all rules at runtime, and the scheduler would not evaluate the other rules. As was said earlier, these rules with side effect introduce many problems (priority, ordering, parallelism, access to state, termination), which are not managed by Prometheus.

If all immediate constraints have been successful when the transaction is to be committed, the constraint manager executes all pending constraints not already evaluated.

In addition to the traditional event triggered mechanism described above, Prometheus supports the user-activation of constraints. At any time, the user can decide to simulate the occurrence of an event for a specific rule. This is particularly useful for example to manage pre- and post-conditions. Indeed, a pre- or post-condition is a rule that can be triggered by the user when the execution of the application enters or leave a specific section (or indeed at any time). In that case, the user can force the execution of a constraint, whatever the activation event that was described for that rule was. The rest of the

evaluation of the rule is as for any rule. Note however that only immediate rules make sense in this case, as a pre or post-condition that would be evaluated at the end of a transaction would not secure a specific section of the code.

If a rule is triggered by many events in the course of a transaction, it will be evaluated for each of the activations as it is queued by the scheduler at each occurrence. Unlike some other systems (e.g. Ode), Prometheus does not restrict the number of times a rule is executed, be it an immediate or a deferred rule, i.e. the transition granularity in Prometheus is 1:1.

The execution of rules is either sequential or parallel, either immediately after the firing of the rules or at commit time in the transaction for deferred rules. The choice of execution is left to the user and has great implications on the result of the evaluation of a set of rules. For example, if rules with side-effects are used, the state in which subsequent rules are executed may be different from the state in which the event occurred. For example, if a rule A monitors changes in the state of an object a and might result in the deletion of the object a, and rule B also monitors the object a, what is the meaning of rule B if it is executed after rule A? The object that it monitors no longer exists. On the contrary in parallel evaluation the implications of rules upon others is more limited, but more unpredictable as it depends on the time spent by the queries relatively. For example, if the same rules A and B are fired at the same time by the parallel scheduler, if A finished before the end of the execution of B, then the state of the database might have changed between the start of the execution of B and its ending.

It should be noted that, as Prometheus is a fully object-oriented database, rules are represented by objects. Therefore, it is conceivable that some rules change the way other rules are defined or the way the rule scheduler behaves. For example, the rule A above might change the content of rule B if certain conditions occur. For example, the fact that A is executed might imply that B becomes meaningless, therefore A might change B so that it does nothing. Likewise, if rule A is triggered, it might switch the mode of operation of the rule scheduler from sequential to parallel, or vice-versa. These actions are possible because the content of queries is based on the query language, therefore rules, as with any query, are able to break encapsulation. Although no direct use for these features has been found in taxonomy, they are provided inherently by the system.

5.2.2.2 Error handling

When an error is detected, an exception is raised. Unlike many other systems (e.g. SAMOS [Gatziu '91], Sentinel [Chakravarthy '94], NAOS [Collet '94], ADAM [Díaz '91]), Prometheus does not implicitly abort the transaction. The firing of an exception allows the user to choose how to deal with the problem. When an exception is raised, the user is able to know what went wrong and decide on the actions to take, which include aborting the transaction.

In addition, interaction with users is sometimes necessary, as only they might have the answer to a given problem. This is especially the case in taxonomy where taxonomists are free to make arbitrary

choices in certain situations, or the publication of erroneous data implies a legal violation of a set of rules. The interaction with the user is an important feature of Prometheus' rules. When a rule that can be ignored happens (once again the ability to be ignored is inherited from a specific rule type in Prometheus, PrometheusInteractiveRule), an automatic user interaction mechanism is fired (e.g. graphical, textual, network-based, dialog with another application), and the evaluation of other rules stops until a reply has been received. If the user chooses to ignore the violation of the constraint, the execution of scheduled rules carries on, whereas if the user does not wish to ignore the rule an exception is raised and the user application can take appropriate action.

5.2.3 PCL

In Prometheus, as with many relational (e.g. Progres, Ariel, SQL-3) and object-oriented (e.g. NAOS, Chimera) systems, conditions and actions in rules are represented by queries. This has many advantages: it allows the rule mechanism to benefit from query optimisations; it provides a powerful way to write rules as the query language can be very complex (including calls to object methods or external mechanisms); it does not require the implementation of additional mechanisms to implement the rule system. However, the definition of conditions and actions as queries is not necessarily straightforward. Indeed, queries are not object centric (they are global), therefore object-centric rules are harder to describe. Figure 22 shows such a constraint. This constraint specifies that a Person object whose OID is #OID must have a positive value as its age attribute.

```
exist(select * from People p where p.age > 0 and p = #OID)
```

Figure 22: Object-centric constraint

This type of constraint requires knowing the OID of the targeted object and to centre the constraint on that object. In most cases, this is impossible, as the OID of an object is not known at the time its class is defined or when the rule is described. There is therefore need for a specialised language that can then be automatically mapped to the query language. In Prometheus this language is called PCL and it is based on OCL [OMG '99].

The main reason why PCL is based on OCL is that, as an abstract database model that captures more semantics and offers a simpler environment by providing higher level concepts has been defined, the constraint language has been designed from an abstract point of view. This has the advantage of being simple and allowing formalisation, therefore it is understandable by specialised people as well as database designers. The other advantage of OCL is that constraints may consist of two parts: a first part that defines the context in which the constraint is applicable, and a second that represents the constraint itself, exactly as Prometheus' ECC rules (Figure 23).

```
Self.age > 21 implies self.registered_for_elections
```

Figure 23: PCL example #1

PCL is built on a subset of OCL. It borrows its form and part of its syntax from OCL, the rest being inspired from POOL. In the current implementation, PCL only covers a fraction of OCL. More specifically, it covers the general syntax and the portions that are close to OQL. In later versions, a complete implementation of OCL will be provided.

Although the type hierarchy is not clearly defined in OCL documents, it is assumed here that it is compatible with the type system of Prometheus or any implementation of Prometheus (e.g. ODMG).

5.2.3.1 Deficiencies of OCL

OCL is not based on a query language. Therefore, it does not define very powerful path expressions. However, these path expressions are essential to taxonomic constraints. In particular, recursive path expressions are often used to represent certain rules (e.g. ensuring the right placement of a taxon in a hierarchy of taxa according to their ranks).

5.2.3.2 Extensions to OCL

In Prometheus, OCL has been merged with the query language in order to offer more powerful constraints (PCL). For example, PCL can describe constraints that involve recursive path expressions, as shown in Figure 24. Figure 24 implements the constraint "if a name (taxon) is placed at rank Species, it must be placed in another taxon at rank [Genus, Species[". Self represents the object itself, as in OCL.

```
Self.rank.name = "Species" implies
Self.theCircumscription.origin.theRank.previous*.name = "Genus" and
Self.theCircumscription.origin.theRank.next+.name = "Species"
```

Figure 24: PCL example #2

5.2.3.3 Translation

As was explained in section 5.2.1, rules in Prometheus are queries. However, the description of such queries is not always straightforward or even possible (e.g. knowing OIDs at design time), which is why PCL has been described. Therefore the rules described with PCL must be translated into POOL. This translation is straightforward at runtime because all the necessary information is available. For example, the PCL constraint shown in Figure 24 would be translated in POOL as shown in Figure 25 (the event part of the rule is not shown, as it is not related to the translation into POOL).

```
Condition: exist(select * from Name n where n.oid = #OID and n.theRank.theName =
"Genus")
```

```
Constraint: exist(select * from Name n where n.oid = #OID and
n.theCircumscription.origin.theRank.previous*.theName = "Genus" and
n.theCircumscription.origin.theRank.next+.theName = "Species"
```

Figure 25: Translation of PCL into a Prometheus rule

5.3 Conclusion

This chapter has introduced the query language and the rule/constraint mechanism provided by Prometheus.

The introduction of relationships as first-class objects in the object model (chapter 4) requires that the query language deals with these new structures in a manner that provides to user applications the full benefit of these new structures. This chapter has therefore presented several operators designed to deal with relationships:

- The dot operator has been extended to deal with relationships in OQL. Relationships are manipulated as transparently as possible when ambiguity can be avoided. The traversal of a relationship occurs in the same way the traversal of a reference in OQL occurs: the name of the relationship is simply given. However, in some cases, ambiguity may arise. For example, relationships may have attributes that may be accessed by the user. The dot operator therefore also allows access to these attributes.
- A selective downcast operator has been introduced to deal with casting that does not raise exceptions if an object of unwanted type is encountered. This has been necessary to cope with the recursive structure of graphs. The selective downcast operator allows the selection of specific sub-classes in queries.
- As relationships are oriented but may support two-way navigation (section 4.4), navigation has been introduced in OQL. In order to make the differences in the model as invisible as possible, this navigation occurs by the specification of the source or destination attributes of relationships.
- As Prometheus supports the definition of semantic relationships through behaviour flags and constraints, it is important that the query language deals with these semantic them. Prometheus offers operators that are able to work on aggregations, for example for the extraction of composite objects in one single query.
- Prometheus also deals with classifications specifically. It is able to traverse weighted graphs consistently, i.e. it only follows relationships exhibiting a specific context, so that when graph traversal occurs, the query evaluation is bound to specific classifications. It also provide an operator that make both recursive joins and selection of context possible.

The query language satisfies the following generic requirement expressed in section 2.4.

9. **Recursive behaviour**: the query language provided by Prometheus supports the definition of recursive queries through regular path expressions, therefore is able to assist in the extraction of specimens contained by hierarchies of taxa.

Table 5 compares the features added to OQL by Prometheus to the query languages offered by the systems that provide explicit relationships.

	ODMG/ OQL	SORAC	OMS	GraphDB	ADAM	AGO91	POOM/ POOL
Traversing collections	×	N/A	✓	N/A	N/A	✓	✓
Selective downcast	×	N/A	in part	×	N/A	×	✓
Relationships in query language	×	N/A	✓	✓	N/A	✓	✓
Aggregate objects in query language	×	N/A	in part	×	N/A	×	✓
Bi-directionality	in part	N/A	✓	✓	N/A	✓	✓
Transitive closure	×	N/A	✓	✓	N/A	×	✓
Extraction of logical subgraphs	×	N/A	×	✓	N/A	×	✓

Table 5: Query languages comparative table

The support for rules and constraints in Prometheus is necessary for the enforcement of working practices (e.g. the ICBN). They have therefore been defined in the proposed system.

These rules and constraints are object-oriented, i.e. they are defined as objects and may be related to classes, therefore they are well integrated in the overall model. There are two reason for this choice: ODMG does not specify constraints, therefore defining new structures to capture rules and constraints would have led to significant changes to the standard. The implementation of rules and constraints as objects is very flexible and powerful, allowing for example the modification of rules by other rules or the creation of inheritance hierarchies of rules that support reuse.

The rules and constraints are generic in that they are specified by a language derived from OCL and they are mapped to the query language. The rules therefore benefit from the full power of the query language (e.g. graph-specific operators).

However, they are specific in the way they are represented: constraints are able to check their condition of applicability, which is necessary in the case where classifications are dynamic and the levels of the classification is represented by outside objects (ranks in plant taxonomy). They are also specific in that in order to accommodate the fact that taxonomist users ultimately make the decisions, even if the data is not correct (see chapter 2), rules that may be violated can be defined as interactive rules, in which case they will interact with the user to find out whether they are ignored by the user or not.

Their execution model, although simpler than some others in that it does not support the definition of all possible composite events, is powerful enough to support the definition and evaluation of all taxonomic rules. Simple composition of events is however possible through inheritance which permits the creation of more expressive event detection.

An advantage of these rules/constraints in that provision of explicit relationships with constraint/rules allows an easier way to check if the implemented model conforms to the object model: it provides a rapid application development environment where concepts (e.g. relationships) and constraints are not scattered, thereby making them easier to maintain [Doherty '93].

The constraints as defined in Prometheus satisfy the following requirement expressed in section 2.4.

> 8. **Rules/constraints**: Prometheus supports the definition of rules. These rules, as they are objects, can be changed at runtime, therefore deal with exceptions as they happen in taxonomy as the result of errors.

6 Architecture and implementation

The prototype is a framework that provides a set of predefined classes that can be used by user applications, or extended or specialised in order to suit specific user needs. For example, the use of Prometheus in conjunction with a specific modelling method such as UML or OMT may require the definition of relationships with special semantics. The framework allows the selection of these semantics from a list of available semantics to suit any use. Additional semantics can also be added if needed.

The prototype was built on top of an existing commercial object-oriented database system: POET [POET '01] and using Java [Sun '95] as a programming language. This was in accordance to the requirements expressed in section 2.4, in particular the integration of the prototype with existing systems to allow reuse of existing schemas and data.

This chapter provides implementation-related information about Prometheus and offers more precise information about some of its features (e.g. implementation of relationship semantics) and how these features can be used by user applications. Note that only a high-level view is presented, Prometheus consisting of 543 classes and 89686 lines of code, too many details would impair clarity.

First, this chapter presents Prometheus' high-level structure. Then the usage of the mechanisms offered by the prototype is shown.

6.1 Architecture of the OODB

The OODB structure is a layered architecture. The fact that the underlying database system needs to be accessible to user applications that use Prometheus requires that Prometheus sits as a package on top of the existing system and only provides additional features. The fact that the system is complex and calls to various levels of the system are performed by several other levels make it suitable to use a layered architecture approach. Therefore, that approach has been chosen. The overall structure of the system is presented in Figure 26.

However, it can be noted that the Event layer is in contact with all the other layers instead of being accessible through lower layers. This is due to the fact that Prometheus is an active database therefore events are required at several layers (e.g. for indexes and for firing rules). Other layers do not call services offered by the Event layer, instead the Event layer notifies of events through an Observer design pattern.

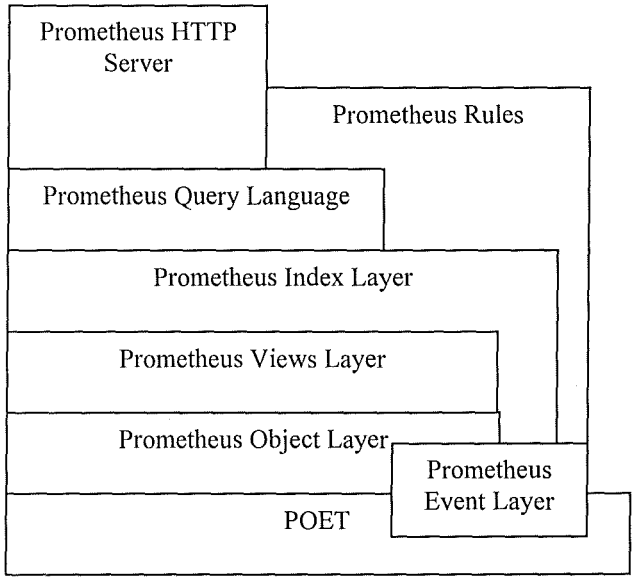


Figure 26: Prometheus architecture

6.1.1 Event layer

The lowest layer of the system above the underlying database system is the Event Layer. This event layer simply collects events from the system (e.g. creation or deletion of objects) and broadcasts these events to the components of the system that have manifested an interest in specific events (this is based on a observer design pattern where objects register the events they monitor to a central object, POETShadow, that then manages broadcasts to the rule and the index layer), as shown in Figure 27.

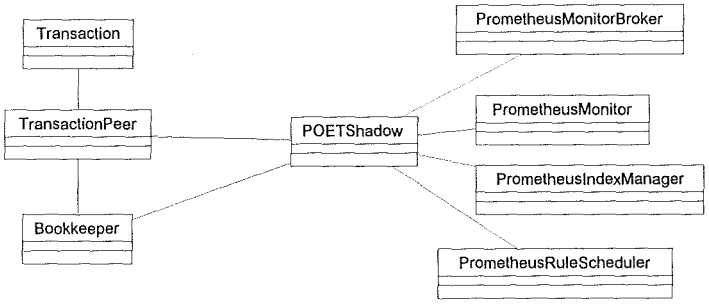


Figure 27: Event layer

The implementation of this layer was not straightforward, as POET, the underlying database chosen for the project, does not implement a full-featured event mechanism in the version available at the time of the project. Therefore, POET was de-compiled and two classes (transaction class and bookkeper class, on the left hand side of Figure 27) replaced by classes that notify events. To do so, Prometheus uses the fact that Java searches classes in the classpath in the order the directories appear. It is therefore possible to write a class that implements the required interface and place it before the original class. When Java tries to find it, it chooses the first it finds.

6.1.2 Object layer

The object layer, that sits on top of the event layer and POET, provides the basic functionality of the objects available in Prometheus: relationships. In this layer, relationship objects are defined, as well as their basic behaviour (e.g. the relationship equality that only compares attributes of relationships, ignoring their source and destination), available constraints as interfaces (e.g. sharing, exclusivity), and mechanisms (deletion propagation of composition). These relationships can be extended by the user in order to capture specific semantics such as the relationships between the matter of which the whole is made and the matter its parts are made of in a part-of relationship (section 4.4.1). Figure 28 shows the Prometheus meta model.

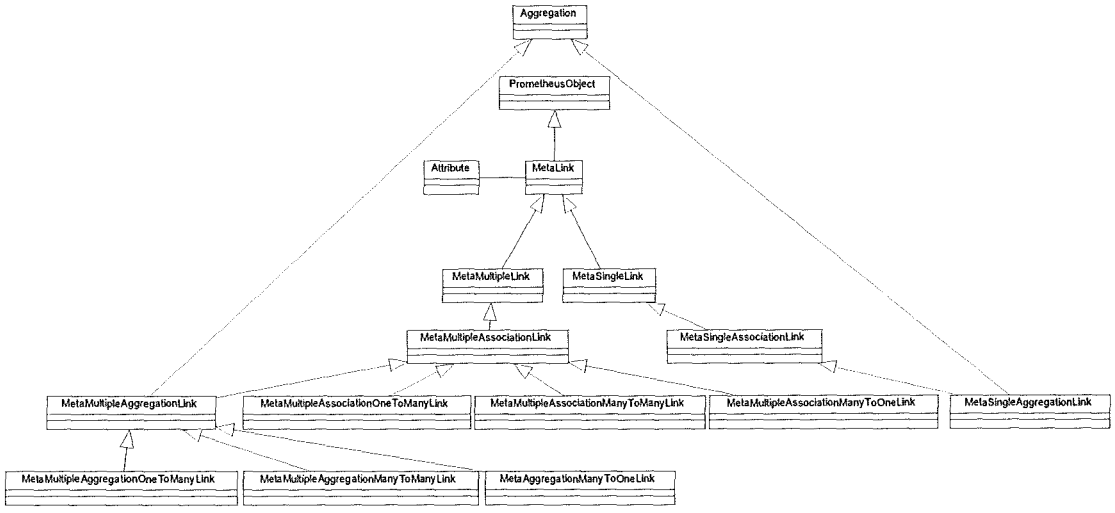


Figure 28: Prometheus object meta model

Opening the database requires performing consistency tests because POET is unable to create and delete a single object in one transaction. This feature is however necessary in the case of temporary objects (e.g. graph reorganisation).

6.1.3 Views layer

Views are filtering views, as for example those of [Zhuge '98]. Views are filters, and unlike for example Multivew, no restructuring is possible. This would not have been justified by the application domain and would not have been practical in Java (restructuring requires a purpose built programming language or a dynamic programming language such as Smalltalk). In addition, views are not duplicates of base objects as in [Zhuge '98], therefore updates in views are propagated to base objects and to all existing views. This can cause inconsistency problems in the context of several view mechanisms (e.g. Multiview), but for Prometheus, changing an object present in a view or adding objects only extends

the view (the filter) via extension. Normally, views are created via a set of queries, i.e. their intension. No view re-computation or maintenance is provided at this stage of the work.

Views can be selected at any time and become a virtual database on which the system works, thereby limiting the amount of data available to the user, making work easier. For this purpose, Extents take into account active views when they present objects to other parts of Prometheus (Figure 29). If a view is active, either as a default view or via user selection, an extent will hide objects that do not belong to that view. In order to maintain consistency, when a view is modified, all extents that have manifested an interest in the view are notified and they must be recomputed.

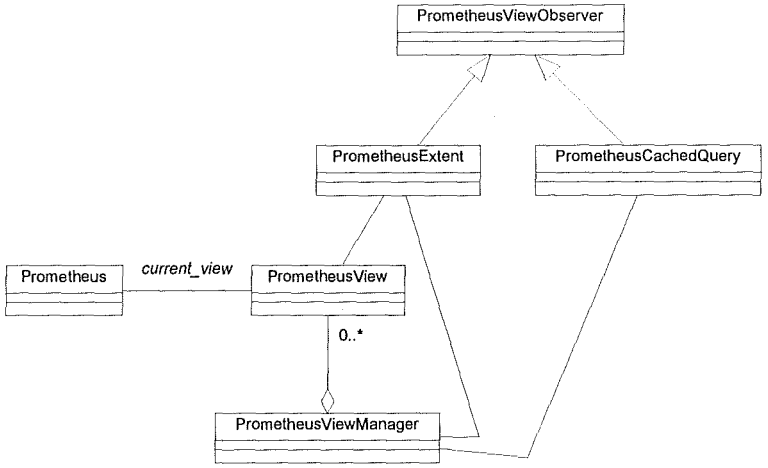


Figure 29: View layer

6.1.4 Index layer

The index layer has been implemented above the view layer because it uses the filters offered by the view layer to restrict available data and the search space for indexes. This configuration may seem unusual, as generally views are a high level feature that use indexes instead of the reverse. But as Prometheus' views are limited to filtering views, they do not need high level services, therefore can be pushed down into the lower implementation layers. In the current implementation of Prometheus, two kinds of indexes exist: relationship indexes, that associate relationships with node objects (it is possible to search on origin or destination objects), and object indexes that index node objects according to their attributes. These indexes are updated when the user performs actions on objects present in the database via the event layer.

Indexes are maintained through events. When an event is fired, e.g. object creation, the index manager intercepts the event and decides whether to update the indexes or not. If this is necessary, the information found in the event object and in the entity that fired the event are sufficient to update indexes. This approach has the significant advantage of being completely transparent to the user. Indeed, no additional code is necessary in user applications to maintain indexes and no changes to

existing applications are required. The disadvantage of this approach is that events are necessary to manage indexes, therefore the overhead generated by event detection and propagation cannot be avoided.

6.1.5 Query layer

The query language developed for Prometheus is based on OQL. It implements a subset of the standard (mostly select queries) plus many features specific to classifications and object-oriented modelling. In the current implementation, it is an independent interpreter, i.e. it does not use the built-in query language provided by the underlying database system. The reasons for this choice are:

The OQL implementation offered by POET is very limited and most queries would require heavy rewrite and many features would have to be implemented in the interpreter. For this reason, a gain in performance was unlikely.

The OQL implementation provided by POET only queries on disk indexes, i.e. it is impossible to query on objects that have not been already committed to the database. This is a serious limitation for an active system. Using that query engine would have meant verifying the results of the query with the information available in memory (e.g. objects in different states), which would have been extremely difficult and inefficient.

These problems are specific to POET, therefore it is conceivable that in a future implementation Prometheus may use the underlying query language to evaluate queries more efficiently.

Several features have been implemented to support more efficient query evaluation: specific execution modes, indexes, and optimisations.

6.1.5.1 Execution

Queries can be executed in different modes that can be selected by the user: sequentially (all selections are made independently of others) or concurrent (selections are grouped to reduce the search space). They can also be indexed by relating path expression and result set in order to make the later execution of the same rule faster. And they can be optimised in various ways. Note however that Prometheus does not support a full algebra that would allow more optimisation or domain driven rewrites.

Prometheus proposes two modes of execution: sequential and parallel. These modes are selected by the user or the user application and apply to all queries executed during the period of time the selection is active.

The sequential mode takes each operation described in the query (e.g. the selections, the join, and the projection) and executes them sequentially. For example, a query that describes two selections and a join will first be interpreted as a sequence of two selections, then a join. However this execution strategy does not benefit from the ability of certain languages and environment to run parallel tasks. Another mode, parallel, has therefore been implemented.

In the concurrent mode, the selections can be evaluated in parallel, i.e. each selection is run independently from the others in a different context (thread in Java). The joins and the projection however are executed sequentially, as they need the results from the previous phases to work (e.g. a join needs the results of selection before it can be evaluated). This execution strategy has the advantage using the resources of the system in a more efficient way (e.g. by using I/O time when the processor is free to execute code), but its drawback is that the selections must be independent, therefore they do not allow the use of certain optimisations (section 6.1.5.3).

6.1.5.2 Indexing

When a query is run, it can be indexed by the system. Indexing means that the selection parts of a query can be stored along with the results of their evaluation and can be later recalled when the same query is executed again. This mechanism makes sense as the same or similar queries are often run many times.

This type of index makes sense particularly in the context of a system that supports rules, as the same rules will be executed often (e.g. cardinality of relationships). See section 5.2 for more information about rules. The improvement provided by this indexing is dramatic, as the execution of a query two or more times takes little time over the execution of the query once (only the time to search a simple index is required). In addition, because only the selection part of the query is indexed, similar queries benefit from the indexing of each other's selections.

Indexing also implies checking and invalidation, i.e. if some objects are changed in the system, they might invalidate the indexed query, and the re-evaluation of the query would be different after their change. The system therefore records in the index the classes of objects that have been traversed during the execution of the query. The kinds of entities kept in the indexes is limited to classes as indexing specific objects would not guarantee that their monitoring would be sufficient. For example, the addition of a new object in the database, therefore an object that cannot be part of any index, might affect the result of the execution of a query that looks for objects of that type having certain properties.

6.1.5.3 Optimisations

The current implementation of Prometheus offers only one very simple execution optimisation but others can be implemented: when two selections are related in a specific way, the system can combine the execution of the two selections in order to restrict the search space. For example, if the user searches for Person objects whose age is 25 and live in Scotland, the two selections are clearly related

and Person objects that is not part of any of the two selections will not appear in the result of the query. In that case, Prometheus executes one selection, then restricts the execution of the other to objects that were returned by the first. Therefore, in this example, only Person objects whose age is over 25 will be candidates for the selection of Person objects living in Scotland.

This optimisation mechanism is not compatible with the concurrent execution of selections, as it implies that one selection is executed once the result of the other is known. Therefore when this optimisation is selected, the execution of queries always becomes sequential. It is not fully compatible with the indexing of selections, as only the selections that are evaluated independently from others can be indexed and reused in the evaluation of other queries. Otherwise, only the first selection is indexed.

Other optimisations are left as future work.

6.1.6 Rules layer

Rules also make use of the event layer. As events are triggered by actions in the database, they are propagated to the rules layer. When the rules layer receives an event, it searches for all rule managers that monitor this event. A rule manager is an object that is associated to persistent objects and that stores and manages their rules (Figure 30). If a rule monitor monitors the triggered event, it fires the execution of the appropriate rule by sending it to the rule scheduler. Depending on the type of the rule, the rule scheduler either evaluates the rule as soon as it is received from a rule manager, or stores it in a queue for later evaluation (e.g. deferred rules that must be evaluated at the end of a transaction). The rule scheduler also receives events from the event layer and may take actions if necessary. This is for example the case for transaction events that represent important points in time for the evaluation of rules. In addition, rules can be called explicitly by the user (user triggered rules in section 5.2.1) and sent to the rule scheduler without rule manager.

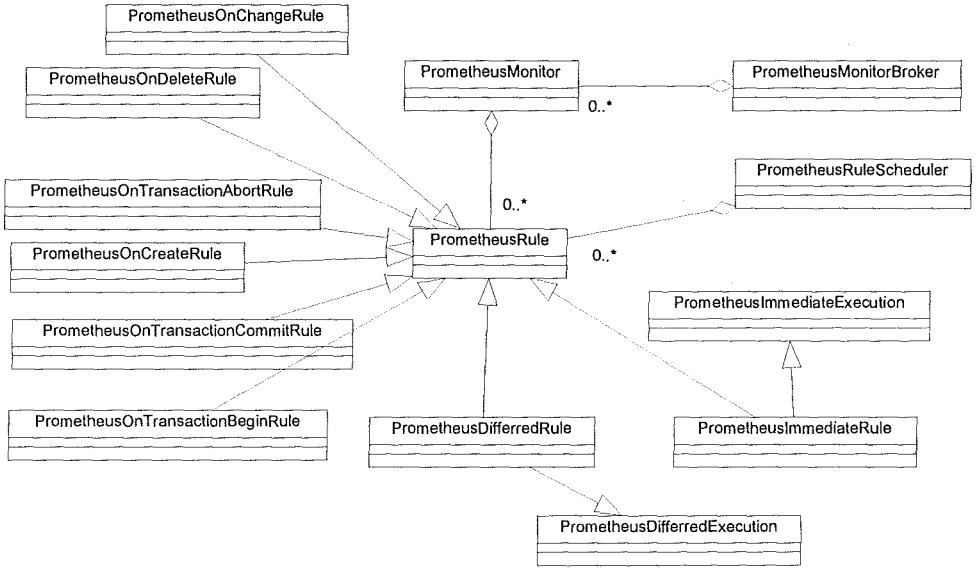


Figure 30: Rule layer

Figure 31 shows a simplified view of the rule execution mechanism. When an object is created/modified/deleted, an event is generated. That event is sent to the rule manager which firstly searches for the active registered rules that react to the notified event, and secondly checks whether the selected rules are attached or monitor the object or the type of the object that triggered the event. If rules that satisfy these constraints are found, they are sent to the scheduler. The scheduler then chooses to execute the condition part of the rule immediately, or stores the rule in a queue. If rules need to be evaluated immediately, the scheduler executes the condition of the rule by calling the POOL interpreter, possibly after having translated the condition from PCL to POOL (see section 5.2.3). If the rule resolves to true, then the action (or constraint) part of the rule must be evaluated. The scheduler then sends the action part of the rule to the POOL interpreter, once again possibly after translating it from PCL. When the event triggered is the end of transaction signal, the scheduler evaluates all the rules that have been stored for differed evaluations.

As described in section 5.2.1.1, rules in Prometheus are attached, either explicitly or implicitly (by analysis of the rule) to classes in order to facilitate event detection. In the current implementation of Prometheus, these entities can only be classes (in most cases these can be inferred automatically). This is motivated by two reasons. The first is that the purpose of the rules is in part the support for relationships and their semantics. Because the constraints associated with relationships are orientated towards the verification of configurations (e.g. sharing of objects, placement in hierarchies), the entities that need to be monitored are usually classes (e.g. relationships in general or a particular kind of relationships). The second motivation is practical: it is hard, when one has no access to the database source code to add the triggering of events dependent on attributes (e.g. change of a particular attribute in an object) or references (the creation of a reference to an object as in Jasmine). As Prometheus should be integrated as smoothly as possible in an existing system, this was not an option.

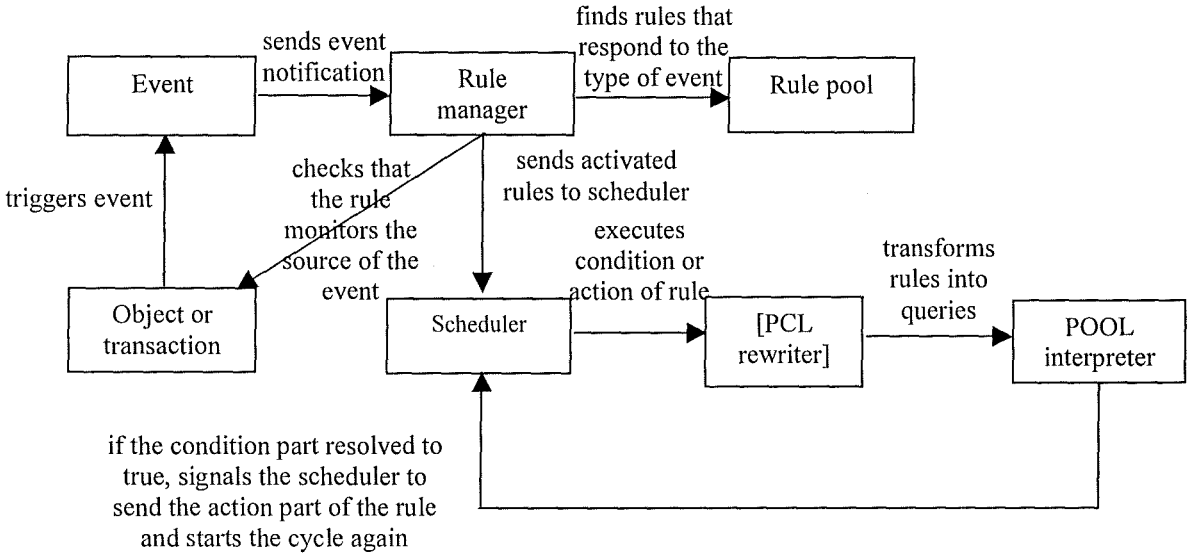


Figure 31: Rule system architecture

6.1.7 HTTP Server

On top of the database, a generic web server is provided with Prometheus. Generic means that it does not only work on top of a taxonomic database. It is designed to be as independent as possible from the underlying database system, but offers an interface to any database system. In addition, it provides means to query and display general classifications. The server has been used as a demo for Prometheus available online to anyone.

In order to make the development easier, the server implements a servlet architecture where new servlets can be added to the system as long as they implement a specific servlet interface.

6.2 Usage

Two kinds of objects are made available to user applications: objects, which are objects in the usual object-oriented sense, and relationships, which are a peculiarity of Prometheus.

6.2.1 Objects

The definition of objects in the context of Prometheus does not require any changes compared to the definition of objects in other database applications. This was an important constraint on the prototype due to the fact that taxonomic databases (and more generally classification databases) already exist and the prototype should reuse that existing data. This also avoids disturbing the inheritance hierarchy that

may be specified at the user application design time to include Prometheus services (especially for single inheritance languages).

However, as Prometheus proposes rules/constraints as a basic feature of the model (unlike ODMG, see section 4.2), rules/constraints must be provided by the prototype. They can be added easily to any object as shown in Figure 32.

```
        // Locally defined rule. The rule specifies that name of rank
        // Species should be placed into a name of rank Genus. The rule
        // is attached to the object "this" (i.e. the one that makes
        // the call), No class to monitor is specified (the choice will
        // be made by Prometheus automatically), and the rule is
        // permanent. The call create an evaluator of PCL rules and
        // send to it the rule in PCL and its arguments, and a rule
        // creator object (for an immediate rule in the example)
theRules.add(new PCL(("self.theRank.name=\"Species\" implies
self.Placement.origin.theRank.name=\"Genus\"", this, "",
false)).translateToRule(new PrometheusImmediateRuleCreator());
```

Figure 32: PCL rule creation

Alternatively, a less user-friendly definition can be given by providing a POOL query instead of a PCL constraint. Figure 33 shows a constraint equivalent to the one presented in Figure 32 but using queries instead of the PCL to POOL translation. The constraint created is permanent (last argument in the call), with a condition that checks that the name is a Species-level name, and ensures that it is placed in a Genus-level name.

```
theRules.add(new PrometheusImmediateRule("The taxon is a Species-
level taxon therefore must be placed into a Genus-level taxon",
"select n from Name n where n = this and n.theRank.name =
\"Species\"", true, " select n from Name n where n = this and
n.Placement.origin.theRank.name = \"Genus\"", true, "", false));
```

Figure 33: Constraint with queries

Note that, as in SAMOS [Gatziu '91], if the rule is not attached to any object, it becomes a general rule and will be evaluated each time an event is fired. This is less efficient than attaching the rule to an object because in that latter case the rule is only evaluated when the object that generated the event is the rule to which it is attached or is an instance of the classes to be monitored.

6.2.2 Relationships

Relationships are defined as sub-classes of one of the predefined relationships provided by Prometheus (section 4.3, Figure 14). These relationships provide the basic behaviours found in object-oriented modelling and implementation, e.g. aggregation relationships with customisable behaviour.

A relationship object is divided into five parts: the definition of the super-classes, the selection of behaviour flags, the declaration of rules/constraints, the definition of attributes, and the definition of methods. Apart from the selection of behaviour flags and the description of rules/constraints, the declaration of relationship classes resembles that of any other class. To make the creation of relationship objects easier, a skeleton is provided (Figure 34). The example shows how to specify a relationship that contains no specific rules (but the rules defined in the MetaLink parent class are inherited), navigable in both directions, not lifetime dependent (weak or strong), changeable, and not private. As was said in section 4.4, other semantics are available, but must be implemented as constraints. These constraints, exclusivity, sharability, cardinality, encapsulation are available as templates, but must be customised to suit the user application. For example, exclusivity must refer to specific classes (the other classes with which the relationship is exclusive), therefore cannot be generically provided by the prototype.

```
public abstract class Relationship extends MetaLink {
	public static String getName() {
		return "Prometheus.Model.Relationship";
	}

	public void setRules() {
		super.setRules();
		// Additional rules/constraints can be added here
	}

	// Used for loading objects. Do not call!
	protected Relationship () {
	}

	protected Relationship (Object anOrigin, Object aDestination) {
		super(anOrigin, aDestination);

		// These are inherited from MetaLink and select behaviour
		navigableFromDestination = true;
		navigableFromOrigin = true;
		lifeTimeDependent = false;
		strongLifeTimeDependent = false;
```

```
            changeable = true;
            private = false;
        }

        public static Collection getOriginSuchAs(Object anObject) {
            return getMetaOriginSuchAs(anObject, getName());
        }

        public static Collection getDestinationSuchAs(Object anObject)
{
            return getMetaDestinationSuchAs(anObject, getName());
        }
}
```

Figure 34: Relationship template

The super-class of all relationship objects, called MetaLink, also provides mechanisms that manage some of the behaviours specified by the behaviour flags. For example, dependency requires that when an object playing the role of a whole is deleted, its parts are also deleted. Similarly, when parts involved in a strong dependency relationship are deleted (section 4.4.3), the object acting as the whole must be deleted. This behaviour is selected by the user and managed automatically by the MetaLink objects.

6.3 Conclusion

Prometheus has been designed to be integrated with existing database models already containing data (requirement 10 in section 2.4). Therefore not only its model was designed to integrate smoothly with the object-oriented database standard model (ODMG, section 4.3), but its prototype implementation has been written to transparently provide services to user applications without requiring rewriting of existing code or reorganisation of existing schemas, i.e. it is non-intrusive.

This goal led to the development of a structure that leaves the management of most of the new features to the new structures provided by Prometheus: relationships. Relationships specify and manage the behaviours presented in section 4.4, freeing the user from this burden and avoiding the involvement of user objects is such behaviours, which would have led to a harder to maintain approach.

The customisation of the semantics given to relationships, necessary to support fully user applications and modelling methods, is done through the selection of predefined flags and the declaration of constraints (whose function is explained in section 4.4) during the definition of relationship classes. If

one of the predefined relationships is sub-classed (e.g. single-valued aggregation), the semantics of the user relationships will be inherited from the predefined parent class.

For the first time, classification applications in general and taxonomic applications in particular can benefit from a system providing a means to represent multiple overlapping independent classifications, and manage them in user applications and in the query language. As chapter 7 will show, the improvements offered by Prometheus support fully taxonomists' needs and their absence would make the development of such applications harder and more costly.

7 Evaluation

This section evaluates the performance of the system built during this work. Two aspects are considered: taxonomic performance, i.e. how well the system answers the taxonomic problem, and database performance. The evaluation of the way the system handles the taxonomic problem is very important, as this is the first prototype that properly tackles the multiple classification problem. This evaluation should show whether the chosen approach was suitable. Performance has not been an issue during the development of the prototype, but it is important to know how it behaves and why. This would form the basis for an extension of the prototype and its refinement.

7.1 Taxonomic evaluation

It is necessary to study the taxonomic merits of the system built. This evaluation is made against a strict benchmark according to a protocol that is defined here. Typical problems generated by taxonomist work are considered and the solutions proposed by the system are explained. The complexity of this solution (both from a user and a system point of view) is evaluated. As there is no accepted benchmark for taxonomic work, a revision has been undertaken and completed with Prometheus to measure its success. This protocol includes 4 sections: the ability of the system to represent multiple classifications and their relationships, the ability to represent historical classifications, the ability to support taxonomic working practices, and the what-if scenarios. All these features are necessary in a taxonomic revision.

7.1.1 Support for multiple classifications

As was presented in section 2.4, one of the most important requirements of a taxonomic system that lets taxonomists work freely and accurately is the ability to represent multiple concurrent classifications without forcing the user to choose one as the "accepted" one.

Multiple classifications in Prometheus are represented as trees (but Prometheus itself is not limited to trees and could accommodate graphs) where the nodes represent the information that is classified and the arcs (directed edges) represent the classification and the classification information.

As the arcs are independent from the nodes (they are not part of the nodes' type, as pointers would be), the classifications can be created independently from the objects that are classified. It could even be possible to reuse an existing taxonomic database system that does not handle multiple classifications and introduce classification arcs from the Prometheus package in order to extend the system (integration requirement from section 2.4). This allows the representation of multiple classifications, as nodes can be shared between an arbitrary number of classifications by the addition of new classification relationships pointing to or pointing from specific nodes. This also allows the

representation of classifications where the classifying objects (taxa in taxonomy) are independent objects that are important in themselves (their information can be published) and are not designed specifically to be classified or classify.

In fact, Prometheus allows the generic classification of entities by context. By "context", one can understand "anything that uniquely identifies a view". In plant taxonomy, this can be a taxonomist, a publication, or a combination of both. For example, one taxonomist's view on the world is a context and in that context a set of specimens is classified in a certain way. Concurrently, another taxonomist's view of the world represents another context where the same specimens (or any other set of specimens) are classified differently. The overall graph that is stored in the database represents a view of taxonomy out of context, or within all contexts concurrently. This view, although it is the most complete because it contains all existing information, does not suit taxonomic work, as taxonomists tend to work in one particular context or in relation to a limited set of contexts at once for comparison purposes (see section 2). By representing classification information on the hierarchies that constitute that graph, Prometheus captures single contexts that can be extracted as necessary (see section 5.1.1.3). Because the distinct hierarchies created in different contexts overlap (in terms of specimens and names), the representation of all contexts in a single graph makes possible the comparison of classifications defined in different contexts and provides the ability to switch between contexts in order to gain knowledge.

Prometheus therefore inherently supports multiple classifications and their complexity and overlap can be arbitrary.

7.1.2 Historical classifications

Historical classifications are classifications published in the past, possibly when the rules of taxonomy were different. Although the system is related to the taxonomic model presented in section 2.3 and uses it as its motivation, it is not directly dependent on it. For example, the taxonomic model stresses that the only piece of information that is objectively present in classifications is the set of specimens put together in the various taxa that compose the classifications. Other information, for example the taxa, is not reliable because these are never clearly described (as anything but a set of specimens [Pullan '00]). This taxonomic model is therefore only able to represent historical classification if either specimen and type information is available, or if a later taxonomist elects new types (lectotypes) in order to make the classification compliant with the more recent working practices [Greuter '94] and the new model of taxonomy.

However, some taxonomist groups insist that this approach is not practical, as this implies the handling of more information than taxonomists are used to. It implies having access to taxonomic type description, and generally it requires that when publications have been published, all appropriate information relating to specimens has been published. As an experiment, taxonomists have used this system to create a revision of the Globba group. This has shown that although it was necessary to

record more information than taxonomists are used to, the amount was still manageable [Newman '00]. However, it is true that the necessity to have access to all specimens and type descriptions related to a particular classification is not always practical. In particular, as the rules of taxonomic practices have changed over time, this information was not published before 1753. Even in later published classifications, the whole set of specimens (non-typical specimens) was not always published. Therefore the representation of historical classifications is problematic in the taxonomic model of section 2.3.

Some taxonomists argue that although not always true, it is often possible to have a reasonably good idea of what a taxonomist intended in a specific classification without access to (unavailable) specimen information. It is especially true if type specimens are emphasised and are said to represent the whole group of specimens placed into a taxon, irrespective of its circumscription. Therefore, they argue, it is possible to adopt a taxonomist's definition of taxa, without objective specimen information, and reuse it in a revision of the group or of a parent of the group. This approach is less reliable than the purely specimen approach proposed in [Pullan '00], but is thought to be a reasonable and practical approach by some taxonomists.

As the work presented here is independent from, although used for, the taxonomic model proposed in [Pullan '00], it is able to represent historical classifications in a more efficient way. Historical classifications can be represented as current classifications when type information, and possibly the whole specimen information, is available to the taxonomist. In this case, taxa are defined as sets of specimens and hierarchies of taxa form classifications. These classifications are as objective as possible and reliable.

When specimen information is not available, the database model allows the representation of classifications based purely on taxa or type specimens. Because the system allows the creation of overlapping classifications, it provides the means to share data between these classifications. This data can be taxa that have not been described by the specimens they contain, but only by the other taxa they contain. Because the system is able to classify entities with mechanisms that are independent from them (relationships), existing taxa can be reused in new classifications if the taxonomist is confident about his or her understanding of other taxonomists' concepts.

The system lets the user choose names (ascribed names) and places these names directly into classifications. When a taxonomist desires to reuse the definition of another taxon, the system can simply create new classification relationships that lead to that taxon with new classification information (e.g. author and publication). This allows the description of multiple overlapping classifications without specimen representation.

In order to make decisions clearer in the situation when specimens are not represented, mechanisms such as those presented in the HICLAS system (see section 2.2) can be implemented in a

straightforward manner. These mechanisms would allow the user to describe the actions that lead to the creation of new groups when specimen information is not available (therefore these actions cannot be inferred automatically). For example, the 8 operations defined by HICLAS to represent the life cycle of taxa (e.g. splitting of a taxon, demotion or promotion) could be represented as new relationships and they could be explicitly created by the user in order to explain his or her actions to other taxonomists (e.g. "this taxon has been divided into these two taxa"). HICLAS allows only the creation of such relationships, without any explanation from the user. However, it is sometimes important to explain one's actions in order to make things clearer to others. For example, the reason for splitting a taxon may be important and could be a consequence of a particular insight in the relationship between specimens. Because the model also supports the definition of attributes on relationships, these relationships could contain explanations in addition to simply the description of the action.

This approach to modelling taxonomy was not possible before Prometheus. Indeed, all existing taxonomic databases take one single view of taxonomy and cannot be applied to an alternative view (e.g. Pandora could not be modified to support a specimen based view of taxonomy as it was designed to handle taxon-based classifications only). Prometheus on the other hand can support all ways of working without modifications.

7.1.3 Support for working practices

The support for working practices involves three aspects of the database: the query mechanism that allows the taxonomists to explore and retrieve information from the database; the rule/constraint mechanism that allows the system to enforce the ICBN; and the ability of the system to work in different contexts (distinct classifications) as well as in the out of context view of taxonomy (all classifications and their relationships). This section therefore discusses these three aspects.

7.1.3.1 Typical taxonomic queries

Prometheus supports typical taxonomic queries through the features added to OQL in POOL. The addition of these mechanisms allows answering queries that would have been impossible with OQL only.

For example, using the schema presented in Figure 20, Prometheus allows the retrieval of whole classifications. What is the classification created by Linnaeus and published in "Flora Europea"?
In OQL:

```
(Q1)  Select t.origin, t, t.destination from theCircumscription t
      where t.theCircPublication.thePublication = "Flora Europea" and
      t.theCircAuthor.surname = "Linnaeus"
```

This query extracts all relationships and all nodes involved in the definition of Linnaeus' classification published in "Flora Europea". The selection is performed on the basis of relationships and their

attributes (a specific context is selected). The resulting collection contains these objects arranged in triples representing atomic portions (two nodes and an arc) of the classification.

Most importantly, Prometheus makes it possible to compare two plant groups in one query, which, unlike specifically written code and canned queries, allows the dynamic construction of queries through a user interface. For example: what specimens appear in the circumscription of Apium according to Linnaeus and also in the circumscription of Heliosciadium according to Watson?
Note that the two groups selected do not contain specimens directly, but rather contain other groups that may contain specimens (e.g. Species) or that may contain other groups that contain specimens (e.g. Sections).

(Q2)
```
Select s from Specimen s where s in (select
t1.destination[Specimen] from theCircumscription t1 where
t1.theCircAuthor.surname = "Linnaeus" and
t1.(theCircumscription.origin)*.theEpithet.theName = "Apium" in
context) and s in (select t2.destination[Specimen] from
theCircumscription t2 where t2.theCircAuthor.surname = "Watson"
and t2.(theCircumscription.origin)*.theEpithet.theName =
"Heliosciadium" in context)
```

The `in context` operator is used here in order to keep the traversal of the graph of objects in the same classification as the first traversed relationship. Each of the sub-queries selects a relationship used for building classifications according to its attributes (here author name), the `in context` clause then ensures that only similar relationships are considered during evaluation, and all specimens targeted by such relationships are extracted. Note also the use of the selective downcast operator to restrict the type of selected objects to Specimen instead of Circumscription, as the typing of the schema would force.

A more complete evaluation of typical queries can be found in appendix II.1.

7.1.3.2 Constraints and the ICBN

The implementation of the ICBN is an essential feature of a taxonomic database. This is also an essential feature of many other domains that require that the data entered in the database be correct. Section 5.2 showed what rules Prometheus offers. These rules can be attached to any object in the system. They can implement the rules of the ICBN as this section shows. In order to prove the ability of Prometheus to implement the ICBN, several typical rules covering a large range of different rules is used. These rules are divided into two categories: object rules and relationship rules. As will be shown, these two categories are very different and are the consequences of different points of view.

7.1.3.2.1 Object rules

Object rules are rules that are encountered in most active systems. They are attached or refer to normal objects in the database and ensure their validity or enforce specific behaviour relative to these objects. One typical such rule in the ICBN is a rule that governs the way names can be created. For example, according to their rank, names must have different endings: a Family name must end with "-eae", a Tribe name must end with "-ae", and a genus name cannot end with "-virus". These rules can be enforced by creating a rule similar to the one shown in Figure 35. This rule states that if the rank of a name is Family, then the name must end with "-eae".

```
Self.theRank.theName = "Family" implies Self.theEpithet.name like "%eae"
```

Figure 35: Family name rule

Similarly, a name at rank Genus could be monitored by a rule such as shown Figure 36.

```
Self.theRank.theName = "Genus" implies Self.theEpithet.name not like "%virus"
```

Figure 36: Genus name rule

To these rules will be associated a set of class names whose instances will trigger the re-evaluation of the rule. The system can automatically derive these class names and they would be this (the object that contains the rule), theRank (in case the rank relationship is changed), and Rank (in case the rank hierarchy is changed). In addition, because they can never be violated, they would be created as immediate effect and non-interactive rules (see section 5.2). They can either be implemented as pre-conditions in a method if the only way to change the value of the name is via that method, or as an invariant.

Another kind of rule is rules that allow user intervention. As was said earlier, taxonomists have a great degree of liberty in their work. In addition, the existence of historical classifications that followed different rules and the presence of mistakes sometimes make the enforcement of the ICBN impossible. For example, since 1753, all names to be published must be published with at least a type. But before that date, there was no obligation of doing so, and many classifications and publications of names do not contain any type. However, the current botanical code insists that all names must have a type, therefore the rule shown in Figure 37 can be devised. That rule makes sure that when a name exists, a LinkToType relationship (or one of its subclasses) starting from it must exist. This rule would be created as deferred (it only needs to be enforced at the end of the transaction as the type cannot be created exactly at the same time as the name).

```
Exists(Self.LinkToType)
```

Figure 37: Type existence rule

However, since old publications or mistakes are made, some names may exist without a type. Although in this case the taxonomist that uses the name should elect a lectotype (see section 2.1), a name without

type may exist in the database for a period of time if it not used by any new classification. Taking into account the date of publication of the name would not necessarily solve the problem as not only historical but also wrongly published new classifications should be handled. Therefore the rule should be created as an interactive rule, in which case when the system realises that it has been violated, an interactive session with the user is set up to ensure that he or she understands that this violates the ICBN and should not occur for new names correctly published, but that it is acceptable for historical publications or wrongly published names.

As was explained in section 2.1, taxonomists can choose to use an arbitrary hierarchy of ranks by ignoring some of them or by deciding to work only between two specific ranks. Therefore the representation of the rank hierarchy would not ensure that the order of the ranks is not violated. This order can be implemented as rules on Rank objects. For example, the ICBN states that the Species rank should always appear below a rank that is described as Genus or one of its sub-ranks until Species excluded (Species cannot be placed into Species). Thus rule is shown in Figure 38.

```
Self.name = "Species" implies Self.previous+.name = "Genus"
```

Figure 38: Species rank rule

This rule only ensures that Species is always placed below Genus, but does not enforce the rank hierarchy between Species and Genus. The definition of similar rules for other ranks ensures that, for example, the Series rank must always be placed below a Section rank. Figure 39 shows such a constraint. All the rules necessary to maintain a valid rank hierarchy can be enumerated in that way.

```
Self.name = "Series" implies Self.previous+.name = "Section"
```

Figure 39: Series rank rule

In the example in Figure 39, the regular operator + is used because the rank Series is not necessarily directly below a rank Section. The rank Subsection can be inserted between them.

7.1.3.2.2 Relationship rules

Other types of rules exist in Prometheus. Instead of being attached to objects, they are attached to relationships (see section 5.2.1.4.4).

Such a rule is used to enforce the placement of names according to the hierarchy of ranks. For example, a name at rank]Genus, Species] should always and necessarily be placed below another name of rank [Genus, Species[. Such a rule would clearly be immediate effect (it can never be violated) and without user interaction, and would be implemented as shown in Figure 40. This rule, attached to the Placement relationship class, would be evaluated each time a Placement instance is created or modified, as well as each time a theRank, and a Rank object are created or changed.

Self.origin.theRank.previous+.name = "Genus" and Self.origin.theRank.next*.name = "Species" implies Self.destination.theRank.previous*.name = "Genus" and Self.destination.theRank.next+.name = "Species"

Figure 40: Placement rule

This rule can be specialised in order to be more specific. For example, each rank may be taken into account and the possible combinations of that name with other names can be expressed individually by distinct rules. This approach has not be chosen in Prometheus as first this requires that the rank hierarchy is known and never changes (and section 2.1 showed that taxonomists have the liberty to use any ranks they want in certain limits and even create theirs) and second this approach would be extremely expensive in terms of the number of rules to be evaluated each time certain events occur in the system (this number could be as large as 276 given the rank hierarchy shown in Figure 1).

Similar rules can be described for the theCircumscription relationship that also requires that groups at certain ranks be placed into other groups at specific ranks. As the theCircumscription hierarchy does not necessarily follow the Placement hierarchy (e.g. a Species name can be the type of a Genus name, but the classification may use additional ranks between Species and Genus such as Series or Section), theCircumscription relationships need their own rules. These rules however are the same as the Placement rules.

7.1.3.3 Querying by context

As was said in section 5.1.1.3 it is sometimes necessary to work within a specific context. This context can be selected automatically by the system in some cases. For example when a classification is encountered, the traversal of graphs or hierarchies should always remain in that classification and never switch to others. It is also sometimes necessary to specify explicitly a context of interest in the queries. This is done using the extended `in context` operator (see section 5.1.1.3).

The `in context` operator can be used in order to express recursive joins, i.e. automatic selection of some relationships via a join in recursive expressions. The `in context` operator can also be used in order to select a particular context, e.g. a classification publication or the author of a classification, and perform all the queries in that context. For example (Q3) shows how all the names in a particular classification that contain directly or indirectly a specific specimen (identified by X) can be extracted:

```
(Q3)  Select * from Name n, theCircumscription t where
      n.(t.destination[Name])*.t.destination[Specimen] = "X" in
      context where t.theCircAuthor.surname = "Linnaeus"
```

In essence, this query contains two different selections: one that selects Name objects (the main selection) and one that selects the relationships of type theCircumscription that must be followed.

This notation has the advantage of acting as a filter on the main part of the query. Indeed, the query written without the `in context` selection would return all the names of all the classifications that contain the specimen identified by X. By adding the `in context` selection, only one or several well defined classifications are considered. It also has the advantage of being able to be carried from query to query if the system works in specific contexts.

7.1.4 What-if scenarios

As Prometheus is the only taxonomic database that represents specimens and implements the rules of taxonomy (the ICBN), it is the only system that supports what-if scenarios. What-if scenarios are tests a taxonomist can make in order to check the consequences of his or her actions. For example, a taxonomist might choose to move a specimen from one taxon to another because it is not clear where that specimen should go, and desire to know what would be the consequences of this change on the way groups are named. This can also allow a taxonomist to compare existing classifications and discover why many versions exist. Because Prometheus supports the definition of arbitrary classifications, the representation of specimens, and the implementation of the ICBN, it is able to recalculate the repartition of names in new situations, thereby allowing a taxonomist to experiment with the available information in a way that would not be possible otherwise.

When a taxonomist chooses this approach, he or she can copy an existing classification (possibly one of his or her own) and edit several taxa in order to test the consequences of these changes. The necessity of copying an existing classification is motivated by the fact that once a classification has been published, it should never be changed. Therefore taxonomists can only work on ongoing classifications that must be their own. Once the groups have been redefined, Prometheus can rearrange classifications accordingly (by managing classification relationships) and apply the rules of the ICBN on this new configuration of specimens and taxa. When the new names have been calculated, they can be shown to the user that can decide to perform more changes, reverse to the previous configuration, or adopt the new configuration as the definitive version of the classification.

For example in the classification sample given in Figure 3, if a taxonomist was not quite sure where the specimen "C. von Linnaeus #Herb.Cliff. 107 Apium 1 BM" should go, he or she might decide to see what would happen if it was moved to Taxon 2. In that case, the application of the ICBN would imply that Taxon 1 should be named Apium L., as the oldest type specimen it contains would be the newly added specimen, that was used as the type of Apium graveolens L., itself the type of Apium L., and Taxon 2 would become Apium graveolens L., as the newly added specimen is the type of that Species and the Species name was published before any other known group. Before the specimen was moved, Taxon 1 was named Heliosciadium W.D.J.Koch and Taxon 2 was named Heliosciadium repens (Jacq.)Raguenaud. This what-if scenario might help to understand for example why Heliosciadium and its species do not exist in a particular classification whereas they existed in another.

Note however that this approach can only be supported when Prometheus is used with full type specimen information. If it is used in a taxon-based fashion, automatic naming of groups is not available, as the data is insufficient.

7.1.5 Conclusion

Taxonomic evaluation of the prototype shows that Prometheus is able to represent current and new classifications as well as older, less complete classifications. These older classifications can be captured by linking taxa via relationships when specimen information is not available. This also shows that if a view of taxonomy unlike the one chosen for this project is favoured, Prometheus remains a suitable taxonomic database system.

The taxonomic evaluation has also shown that the prototype copes well with the requirements of taxonomic work. Typical queries have been shown to be at least equivalent to OQL queries, and often to be able to answer queries impossible in OQL. These queries are classification specific queries that taxonomists often use to explore or compare alternative classifications. Without them a taxonomic system could not support taxonomic work properly.

The taxonomic evaluation also shows that the rules defined in Prometheus are appropriate to taxonomic work. They allow the implementation of the ICBN to enforce working practices.

Finally, the ability to play what-if scenarios allows taxonomists to discover and explore alternative classifications when it is not clear where specific specimens should be classified. This may in addition provide means to achieve better understanding of existing classifications as it allows the recreation of such classifications by trial and error until the way they were built is understood.

7.2 Database performance evaluation

The performance of the database needs to be addressed. This performance is considered relative to the technology involved in the implementation of the ideas developed in this thesis. For example, Java was chosen as an implementation language in order to increase the portability of the system and one consequence of that choice is a slower system. For the same purpose, the system has been implemented independently of any specific database system, hence losing the benefit from various facilities offered by a specific database. The only restriction for an underlying database system is transparent object persistence and transaction management, which are offered by most commercial and research object-oriented database systems.

Precise performance evaluation is a difficult problem in the case of Prometheus. Because it was built on top of an existing commercial database, its performance is linked to the performance of the

underlying system, i.e. performance(Prometheus) = f(performance(underlying DB)). Furthermore, the mechanisms that are to be tested need to be equivalent. Since Prometheus provides mechanisms that no other database offers (especially commercial databases), some features cannot be tested against known systems (e.g. recursivity, graph traversal, handling of semantic relationships). Therefore, when possible, two possible approaches are tested in Prometheus (e.g. graph traversal) and their results compared.

This section is laid out as follows: first the test protocol is explained and the test results given. Then they are analysed and the causes of overhead or performance relative to the underlying database system are explained.

7.2.1 Performance testing

The testing of the system requires the definition of a test protocol. This protocol is explained and justified here. The results are then discussed and the causes of the various aspects of the results are shown.

Note that the purpose of this performance test is not to show how good or how fast Prometheus is compared to other databases. Performance issues have been ignored for most of the project as modelling was felt to be a much more important aspect of the taxonomic problem. This performance test should be seen as an indicator of the performance overhead generated by the additional features offered by the system, and it provides a basis for the analysis of the internal workings of the database system. As the purpose of this benchmark is therefore the evaluation of the cost of the features offered by Prometheus, the testing of the system is managed relative to the underlying database system. This makes sense as, as was said earlier, the performance of Prometheus is dependent on the performance of the underlying persistent system (e.g. for object loading, reference resolution, persistence mechanism, and transaction management). By testing the new features against the underlying database system, the added features can be evaluated independently from any consideration of transaction management, persistence performance, index performance, object clustering, I/O timing, and other system dependant performance issues. In addition, as other database systems are not available at the time of this benchmark and as the hardware set-up used for this work is different from the one used by the other published benchmarks, their results are not considered here.

This section is divided as follows: first a schema for the tests is defined. This schema is inspired from the OO7 Benchmark schema but restricted to the sections that are relevant to this work. Then the features tested and the algorithms are presented. Several notes and restrictions are then noted. Finally the results of the test are given. A complete description of the schemas used in this benchmark can be found in Appendix II.

7.2.1.1 From OO7

The choice of the features to be tested in this small benchmark is inspired from the OO7 Benchmark [Carey '93] because of its similarity to the type of schema typical of Prometheus applications. However, the purpose of this evaluation is not the complete evaluation of all performance aspects of an OODB. As Prometheus has been developed with a specific kind of application in mind, namely classification applications and classification databases (of which plant taxonomy is a special case), most of the OO7 tests are irrelevant and would only test the underlying database system (e.g. object cache). A set of features has therefore been extracted from the OO7 Benchmark in order to test specifically some of the features that Prometheus provides.

Several sections of the OO7 Benchmark are of particular interest to this work. A subset of the schema describes an Assembly hierarchy where graphs which are constituted of nodes (ComplexAssembly) and leaves (BaseAssembly) can be described. Figure 41 shows a subset of the OO7 schema. As can be seen, it is very similar to the taxonomic schema used as an application example of Prometheus, where ComplexAssembly would be replaced by Name and BaseAssembly by Specimen (see Figure 20). In addition, each of these objects contains references to other objects in the system, as Name and Specimen do in the taxonomic schema. A notable difference is that Assembly parts are not shared between ComplexAssembly parts, as the cardinality of the SubAssemblies relationship is 1 from Assembly to ComplexAssembly, whereas classifications in the taxonomic schema allow overlap, i.e. sharing of nodes between classifications. However, provision is given in the OO7 Benchmark to test shared and unshared components as components can be private or not, but this feature is not tested in the OO7 Benchmark. By using this section of the OO7 Benchmark, it is possible to test the classification mechanism used in this work.

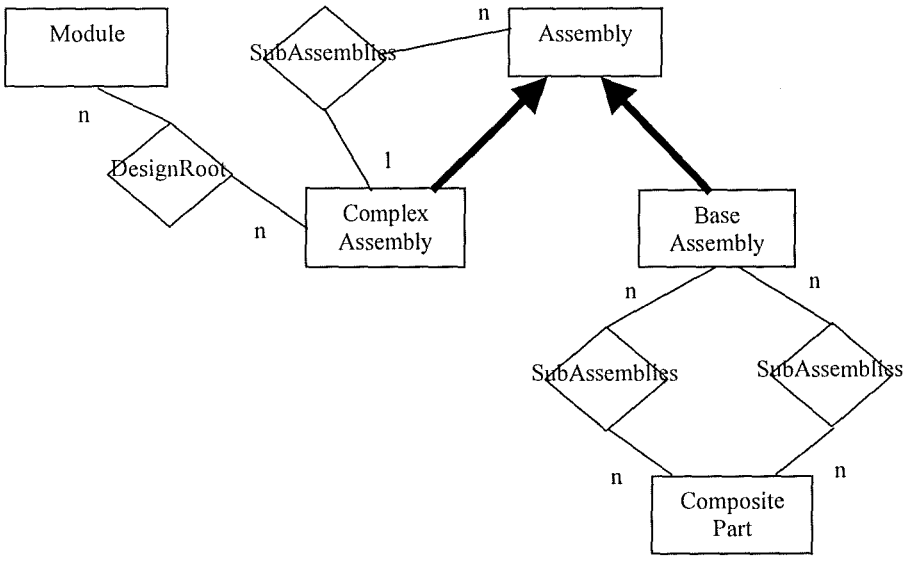


Figure 41: OO7 partial schema 1

Another part of the OO7 schema of interest is the AtomicPart section. An AtomicPart is connected to other AtomicParts via a Connection object that serves as a relationship to which values are attached. Figure 42 shows the OO7 schema. This subset of the OO7 schema looks similar to the way classification relationships are implemented in the taxonomic schema. Indeed, they have attributes that allow the system to distinguish single classifications. However, a noticeable difference is the fact that these attributes are simple atomic values that are embedded in the Connection object type (integers). In the case of taxonomy, these attributes are references (even explicit relationships) to other types. It is therefore expected that access to these Connection objects would be more expensive than simply doing so if these attributes were atomic values. Using this section of the OO7 Benchmark would allow comparing the use of relationships as first-class objects in the system.

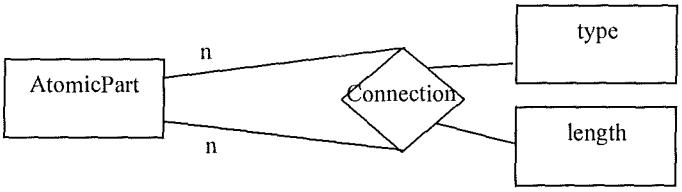


Figure 42: OO7 partial schema 2

These two sections of the OO7 Benchmark have therefore been implemented in POET (Prometheus' underlying database system). In Prometheus, wherever references are used, they have been replaced by explicit relationships without rules (no rule is described in the OO7 Benchmark).

7.2.1.2 What is tested

The following operations are tested against the two databases. They are divided into three groups: program tests, query tests, and structural modifications. Program tests measure the basic performance of the system, e.g. raw object access time. Query tests measure the performance of the query engine. Structural modifications indicate the time necessary for handling operations on objects such as creation and deletion.

7.2.1.2.1 Raw performance

Finding and resolving a specific object

The time necessary to find a specific object is a good indicator of the efficiency of object access in the database. This object access time underlines all other performance tests, as it has to be done many times in queries (e.g. path expression evaluation) or in the life of an application. Two kinds of objects are retrieved in the case of Prometheus: "normal" objects and relationship objects. The distinction is necessary because relationship objects are indexed (see section 6.1.5.2) and subject to more treatments (e.g. constraints and rule triggering) than native objects. Only testing accessing "normal" objects would not produce different results in the two systems, as accessing a "normal" object in Prometheus is simply left to POET.

This object retrieval can be performed at two very different points in time: at the beginning of a clean transaction or inside an existing transaction. The difference is that the database cache may influence the performance of the system. Therefore retrieving objects that already exist in memory will show the difference between cold and hot object retrieval. These two approaches are tested here. In the case of a new clean transaction, it must be ensured that the objects retrieved from the database are not already loaded in memory or in the database cache because they are indexed and a previous access to the index has triggered their loading or resolution. In the second case, it must be ensured that the object to be retrieved is already in memory. A set of objects containing that object is therefore loaded into memory, and the object searched and resolved.

Note that the resolution mechanism is important in this test. Indeed, if accessed objects are entirely resolved, then the size of the object is important in the measurement. Therefore the "normal" objects that are accessed in POET should be sensibly the same size as "normal" and relationship objects in Prometheus. This would avoid distorting results by including I/O timings in the results. Relationship objects in Prometheus contain references to at least two objects (their source and destination) plus a set of boolean values that capture their semantics. Therefore the objects that are accessed during this test in the two systems should have the same structure. In the OO7 Benchmark, no object has exactly this structure. It can be seen in the OO7 schema (Figure 42) that the Connection objects, that serves the purpose of a relationship, contains references to a source and a destination (both AtomicPart objects) and two attributes, type and length. It can easily be extended to contain several more attributes that represent some of the properties of Prometheus relationships as shown in Figure 43.

Retrieve a "normal" object

(T1) `In a clean transaction`

(T2) `In an existing transaction`

Retrieve a relationship object

(T3) `In a clean transaction`

(T4) `In an existing transaction`

Traversing a chain of pointers vs. traversing a chain of relationships

Prometheus offers relationships as first-class objects. This has many advantages, but it may have a negative impact on the overall system performance, as more information must be retrieved in order to traverse a graph of objects (relationships are themselves objects). Therefore one of the tests borrowed from OO7 is the traversal speed test. This test is performed on references (pointers) in POET and on relationship objects in Prometheus. The results of this test will not simply show that traversing relationships is equivalent to traversing "fake" relationship objects (as used in OO7, e.g. the Connection object). Indeed, with relationship objects are associated a set of indexes and caches that hopefully increases relationship traversing performance. This test will therefore not show the raw traversal speed, but, compared with POET results and compared to object access results, it will show

the advantages of using the relationship indexes. OO7 operations T1 and T6 are run to test this traversal time. OO7 T1 tests the raw traversal speed by traversing the Assembly hierarchy: "*as each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return the count of the number of atomic parts visited when done*". The OO7 T2 operation tests the sparse traversal speed: "*Traverse the assembly hierarchy. As each base assembly is visited, visit each of its references unshared composite parts. As each composite part is visited, visit the root atomic part. Return a count of the number of atomic parts visited when done*".

(T5) `As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return the count of the number of atomic parts visited when done`

(T6) `Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, visit the root atomic part. Return a count of the number of atomic parts visited when done`

OO7 also tests the update time of objects. Updates are interesting, because as relationship objects are indexed, the creation or modification of these objects may have consequences on the performance of the system. In addition, objects in Prometheus trigger the evaluation of rules when they are created, modified, or deleted, which may have a performance hit (even when there is no rule to be evaluated). Therefore comparing the update or creation time between "normal" objects in POET and "normal" and relationship objects in Prometheus, this performance overhead can be calculated. OO7 proposes operation OO7 T2 for this purpose: "*Repeat traversal OO7 T1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (x, y) attributes. The three types of updates are:*
A Update one atomic part per composite part.
B Update every atomic part as it is encountered
C Update each atomic part in a composite part four times"
As rule evaluation is triggered when the source or destination objects of relationships are changed, these updates will test the rule triggering mechanism in Prometheus. In order to test the update time of relationship objects, another test must be performed: instead of updating atomic parts, the Connection relationship is updated (to non-changeable for example, see section 4.4.3). Therefore the OO7 T2 operation is extended with:
D Update each Connection relationship as it is encountered

Repeat traversal OO7 T1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (x, y) attributes. The three types of updates are:

(T7) A Update one atomic part per composite part.

(T8) B Update every atomic part as it is encountered

(T9) C Update each atomic part in a composite part four times

(T10) D Update each Connection relationship as it is encountered

7.2.1.2.2 Queries

Retrieving an object via queries

Q1 of OO7 calculates the time necessary to create 10 Part objects with random ids and retrieve them. This query is of interest for the benchmark because this will show the difference in performance between POET's query engine and Prometheus' on very simple queries. These two engines are radically different in design and features, and this may have an effect on their performance, and these differences should appear on these simple queries.

(Q1) create 10 Part objects with random ids and retrieve them

Note that the creation of objects is not possible in POET's OQL nor Prometheus' OQL. These two languages focus on selection queries and do not support update queries. Therefore the creation part will be written as a program whereas the selection part of the query will be written as a query.

Q1 is run in three ways: cold, warm, and hot. The cold run requires running the retrieval of the created objects just after the database has been opened and in a new transaction. The warm run requires a transaction where many objects have been loaded, some of which are useful for the query. The hot run implies running the retrieval twice in a row and only measuring the time necessary to run the second time (the time necessary for the first run is tested by the cold run).

Scan of a collection

Q2, Q3, and Q7 of OO7 test the ability to scan collections and perform aggregate functions on these collections (e.g. average, percentile). These queries are not possible as they are defined in OO7 in POET and Prometheus because these two implementations of OQL do not support aggregate functions. Only OO7 Q7 can be executed as it is defined (scan of an entire collection). However, the testing of scan speed is interesting, as, combined with the retrieval of objects, it shows how collections and indexes are handled in the two systems. Indeed, looking for a particular object with a given attribute value can be performed in two ways: by scanning the entire collection (extent) to which the object is likely to belong, or searching for the attribute value in an index. Combining OO7 Q1 and OO7 Q7 will allow measuring by difference the time spent to load the object to be found and will show how object searching is implemented.

(Q2) Scan the extent of the class AtomicParts

Recursive exploration of the graph

It was argued in several sections that the recursive exploration of a graph of objects is central to the working of taxonomy. For example, finding specimens that occur at the lowest levels of a classification is necessary in order to derive names for the various groups that constitute the classification, and to compare these groups. Operation T5 can therefore also be executed as a query in Prometheus using a recursive exploration the object graph in the projection clause. This is therefore tested, even though it cannot be compared to POET's OQL. The results however, can be compared to the execution time of hand-written programs performing the same task.

(Q3) As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return the count of the number of atomic parts visited when done

7.2.1.2.3 Structural modifications

OO7 proposes to test two structural modifications: the insertion of a new object and the deletion of an existing object.

Firstly, it is possible to test object insertion. In Prometheus, "normal" objects and relationships are created differently. Relationship objects are indexed objects, therefore creating one requires the insertion in memory and persistent indexes, and firing of rule events. Therefore, by running the OO7 operation that consists of creating several new objects and their parts, it is possible to compare the time necessary to create "normal" objects and indexed objects (relationships) in a real world situation.

(S1) Create five new composite parts, which includes creating a number of new atomic parts and insert them into the database by installing references to these composite parts into 10 randomly chosen base assembly objects.

Secondly, OO7 tests the time necessary to delete the created objects (including their parts). This can be done in two ways in Prometheus: by manually traversing the graph of objects and deleting them one by one; it can also be done by selecting the lifetime dependency behaviour on the relationships so that the system takes care of the deletion of parts. The first approach is chosen because it is common to both systems. It is unlikely that the other approach would generate different timings, as the algorithm would be very similar.

These deletions in the case of Prometheus include two semantically different deletions: the deletion of "normal" non-indexed objects, and the deletion of relationships that are indexed. The ratio of "normal" to indexed deletions in Prometheus allows the evaluation of the time necessary to delete indexed relationships. Deleting a "normal" object in Prometheus takes the same time as in POET, as this is in both cases a POET action. The time difference between the two systems therefore indicates the overhead of deleting relationship objects.

(S2) `Delete the five newly created composite parts (and all of their associated atomic parts objects).`

7.2.1.3 Notes

Cold vs. hot

The OO7 Benchmark makes a clear distinction between "cold" and "hot" runs. A cold run is the execution of an operation or query with all caches emptied so that all accessed objects are resolved as necessary from the database during the running of the operations. OO7 tests these two aspects of the runs separately in order to test both object access performance and caching performance. In the case of Prometheus however, this distinction is not easy to make for two reasons. Firstly, the database cache is database-wide in POET (it is created when the database is opened and discarded when the database is closed) and is not accessible from Prometheus. This means that much care can be taken to run the tests before the cache is used, but no control over its behaviour (e.g. disabling) is possible. Secondly, Prometheus uses indexes to speed up relationship access. This index is loaded at start-up and is used to perform a series of consistency tests when the database is opened. This means that normally, as soon as Prometheus is started, many objects are likely to reside in the memory cache. This option can be disabled and therefore more control over the content of the cache is possible. It has been switched off for the benchmark but it is not disabled in normal operations, as consistency checking is important.

Type mismatch

As Prometheus and POET use a Java interface, the types of the objects defined in the OO7 Benchmark have been adapted to fit Java's types. For example, int4 has been replaced by int that takes up 4 bytes in Java.

Likewise, some types are different in the C++-like OO7 design and in Java. For example, char in C++ is represented by one byte (as the ASCII code is 7 or 8 bits based). In Java, a char is represented by 2 bytes as the encoding mechanism used is UNICODE.

These changes may affect the results and make any comparison with the actual results of a standard OO7 benchmark impossible or at least questionable. However, this is not an impediment for the benchmark described here, as the focus is on the comparison between POET and Prometheus, both using the same type system.

Semantic differences

The OO7 Benchmark uses different kinds of collections in order to maintain sets of objects and multiple references (and inverse references). These collections are sets and bags. In the POET implementation of the benchmark, the same semantics have been used. However, as Prometheus offers relationships as first-class constructs, multiple references and inverse references do not need to be handled by the user. Relationships with the correct cardinality constraints (see section 4.4.4) can be implemented and the cardinality can be managed by the system automatically. The limitation of this approach is that these relationships are the equivalent of a bag of objects (duplicates allowed, no order), therefore some semantics may be lost if they are not managed by the application.

Active system

Active rules in Prometheus have been disabled for this benchmark. As was shown in section 5.2, constraints are an essential feature of Prometheus. They are used to implement the ICBN and provide security to taxonomists that are not allowed to make detectable mistakes (e.g. placing a Genus group into a Species group). However, it is not possible to find a typical constraint that could be used in this benchmark. Some constraints are very simple (e.g. checking the ending of a name), and others are very complex (e.g. checking the placement of a circumscription). Therefore including some of these rules in the benchmark would be at best unfair for Prometheus if they are badly chosen, and at worst would confuse the results of the tests (e.g. testing the time necessary to create an object might trigger hundreds of rules, and what was to be actually measured disappears in the processes associated with it). Therefore, although events are detected and messages are sent through the system to signal actions in the system that might trigger rules (which consumes resources), no rule evaluation takes place. This makes sense, as what this benchmark endeavours to test is the relative performance between the underlying database system (POET) and Prometheus, not the raw performance of Prometheus. Moreover, rules in Prometheus are mapped to queries through PCL (section 5.2.3.3), therefore their performance is a direct function of queries' performance (with the additional cost of event detection and rule scheduling).

Indexing

The OO7 Benchmark makes use of indexing techniques in the definition of the benchmark classes. These indexes are defined on attributes of classes and will provide a faster access to the instances of the classes if they are used (and judiciously chosen). However, this feature is not available in POET and only partially available in Prometheus. In order to proceed to a fair test, no indexes have been defined on user classes.

Disk cache

Disk caching is not taken into account during these tests. It is managed by the system (as an undistinguished part of the operating system in Windows NT that the test computer runs) and as a user application, neither database has access to it. This disk cache may advantage smaller databases that might be loaded entirely in the disk cache during the execution of the test.

7.2.1.4 Benchmark schema

The schema used for the benchmark is shown in Figure 43. It is a simplified version of the OO7 Benchmark with a few minor corrections in order to make the tests more precise (e.g. the extension of the Connection object). All cardinalities and graph connectivity remain as in OO7. The two schemas used for this test are available in Appendix II.

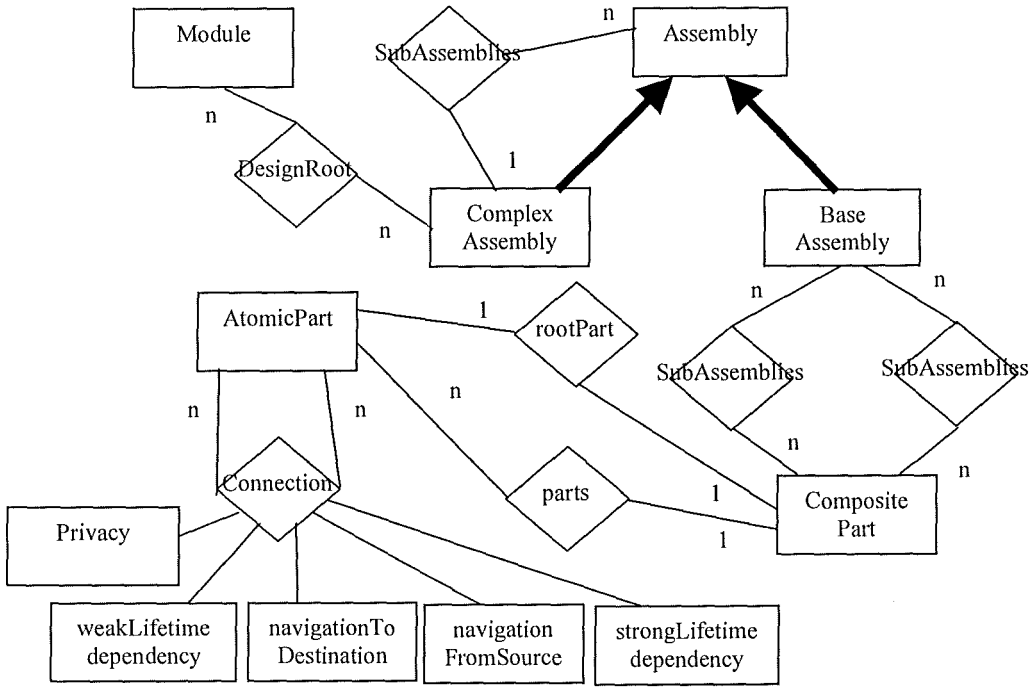


Figure 43: Benchmark schema

Note that except where explicitly stated, all tests are cold tests, i.e. the database cache was empty at the beginning of the test and a new fresh transaction is opened in order to run the test. This choice is motivated by the fact that control over the database cache is impossible in POET. For each test, the OO7 small database with connectivity 3, 6, and 9 has been used.

7.2.2 Analysis

The detailed results of the benchmark can be found in appendix II.2.3. These results show that Prometheus imposes a constant increase in cost for most of the operations (e.g. test (T5) in Figure 44). This proves that Prometheus is as scalable as the underlying database system for these operations. Notable exceptions are the addition and deletion of objects (Figure 45 and Figure 46). Prometheus's cost increases faster than the same operation in the underlying database, i.e. Prometheus is less scalable than the underlying database.

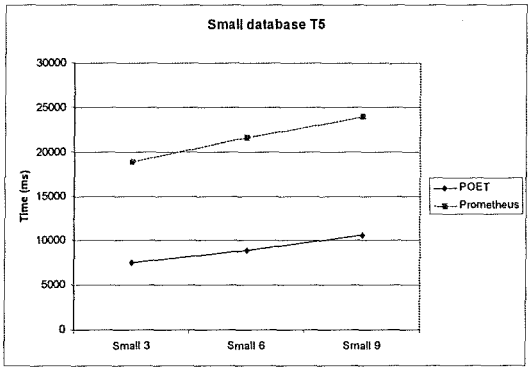


Figure 44: Constant increase in cost (T5)

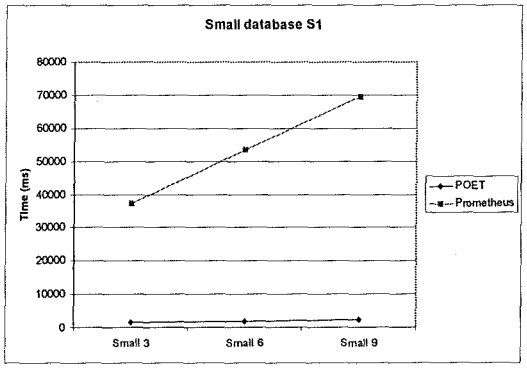


Figure 45: Non-constant increase in cost (S1)

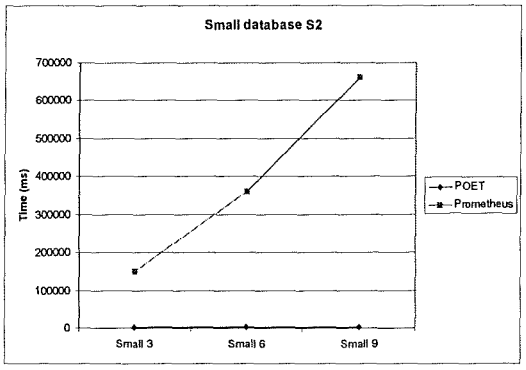


Figure 46: Non-constant increase in cost (S2)

This non-constant increase in cost is mainly due to how the rule layer is implemented and how it interacts with POET, and how indexes are managed and interact with POET.

7.2.2.1 Rule layer

The addition or deletion of objects in Prometheus generates events that allow Prometheus to enforce constraints and fire rules. When an event is detected in POET, Prometheus is notified via the

POETShadow object (see section 6.1.1). That object then notifies other parts of Prometheus that have registered an interest in specific events, e.g. rule monitors, and index manager.

Because Prometheus is implemented as a package independent from the underlying database, this notification requires that some notified objects are loaded from disk or, if they already exist in memory, that they are checked for locks. This happens to objects that are persistent only (PrometheusMonitor and PrometheusIndexManager in Figure 27). The loading or checking of these objects may in turn generate events that trigger event notification within Prometheus. This could not be disabled, as Prometheus' view of rules is that they must be able to do whatever is necessary to react to events, including modifying themselves or Prometheus, and all objects in Prometheus are rule/constraint capable (section 4.2), including Prometheus itself.

The interaction with POET is unfortunate at this point because POET uses an expensive mechanism to check that objects are not shared by several transactions in several running applications. This mechanism involves disk accesses that are very slow (compared to memory-only object access) to check flags attached to objects in the database. It is therefore logical that the notification of events that may generate new events leads to a non-constant increase in cost. If n objects are loaded or checked when an object is created or deleted, then the cost is n. If m objects are created or deleted during a transaction, then the cost is m*n, which is n times more expensive than POET alone (which is not an active system). This trend can clearly be seen in Figure 45 and Figure 46 where the more objects are present in the system, the more events are fired and the more rules are triggered. This cascade of events explains in part the fast increase in cost of inserting and deleting objects.

In order to improve this mechanism, techniques to eliminate unnecessary event propagation must be devised. At the moment examination of the objects that trigger events is performed by the monitors themselves, which allows a good independence of the rule management system from the event management system. However, this is expensive because monitors need to be resolved each time an event is fired. It would be possible to keep track of which objects monitors are interested in outside the definition of monitors (i.e. in the event layer, e.g. in POETShadow) so that they do not need to be resolved to check whether they are interested in the object that triggered an event they monitor. This would be more efficient, but would make event layer and rule layer tightly coupled.

7.2.2.2 Indexes

Prometheus uses indexes to speed up access to object attributes that are represented as explicit relationships (section 6.1.4). These indexes use several hash tables to provide many ways to efficiently access objects and relationships.

When an object is created or deleted, its slot in the index must be found (to be inserted or removed respectively). The comparison of object hashes requires that the objects are available in memory therefore reading an object's hash code involves loading it from the disk or importing it in the current

transaction. In both cases, this implies checking (the object's locking status) or loading information (the object data) from disk. In addition, loading an object from disk generates an event that must be dealt with by the event manager (see section 7.2.2.1). As indexes are a central part of Prometheus, the cost of managing them is amplified by the number of times they are accessed by other components of the system.

One possible approach to solve this problem would be the introduction of representative objects that are transient only. These objects would hold enough information to allow the indexes to work without requiring accessing persistent objects directly. This approach was not considered during the building of the prototype because it introduces the possibility to lose data: if the system is terminated unexpectedly instead of being shut down properly, the changes made to indexes are lost and indexes become inconsistent. Additional mechanisms to ensure the validity of these indexes would need to be developed.

7.2.3 Conclusion

The performance comparison between POET and Prometheus has shown that performance should be one of the next research areas regarding this prototype. Most operations are more expensive with Prometheus than with POET, but many of them (e.g. graph traversal) are only constantly more expensive, i.e. the scalability of the two systems is similar.

However, the tests have also shown that creating and deleting objects in Prometheus is more expensive than updating objects in POET. This is due to the fact that Prometheus manages indexes and is an active system. Creating or deleting objects require the update of indexes and as Prometheus is implemented as a package on top of POET, these updates necessitate the loading of indexes and update of additional index objects. In addition, all operations on objects fire events that are managed by the rule manager. Although the evaluation of events was switched off during the tests, the firing of events was still active because it participate in the normal workings of Prometheus (e.g. to signal the necessity of index updates).

This performance problem became apparent toward the end of the testing period by taxonomists. When the number of classifications and their sizes increase, the speed of the system became an issue.

7.3 Conclusion

This section has presented an evaluation of the system built during this thesis according to two important criteria: its ability to support taxonomic work (and more generally classification work), and an analysis of its performance.

This section has shown that the implementation of the Prometheus model and query language allow the system to support taxonomic work. Prometheus inherently supports the definition of multiple overlapping classifications, but not at the price of the loss of individual classifications (description of contexts and querying in context). Prometheus also provides support for the typical queries taxonomists require to extract knowledge from the database (query language POOL) and for the enforcement of the ICBN (constraints or active rules). In addition, Prometheus is not tied to a particular view of taxonomy and could be used as the basis for different taxonomic philosophies (e.g. taxon based taxonomy as opposed to the current specimen based taxonomy).

The database performance has shown that on simple tasks (e.g. loading objects from the database), Prometheus performs relatively well on some operations. However, the most important drawback is that Prometheus manages events and indexes that are expensive to maintain (e.g. (S1)). These costs are due in part to the choices made at the beginning of the project (underlying database technology, development language, implementation approach as a package, non participation of existing objects to the classification process). Most of these choices were unavoidable, as they are the consequences of a study of the requirements of a taxonomic environment (section 2.4). Approaches to solve these problems have been identified but need more work.

The prototype is therefore very usable and offers clear advantages for taxonomists and the support for their working practices both from the point of view of application development and power, and from the point of view of expressible conceptual queries. But the scalability of the system will become the main issue when large classifications are created and compared.

Further work includes the exploration of new and extensive indexing techniques, and the port of the system to a more powerful underlying database system.

8 Conclusion and discussion

This research work was concerned with the provision of an approach to manipulating multiple overlapping classifications. It was motivated by plant taxonomy (chapter 2) which generates such overlapping classifications and requires handling of complex processes (e.g. naming, section 2.1.2), and which acted as a demonstration application example of the proposed techniques (e.g. section 7.1). The thesis first analysed taxonomy and its requirements (section 2.1) in order to extract the necessary features for a suitable taxonomic database system (section 2.4). Then it was shown that most existing database models fail to some extent to support all the features a taxonomic system needs (section 3.2), but also that some provide interesting mechanisms that could be mixed to create a suitable taxonomic database system. A model providing relationships was then proposed and implemented (chapter 4), along with its query language and rule management system (chapter 5), and finally tested against typical taxonomic processes (section 7.1) and against the underlying persistent system (section 7.2).

This chapter discusses this work and it then explores further work and new application domains for the proposed techniques.

8.1 Discussion

Several aspects of this thesis are discussed here. They follow the development of the document: classifications, model, query language, and rules.

8.1.1 Classifications

The creation of a database system to support taxonomic work requires the ability to handle multiple overlapping classifications (section 2.1.3). Each of these classifications represents a taxonomist's opinion or the refinement of an existing classification because of new information (e.g. new technology, new collections). These peculiar classifications have not been tackled in the literature. Classification mechanisms found are either schema-level classifications (materialization, class hierarchies) or too simplistic to support overlap without loss of information (power types). The properties of taxonomy, the application domain chosen for this work, make these approaches unsuitable as the number of classifying entities is very large (see section 1.1).

These classifications can overlap at all levels, leaves or nodes (specimens or taxa in taxonomy), without ever losing the sense of which classification/classifications is/are currently studied. They are supported in Prometheus by a new approach to classifications: the relationships between classified/classifying entities are the only objects that manage classifications (section 4.6). These relationships can contain information that allows the distinction between distinct classifications, even if

they overlap or are virtually identical. Other classification mechanisms would not have allowed the distinction between classifications because the classification information is either embedded in classified objects (materialisation where the meta-class acts as the classifications status, section 3.1.1) or relationships between classified objects is one to many without information (power types).

In effect, the relationships represent both the classifications and their context (the information that allows the distinction between two distinct classifications, e.g. author and publication in taxonomy).

This approach allows not only the description of overlapping classifications, but also the definition of classifications where the classified objects are independent of their classification status. This is useful for example for extending existing models, or when classified and classifying objects should be independent from the classification process. Other approaches such as materialization would not allow such classifications to be defined because the classified objects manage classifications.

8.1.2 Model

The need for support of classification information led to the description of a new database model that emphasises relationships as well as objects (chapter 4). This approach to implementation has been suggested by many language experts and data modellers (e.g. Rumbaugh in DSM [Shah '89], Consens [Consens '94b]) and satisfies the requirements identified in section 2.4. Relationships are also an integral part of data modelling and have been used for a long time to design systems (e.g. OMT, Booch, UML). They are less common in programming languages/databases because they are often influenced by traditional programming languages, themselves influenced by hardware implementation.

Unlike other model proposals (e.g. OMS), Prometheus provides the means to describe both aggregations and associations with their full semantics independently of the objects that participate in the relationships. The representation of aggregations as references would not allow the description of exact semantics (relationships do not exist per se), or would impose the description of such semantics on the objects that participate in the relationship (as in [Bertino '98]). SORAC showed that this approach is not suitable for at least some semantics and that explicit relationships are a better mechanism [Doherty '93]. However, unlike OMS or Albano's approach, Prometheus does not allow the description of inheritance as a relationship on which constraints can be placed. This is due to the ODMG approach to the problem.

The implementation of explicit relationships described earlier requires the definition of semantics, which has been overlooked by some database proposals (e.g. GraphDB). For example, if relationships are represented as entities external to objects, some means must be provided to represent *privacy*. This point has been ignored by semantic relationships analysis (e.g. [Pirotte '97], [Kolp '97]). The list of basic semantics to be offered to user applications was selected from an analysis of existing semantics specifications (e.g. Rumbaugh, Henderson-Sellers, Bock, Odell). These specifications are sometimes

contradictory, therefore it was not possible to take one specification and use it as the basis for the model. These semantics are therefore not new, but an analysis was performed in order to extract the simplest constituents of all specifications and provide them to the user. This way, it is possible to select several basic building blocks in order to work with one of these specifications or create new ones.

The model proposed has been implemented over an ODMG compliant model. ODMG has been poorly designed regarding semantics of relationships. Even the use of the term "relationship" in ODMG is confusing as it refers to references with inverse references, not actual objects, and not to semantics. Very few attempts have been made to make ODMG a more expressive model. A notable attempt by Bertino [Bertino '98] proposes the definition of constraints/flags on objects in the way Orion did in order to enforce semantics. For this thesis, another approach has been chosen: the definition of explicit semantic relationships. These relationships can capture their own behaviour and constraints, and free objects participating in the relationship from supporting that behaviour (as in SORAC). As a side effect, it makes the definition of semantic relationships simpler as behaviour is defined in only one place (instead of in every object participating in the relationship) and it allows reuse of existing semantics by instantiation/sub-classing of relationship classes (similar constraints do not need to be implemented in a large number of classes). Implementation could be made faster through the use of this technique.

These relationships are therefore used both for the representation of classifications and for the representation of the semantics of data. For classification information, contexts are captured by attributes on classification relationships. For semantic information, constraints and built-in attributes are used. This distinction allows the unambiguous description of the classifications and the objects classified, a feature that non-semantic models could not support fully.

8.1.3 Query language

As the model has been built on top of an ODMG compliant model, the query language chosen as the basis for the system is OQL. However, OQL has several limitations that must be overcome in order to support classification manipulation.

Recursivity is one extension provided for OQL in Prometheus. Recursivity, in particular through regular expressions, is not a new concept. A few object-oriented/semi-structured query languages propose such an approach (e.g. Kifer [Kifer '92], GraphDB, OMS, and others). However, the introduction of regular expressions in an object-oriented language brings new problems, especially problems related to typing/sub-classing (e.g. in the case of the common composite design pattern). Therefore an operator handling these features elegantly has been devised: the selective downcast operator (section 5.1.1.2). This operator is not found anywhere else, and unlike other downcast operators (e.g. OQL's), it selects objects of specific type instead of enforcing down casting (and raising exceptions).

As the query language designed for this work manipulates classifications, a mechanism to handle them as single entities needed to be devised. GraphDB offers such a mechanism by providing path classes that contain data describing a path and being used in queries. This approach has the disadvantage of requiring the creation of new meta-types (path classes and their specific behaviour, which would mean more changes to the ODMG standard), and the extension of the query engine to manipulate them. Another approach has been proposed in this thesis: instead of creating new meta-types, the query language has been extended with an operator borrowed from SQL (the SQL *in* operator). That operator allows the nesting of queries therefore the extraction of classifications in sub-queries and their manipulation in outer queries (this operator was described in section 5.1.2.5).

The exploration of classifications also requires the ability to do two selections at a time: selection of objects and selection of paths through overlapping classifications. The mechanism proposed in this work is the declaration of two selection clauses in queries that go through classifications: the classical selection of objects from the classifications (*where* clause), and the additional selection of path relationships in an external selection (the *in context* clause). That external selection acts as a filter and allows the description of recursive selections (section 5.1.1.3). Existing object-oriented query languages can only support this feature by defining the multiple selections in the unique selection clause. This leads to the loss of path expressions (as they are replaced by multiple joins), which makes queries harder to write and read (see section 5.1.1.3).

8.1.4 Rules

ODMG evokes constraints, but never defines their structure and behaviour. The approach chosen in this work is therefore not in conflict with the ODMG standard. Constraints are objects that can be attached to other objects.

Prometheus' constraints/rules are ECC (or ECCA if an action is necessary) instead of the classical ECA rules (e.g. NAOS [Collet '94], SAMOS [Gatziu '91]). The majority of the rules handled by the system are context-dependant rules. They therefore need to check that they are applicable in the context in which they are activated. This applicability is captured by the first condition of the rule, which stops the evaluation of the rule without error if it is violated. The second condition is the condition of activation of the action part of the rule, or the constraint not to be violated. Violation of the second constraint requires action (e.g. abortion of the transaction).

Unlike most other rule mechanisms, Prometheus' rules can be user-triggered as well as system-triggered. User triggering of rules allows the implementation of assertions that support programming by contract [Meyer '97] and a safer design of applications. System triggering allows the enforcement of constraints and the execution of rules as can be found in many object-oriented databases (e.g. O_2 [Collet '94], ADAM [Díaz '91]).

The activation and scheduling mechanisms are not particularly new in Prometheus. Unlike Ode [Gehani '91] or Jasmine [Ishikawa '93], Prometheus does not support the definition of composite events, although alternatives (OR) are supported through multiple inheritance of rules. Prometheus does not support complex execution models with conflict detection or selection of the database state in which rules are evaluated. Prometheus only proposes immediate and deferred constraints that work in a similar way to hard and soft constraints in Ode.

A peculiarity of Prometheus over other rules mechanisms is that it does not necessarily abort transactions in which a constraint has been violated. Constraint violation raises exceptions that can be handled by other parts of user applications. Constraints can also be interactive, i.e. the user is prompted for an action (section 5.2.2.2). This is motivated by the fact that taxonomy may contain a lot of errors or changes in the botanical code over time, and may require authorised constraint violations that only taxonomists may decide.

8.2 Other application domains

The principles described here could be generalised and applied to new domains. Classifications are not useful only to taxonomists. The model and the query language are not limited to taxonomy, but are suitable for a wider range of applications. The particular group of applications that Prometheus tackles are applications that require the definition of multiple classifications, either simultaneously or over time, and require the ability to compare them. For example, a library information system can make use of such a system, as the same books may be classified differently depending on the point of view of the user (e.g. a book can appear both in English writings and fiction), or the categories may overlap and be shared between different views. The applications that would benefit most from the model presented here are applications that require working within contexts. For example, the overlapping classifications could also be used to solve the problem of context in ontologies [Priss '01]. Relationships would hold information describing the context in which concepts are defined. This would allow the navigation of the database according to contexts as it currently allows the navigation within chosen classifications (which details are also called contexts, section 4.6.2). Other science domains use classifications to order knowledge (e.g. protein research, genetic research, genome research). These classifications may conflict and therefore require an approach as the one proposed in Prometheus to represent all knowledge without forcing users into making decision or choosing a favourite view of the world.

Other features offered by Prometheus could be of interest specifically to the database community. For example, instance synonyms would be a suitable mechanism to integrate data from redundant or overlapping independent sources. These data sources may describe the same information under different forms or to different degrees of exactitude and completeness. As instance synonyms allow the grouping of several instances into one conceptual entity that captures the real world entity described, they could group all these representations extracted from these data sources and simplify knowledge

without losing information (e.g. errors may be of interest). In addition, their integration with the query language makes them powerful not only for capturing data, but also for its retrieval. This way, grouping representations into one entity would simulate querying on all data sources simultaneously, making it possible to optimise the querying.

8.3 Further work

As discussed in the evaluation section, Prometheus suffers from a performance problem. Although performance was not the focus of attention during this work, it becomes an important issue when users interact with the system. The reasons for this performance problem have been explained in section 7.2.2 and some of them can be resolved. For example, the addition of more indexes, especially on node and transient objects, could improve greatly the overall performance of the system. A less naive evaluation of queries, based for example on a set oriented algebra, can also improve the performance of the system. For example, the representation of knowledge about the data stored can help (e.g. the consideration of cardinalities in order to evaluate queries in a bottom-up fashion instead of the current top-down approach). Specific efficient graph-traversing techniques could be investigated or created. Therefore, future work includes the integration of such techniques in order to provide a more efficient system.

However, some of the limitations of Prometheus are not easily solvable. For example, the package approach will always mean that some information is out of reach of Prometheus and that accessing objects means instantiating and resolving them. As long as this approach is favoured, Prometheus will suffer performance problems. Whether this is an acceptable price for increased portability is debatable. If portability is not an issue, reducing the independence of Prometheus vis-à-vis its underlying storage system could be considered and would likely provide significant performance improvements.

In addition, a distributed version of Prometheus could be an interesting development for taxonomists in particular, and many other types of users in general. As was said in section 2.1, taxonomists work generally on small parts of the plant knowledge available to them because they are only interested in information relating to a single plant group. Furthermore, they tend to work in their area of expertise and are not so concerned with others. The consequence of such patterns of work is isolated database systems, each of them containing detailed information relating to a very limited number of plant groups. This situation is acceptable for most taxonomists. But as the general public (e.g. for conservation or protection purposes) and other areas of biology take interest in the output from taxonomy (e.g. pharmaceutical research centres in order to name their specimens or access information about plant groups), larger information repositories are necessary. The building of centralised repositories have their disadvantages: they are expensive because they must support more important usage loads; they need to be available at all times; and there is the risk that they are not up-to-date, as certain categories of users (e.g. taxonomists) need to update the information they are responsible for on a regular basis. Instead of a central approach, it would therefore be more sensible to adopt a distributed

approach where taxonomists make (part of) their information available to others and a mechanism allows searches, retrieval, and sharing of information over the network of databases. This approach would also provide a way to improve scalability of the system, as each independent database would manage relatively small amounts of data.

However, these local repositories may be unreliable, as they would run on personal computers or small servers. Therefore the distribution of Prometheus would require the investigation of techniques allowing working in such an environment (e.g. duplication, network caching). Currently, Prometheus does not support the management of distributed sources of information, but the implementation of such mechanisms could be part of its further development. Another problem generated by distributed repositories is the overlap that exists between distinct classifications: taxonomists proceed to revisions of existing groups or work in parallel on similar plant groups. Therefore the distribution of Prometheus would also require the investigation of integration techniques that would allow connection of information between independent databases and would avoid the unnecessary duplication of information across data sources.

Currently, Prometheus only records the result of taxonomic work: the classifications are what taxonomists produce once they have studied and ordered plant specimens or plant groups. Another further area of investigation is the recording of the reasons why classifications are created, i.e. what the criteria were that resulted in the decisions made by taxonomists during the classification process. These decisions usually involve examination of specimens, therefore the characters these specimens exhibit. This information would provide new insights in the data generated by other taxonomists, as it would be possible to understand what they mean when they describe a group, and follow their reasoning. This is the focus of a new research project in which features offered by Prometheus are used.

9 References

[Abiteboul '98] S. Abiteboul, J. Mc Hugh, M. Rys, V. Vassalos and J. Wiener, *Incremental maintenance for materialized views over semistructured data*, VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York City, New York, USA, pp 38-49, 1998

[Abiteboul '96] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J.L. Wiener, *The Lorel Query Language for Semistructured Data*, International Journal on Digital Libraries, 1 (1), pp 68-88, 1996

[Abiteboul '95] S. Abiteboul and C. Souza dos Santos, *IQL(2) : A Model with Ubiquitous Objects*, Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, pp 10, 1995

[Abiteboul '97] Serge Abiteboul, *Querying semi-structured data*, Proceedings of the International Conference on Database Theory, Delphi, Greece, pp 1-18, 1997

[Alagic '99] Suad Alagic, *Type-Checking OQL Queries In the ODMG Type Systems*, ACM Transactions on Database Systems, 24 (3), pp 319-360, 1999

[Albano '91] Antonio Albano, Giorgio Ghelli and Renzo Orsini, *A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language*, Proceedings of the seventeenth international conference on very large data bases, Barcelona, Spain, pp 565-575, 1991

[Amann '92] Bernd Amann and Michel Scholl, *Gram: A Graph Data Model and Query Language*, INRIA, Le Chesnay, France, Verso report number 046 (ECHT), 1992

[Arocena '98] Gustavo O. Arocena and Alberto O. Mendelzon, *WebOQL: Restructuring Documents, Databases, and Webs*, Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, pp 24-33, 1998

[Ashish '97/12] Naveen Ashish and Craig A. Knoblock, *Wrapper Generation for Semi-structured Internet Sources*, SIGMOD Record, 26 (4), pp 8-15, 1997/12

[Atkinson '90] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier and Stanley B. Zdonik, *The Object-Oriented Database System Manifesto*, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto research Park, Kyoto, Japan, pp 223-240, 1990

[Banerjee '87] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk and N. Ballou, *Data model issues for object-oriented applications*, ACM Transactions on Office Information Systems, 5 (1), pp pp 3-26, 1987

[Barbier '01] Franck Barbier and Brian Henderson-Sellers, *The whole-part relationship in object modelling: a definition in cOlOr*, Information & Software Technology, 43 (1), pp 19-39, 2001

[Barclay '94] P. J Barclay and J. B. Kennedy, *A Conceptual Language for Querying Object Oriented Data*, British National Conference On Databases, BNCOD-12, Guildford, UK, pp 187-204, 1994

[Barsalou '91] T. Barsalou, A. M. Keller, N. Siambela and G. Wiederhold, *Updating relational Databases through Object-Based Views*, Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, pp 248-257, 1991

[Bellahsene '97] Zohra Bellahsene, *Updating Virtual Complex Objects*, OOIS'97, 1997 International Conference on Object Oriented Information Systems, Brisbane, Australia, pp 422-432, 1997

[Bellahsene '96] Zohra Bellahsene, *View Mechanism for Schema Evolution in Object-Oriented DBMS*, 14th British National Conferenc on Databases, BNCOD 14, Edinburgh, Scotland, pp 18-35, 1996

[Berendsohn '97] W. Berendsohn, *A taxonomic information model for botanical databases: the IOPI model*, Taxon, 46, pp 283-309, 1997

[Berendsohn '99] W. G. Berendsohn, A. Anagnostopoulos, G. agedorn, J. Jakupovic, P. L. Nimis, B. Valdés, A. Güntsch, R. J. Pankhurst and R. J. White, *A comprehensive reference model for biological collections and surveys*, Taxon, 48, pp 511-562, 1999

[Bertino '98] Elisa Bertino and Giovanna Guerrini, *Extending the ODMG Object Model with Composite Objects*, Thirteenth International Conference on Object-Oriented Programming: Systems, Languages, and Applications, Vancouver, Canada, pp 259-270, 1998

[Bertino '93] Elisa Bertino and Lorenzo Martino, *Object-Oriented Database Systems, Concepts and Architectures*, Addison-Wesley, I.S.B.N. 0-201-62439-7, 1993

[Blaha '93] Michael Blaha, *Aggregation of parts of parts of parts*, Journal of Object-Oriented Programming, 6 (5), pp 14-22, 1993

[Bock '94] Conrad Bock and James Odel, *A foundation for composition*, Journal of Object-Oriented Programming, 7 (6), pp 10-14, 1994

[Bock '98] Conrad Bock and James Odell, *A More Complete Model of Relations and their Implementation: Aggregation*, Journal of Oject-Oriented Programming, 11 (5), pp 68-70, 1998

[Borgida '88] Alexander Borgida, *Modeling Class Hierarchies with Contradictions*, Proc. ACM SIGMOD '88 Conference, Chicago, USA, pp 434--443, 1988

[Borgida '89] Alexander Borgida, *Type systems for querying class hierarchies with non-strictinheritance*, Proc. ACM PODS Conference, Philadelphia, USA, pp 394--400, 1989

[Bowker '99] Geoffrey C. Bowker and Susan Leigh Star, *Sorting things out, classification and its consequences*, Massachusetts Institute of Technology, I.S.B.N. 0-262-02461-6, 1999

[Brady '97] Declan Brady and John Murphy, *Relational Vs Object-Oriented Database Systems*, Dublin City University, Dublin, CA-1497, 1997

[Bretl '89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchart, Jacob Stein, E. Harold Williams and Monty Williams, *The Gemstone Data Management System*, in *Object-Oriented Concepts, Databases, and Applications*, pp 283-308, ACM Press, 1989

[Buneman '96] Peter Buneman, Susan Davidson, Gerd Hillebrand and Dan Suciu, *A query language and optimization techniques for unstructured data*, Proceedings of ACM-SIGMOD International Conference on Management of Data, Montreal, Canada, pp 505-516, 1996

[Buneman '95] Peter Buneman, Susan Davidson and Dan Suciu, *Programming Constructs for Unstructured Data*, DBPL-5 Proceedings of the Workshop on Database Programming Languages, Gubbio, Umbria, Italy, pp 12, 1995

[Bussche '92] J. Van den Bussche, D. Van Gucht, M. Andries and M. Gyssens, *On the completeness of object-creating query languages*, Proceedings 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Pittsburgh, Pennsylvania, USA, pp 372--379, 1992

[Calvanese '99] Diego Calvanese, Giuseppe De Giacomo and Maurizio Lenzerini, *Queries and Constraints on Semi-Structured Data*, Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering (CAiSE'99), Heidelberg, Germany, pp 434-438, 1999

[Cannan '92] S. J. Cannan and G. A. M. Otten, *SQL - The Standard Handbook*, McGraw-Hill, 1992

[Carey '93] Michael J. Carey, David J. DeWitt and Jeffrey F. Naughton, *The OO7 Benchmark*, 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., pp 12-21, 1993

[Catarci '95] T. Catarci and L. Tarantino, *A Hypergraph-based Framework for Visual Interaction with Databases*, Journal of Visual Languages and Computing, 6 (2), pp 135-166, 1995

[Cattell '97] R. G. G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland and Drew Wade, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, Inc., I.S.B.N. 1-55860-463-4, 1997

[Chakravarthy '94] S. Chakravarthy, V. Krishnaprasad, E. Anwar and S.-K. Kim, *Composite Events for Active Databases: Semantics, Contexts and Detection*, 20th International Conference on Very Large Data Bases, Santiago, Chile, pp 606-617, 1994

[Chawathe '94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman and J. Widom, *The TSIMMIS Project: Integration of Heterogeneous Information Sources*, Proceedings of IPSJ Conference, Tokyo, Japan, pp 7-18, 1994

[Christophides '96] Vassilis Christophides, Sophie Cluet and Guido Moerkotte, *Evaluating Queries with Generalized Path Expressions*, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, pp 413-422, 1996

[Cluet '98] Sophie Cluet, *Designing OQL: Allowing Objects to be Queried*, Information Systems, 23, pp 279-305, 1998

[Codd '70] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM (CACM), 13 (6), pp 377-387, 1970

[Collet '94] C. Collet, T. Coupaye and T.Svensen, *NAOS: Efficient and modular reactive capabilities in an Object-Oriented Database System*, 20th International Conference on Very Large Data Bases, Santiago, Chile, pp 132-143, 1994

[Consens '94a] Mariano P. Consens, *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*, Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, 1994a

[Consens '94b] Mariano P. Consens, Alberto O. Mendelzon and Arthur G. Ryman, *Looking at the Relations Among Software Objects*, November 10, 1994, 1994b

[Crestana-Taube '96] V. Crestana-Taube and E. A. Rundensteiner, *Schema Removal Issues for Transparent Schema Evolution*, Sixth International Workshop on Research Issues on Data Engineering, Interoperability of Non traditional Database Systems, RIDE'96, IEEE, New Orleans, Louisiana, pp 138-147, 1996

[Darwen '95] H. Darwen and C. J. Date, *The Third Manifesto*, SIGMOD Record 24, 24 (1), pp 39-49, 1995

[Delobel '95] C. Delobel, C. Souza dos Santos and D. Tallot, *Object View of Relations*, INRIA, Rocquencourt, France, Verso report number 072 (INFORSID 95), February 1995, 1995

[Deßloch '92] S. Deßloch, T. Härder, F.-J. Leick, N.M. Mattos, C. Laasch, C. Rich, M. Scholl and H.-J. Schek, *COCOON and FRISYS - a comparison* -, 1992

[Díaz '90] Oscar Díaz and Peter M. D. Gray, *Semantic-rich User-defined Relationship as a Main Constructor in Object Oriented Database*, Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proceedings of the IFIP TC2/WG 2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction, Windermere, UK, pp 207-224, 1990

[Díaz '91] Oscar Díaz, Normal Paton and Peter Gray, *Rule Management in Object-Oriented Databases: A Uniform Approach*, 17th International Conference on Very Large Data Bases, Barcelona, Spain, pp 317-326, 1991

[Doherty '93] Michael Doherty, Joan Peckham and Victor Fay Wolfe, *Implementing Relationships and Constraints in an Object-Oriented Database Using a Monitor Construct*, in *Rules in Database Systems*, pp 347-363, Springer-Verlag, 1993

[Fahl '97] Gustav Fahl and Tore Risch, *Query processing over object views of relational data*, VLDB Journal, 6 (4), pp 261-281, 1997

[Fernandez '97a] Mary F. Fernandez, Lucian Popa and Dan Suciu, *A Structure-Based Approach to Querying Semi-Structured Data*, Database Programming Languages, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, pp 136-159, 1997a

[Fernandez '97b] Mary Fernandez, Dana Florescu, Alon Levy and Dan Suciu, *A Query Language for a Web-Site Management System*, SIGMOD Record, 26 (3), pp 4-11, 1997b

[Fernandez '99] Mary Fernandez, Dana Florescu, Alon Levy and Dan Suciu, *Verifying Integrity Constraints on Web Sites*, IJCAI-99, the Sixteenth International Joint Conference on Artificial Intelligence, Stockholm, Sweden, pp 614-619, 1999

[Filer '94] D. L. Filer, *Botanical Research and Herbarium Management System. A pocket introduction and demonstration guide*, Oxford Forestry Institute, Oxford, 1994, 1994

[Fishman '89] D. H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J.W. Davis, W. Hasan, C. G. Hoch, W. Kent, W. Lwichner, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. Rish, M. C. Shan and W. K. Wilkinson, *Overview of the Iris DBMS*, in *Object-Oriented Concepts, Databases, and Applications*, pp 219-250, ACM Press, 1989

[Fong '97] Joseph Fong, *Converting Relational to Object-Orented Databases*, ACM SIGMOD, (26), pp 53-58, 1997

[Gamma '94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, I.S.B.N. 0-201-63361-2, 1994

[Gatziu '91] Stella Gatziu, Andreas Geppert and Klaus R. Dittrich, *Integrating Active Concepts into an Object-Oriented Database System*, Proc. 3. Intl. Workshop on Database Programming Languages (DBPL), Nafplion, Greece, pp 399-415, 1991

[Gehani '91] N. Gehani and H. V. Jagadish, *Ode as an Active Database: Constraints and Triggers*, Proceedings of Conference on Very Large Databases, Barcelona, Spain, pp 327-336, 1991

[Gemis '93] M. Gemis, J. Paredaens, I. Thyssens and J. Van den Bussche, *GOOD, A graph-Oriented Database System*, Proceedings of SIGMOD, SIGMOD Record, 22 (2), pp 505--510, 1993

[Goldman '97] Roy Goldman and Jennifer Widom, *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*, VLDB'97, Proceedings of 23th International Conference on Very Large Data Bases, Athens, Greece, pp 436-445, 1997

[Gottlob '96] Georg Gottlob, Michael Schrefl and Brigitte Röck, *Extending Object-Oriented Systems with Roles*, ACM Transactions on Information Systems, 14 (3), pp 268-296, 1996

[Greuter '94] W. Greuter, F. R. Barrie, H. M. Burdet, W. G. Chaloner, V. Demoulin, D. L. Hawksworth, P. M. Jørgensen, D. H. Nicolson, P. C. Silva, P. Trehane and J. McNe, *International code of botanical nomenclature (Tokyo Code)*, Koeltz Scientific Books, I.S.B.N. 3-87429-367-X, 1994

[Grumbach '99] Stephane Grumbach, Philippe Rigaux and Luc Segoufin, *Notes on Query Evaluation and Optimization*, INRIA, Verso report number 172, Le Chesnay, France, 1999

[Güting '94] Ralf Hartmut Güting, *GraphDB: Modeling and Querying Graphs in Databases*, Proc 20th Int. Conf. on Very Large Databases, Santiago, Chile, pp 297-308, 1994

[Henderson-Sellers '97] B. Henderson-Sellers, *OPEN Relationships - Compositions and Containments*, Journal of Object-Oriented Programming, 10 (7), pp 51-55, 1997

[Hohenstein '96] Uwe Hohenstein and Volkmar Plesser, *Semantic Enrichment: A First Step to provide Database Interoperability*, Workshop Föderierte Datenbanken, Magdeburg, Germany, pp 3-17, 1996

[Hori '95] Nobuo Hori, Masatoshi Yoshikawa and Shunsuke Uemura, *ASKA: An Object-Oriented Data Model with Multiple Hierarchies and Multiple Object-Perspectives*, 6th Int. Conf. and Workshop on Database and Expert Systems Applications (DEXA'95) - Workshop Proceedings, London, UK, pp 142-151, 1995

[Iivari '92] J. Iivari, *Relationships, aggregations and complex objects*, in *Information Modelling and Knowledge Bases*, pp 141-159, IOS Press, 1992

[Irani '93] B.B. Irani, *Implementation of the TIGUKAT Object Model*, University of Alberta, Edmonton, TR93-10, June 1993, 1993

[Ishikawa '93] Hiroshi Ishikawa, Fumio Suzuki, Fumihiko Kozakura, Akifumi Makinouchi, Mika Miyagishima, Yoshio Izumida, Masaaki Aoshima and Yasuo Yamane, *The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System*, ACM Transactions on Database Systems, 18 (1), pp 1-50, 1993

[Jarke '95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt and S. Eherer, *ConceptBase - a deductive object base for meta data management.*, Journal of Intelligent Information Systems. Special Issue on Advances in Deductive Object-Oriented Databases, 4 (2), pp 167-192, 1995

[Jarke '93] M. Jarke and M. Staudt, *An application perspective to deductive object bases*, ACM-SIGMOD Workshop on Combining Declarative and Object-Oriented Databases, Washington D.C., pp 17-30, 1993

[Joseph '91] John Joseph, Satish Thatte, Craig Thompson and David Wells, *Object-Oriented Databases: Design and Implementation*, Proceedings of the IEEE, 79 (1), pp 42-64, 1991

[Keller '94] Arthur M. Keller, *Penguin: Objects for Programs, Relations for Persistence*, 1994

[Keller '93] Arthur M. Keller, Richard Jensen and Shailesh Agarwal, *Persistence Software: Bridging Object-Oriented Programming and Relational Databases*, SIGMOD, (May 1993), 1993

[Khoshafian '86] Setrag N. Khoshafian and George P. Copeland, *Object Identity*, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, pp 406-416, 1986

[Kifer '92] Michael Kifer, Won Kim and Yehoshua Sagiv, *Querying Object-Oriented Databases*, Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, pp 393-402, 1992

[Kim '89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge G. Garza and Darell Woelk, *Features of the ORION Object-Oriented Database System*, in *Obje-Oriented Concepts, Databases, and Applications*, pp 251-282, ACM Press, 1989

[Kim '87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza and Darell Woelk, *Composite Object Support in an Object-Oriented Database System*, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), Orlando, Florida, USA, pp 118-125, 1987

[Kim '95] Won Kim and William Kelley, *On View Support in Object-Oriented Database Systems*, in *Modern database systems: The Object Model, Interoperability, and Beyond*, pp 108-129, Addison-Wesley Publishing Company, 1995

[Kolp '97] Manuel Kolp, *A Metaobject Protocol for Reifying Semantig Relationships into Reflexive Systems*, 4th doctoral Consortium of the 9th Conference on Advanced Information Systems Engineering (CAise'97), Barcelona, Spain, pp 89-100, 1997

[Koymen '93] Kemal Koymen and Qujun Cai, *SQL*: a recursive SQL*, Information Systems, 18 (2), pp 121-128, 1993

[Kuno '95a] H. A. Kuno and E. A. Rundensteiner, *Materialized Object-Oriented Views in MultiView*, Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management (RIDE-DOM'95), IEEE, 1995, Taipei, Taiwan, Taipei, Taiwan, pp 78-85, 1995a

[Kuno '95b] H. Kuno, Y.G. Ra and E. A. Rundensteiner, *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation*, Electrical Engineering and Computer Science Dept., University of Michigan, Ann Arbor, CSE-TR-241-95, April 1995, 1995b

[Kuno '98] Harumi A. Kuno and Elke A. Rundensteiner, *Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation*, Knowledge and Data Engineering, 10 (5), pp 768-792, 1998

[Leung '93] Throdore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg and Stanley B. Zdonik, *The Aqua Data Model and Algebra*, Brown University, Providence, Rhode Island, CS-93-09, 1993

[Maier '90] David Maier and Jacob Stein, *Development and Implementation of an Object-Oriented DBMS*, in *Readings in object-oriented database systems*, pp 167-185, 1990

[McHugh '97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass and J. Widom, *Lore: A Database Management System for Semistructured Data*, SIGMOD Record, 26 (3), pp 54-66, 1997

[Mendelzon '97] Alberto O. Mendelzon, George A. Mihaila and Tova Milo, *Querying the World Wide Web*, Int. J. on Digital Libraries, 1 (1), pp 54-67, 1997

[Mendelzon '98] Alberto O. Mendelzon and Tova Milo, *Formal Models of Web Queries*, Information Systems, 23 (8), pp 615-637, 1998

[Meyer '97] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, I.S.B.N. 0-13-629155-4, 1997

[Mitchell '93] Gail Anne Mitchell, *Extensible Query Processing in an Object-Oriented Database*, thesis, Department of computer science, Brown University, Providence, Rhode Island, 1993

[Mitchell '91] Gail Mitchell, Stanley B. Zdonik and Umeshwar Dayal, *Object-Oriented Query Optimization: What's the Problem?*, Brown University, Providence, Rhode Island, CS-91-41, June 1991, 1991

[Motschnig '99] Renate Motschnig and Jens Kaasbøll, *Part-Whole elationship Categories and their Application in Object-Oriented Analysis*, IEEE Transactions on Knowledge and Data Engineering, 11 (5), pp 779-797, 1999

[Münch '98] Manfred Münch, Andy Schürr and Andreas Winter, *Integrity Constraints in the Multi-paradigm Language PROGRES*, Declarative Systems and Software Engineering Group, University of Southampton, UK, Technical report DSSE-TR-98-2, February 13 1998, 1998

[Mylopoulos '90] John Mylopoulos, Alex Borgida, Matthias Jarke and Manolis Koubarakis, *Telos: Representing Knowledge About Information Systems*, ACM Transactions on Information Systems, 8 (4), pp 325-362, 1990

[Nestorov '97] S. Nestorov, J. Ullman, J. Wiener and S. Chawathe, *Representative Objects: Concise Representations of Semistructured, Hierarchial Data*, Proceedings of the 13th International Conference on Data Engineering (ICDE'97), Birmingham, U.K., pp 79-90, 1997

[Nestorov '97/12] Svetlozar Nestorov, Serge Abiteboul and Rajeev Motwani, *Inferring Structure in Semistructured Data*, SIGMOD Record, 26 (4), pp 39-43, 1997/12

[Newman '00] Mark Newman, Martin Pullan and Mark Watson, *Using Prometheus to produce a taxonomic revision*, Royal Botanic Garden, Edinburgh, Technical report Prometheus #10, 2000

[Norrie '95] M. C. Norrie, *Distinguishing Typing and Classification in Object Data Models*, in *Information Modelling and Knowledge Bases VI*, IOS, 1995

[Norrie '92] Moira C. Norrie, *A collection model for data management in object-oriented systems*, PhD thesis, Department of computing science, University of Glasgow, Glasgow, 1992

[Norrie '93] Moira C. Norrie, *An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems*, Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, pp 390-401, 1993

[O2 '92] O2, *Building an object-oriented database system, the story of O2*, Morgan Kaufmann publishers, San Mateo, California, I.S.B.N. 1-55860-169-4, 1992

[Odell '94a] J. Odell, *Power types*, Journal of Object-Oriented Programming, 7 (2), pp 8-12, 1994a

[Odell '94b] James Odell, *Six different kinds of composition*, Journal of Object-Oriented Programming, 6 (8), pp 10-15, 1994b

[OMG '99] OMG, *The OMG Unified Modeling Language Specification*, Rational, http://www.omg.org/uml, Version 1.3, 1999

[Oracle '01] Oracle, *Oracle, http:///www.oracle.com*, 2001

[Ozsu '95] M. T. Ozsu, R. J. Peters, D Szafrom, B. Irani, A. Lipka and A. Munoz, *TIGUKAT: A Uniform Behavioral Objectbase Management System*, VLDB Journal, 4 (3), pp 445-492, 1995

[Pankhurst '93] R. J. Pankhurst, *Taxonomic Databases: The PANDORA System*, Advances in computer methods for systematic biology: Articificial Intelligence, databases, computer vision, Baltimore, USA, pp 229-240, 1993

[Papakonstantinou '95] Yannis Papakonstantinou, Hector Garcia-Molina and Jennifer Widom, *Object Exchange Across Heterogeneous Information Sources*, Proceedings of the Eleventh International Conference on Data Engineering, Taipei, Taiwan, pp 251-260, 1995

[Paredaens '93] J. Paredaens, P. Peelman and L. Tanca, *Merging Graph Based and Rule Based Computation*, Workshop in Computing, Rules in Database Systems, Edinburgh, UK, pp 212-233, 1993

[Paton '99] Norman W. Paton and Oscar Diaz, *Active Database Systems*, ACM Computing Surveys, 31 (1), pp 63-103, 1999

[Peters '93a] R. J. Peters, A. Lipka, M. T. Ozsu, D. Szafrom, B. Irani and A. Munoz, *TIGUKAT Object Management System: Initial Design and Current Directions*, Proceedings of CASCON'93, Toronto, Canada, pp 595-611, 1993a

[Peters '93b] Randal J. Peters, Anna Lipka, M. Tamer Ozsu and Duane Szafron, *The Query Model and Query Language of TIGUKAT*, University of Alberta, Edmonton, TR 93-01, January 1993, 1993b

[Pirotte '94] Alain Pirotte, Esteban Zimányi, David Massart and Tatiana Yakusheva, *Materialization: a powerfull and ubiquitous abstraction pattern*, Very Lage Data Bases (VLDB'94), Santiago, Chile, pp 630-641, 1994

[Pirotte '97] M. Kolp and A. Pirotte, *An aggregation model and its C++ implementation*, 4th Int. Conf. on Object-Oriented Information Systems, OOIS'97, Brisbane, Australia, pp 211-224, 1997

[POET '01] POET, *POET Object-Oriented Database*, http://www.poet.com, 2001

[Poulovassilis '98] A. Poulovassilis and S. Hild, *Hyperlog: a graph-based system for database browsing, querying and update*, To appear in IEEE Knowledge and Data Engineering, 1998

[Priss '01] Uta Priss, *Ontologies and Context*, 12th Midwest Artificial Intelligence and Cognitive Science Conference, Miami University, Oxford, OH, USA, pp 12-13, 2001

[Pullan '00] Martin R. Pullan, Mark F. Watson, Jessie B. Kennedy, Cedric Raguenaud and Roger Hyam, *The Prometheus Taxonomic Model: a practical approach to representing multiple taxonomies*, Taxon, 49 (1), pp 55-75, 2000

[Quass '95] Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman and Jennifer Widom, *Querying Semistructured Heterogeneous Information*, Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, Singapore, pp 319-344, 1995

[Raguenaud '01] Cédric Raguenaud, *Support for classification mechanisms in existing database models*, School of Computing, Napier University, Edinburgh, Technical report 2001

[Raguenaud '00] Cedric Raguenaud, Jessie Kennedy and Peter J. Barclay, *A database system for supporting taxonomic work*, School of Computing, Napier University, Edinburgh, Scotland, Technical Report Prometheus #5, 2000

[Ramanathan '97] Shekar Ramanathan, *Extraction of Object-Oriented Structures from Existing Relational Databases*, SIGMOD Record, 26 (1), pp 59-64, 1997

[Richardson '91] Joel Richardson and Peter Schwarz, *Aspects: Extending Objects to Support Multiple, Independent Roles*, 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA, pp 298-307, 1991

[Riedel '97] Holger Riedel and Marc H. Scholl, *A Formalization of ODMG Queries*, Database Semantics DS-7, Leysin, Switzerland, 1997

[Rodgers '97] P. J. Rodgers and P. J. H. King, *A graph rewriting visual language for database programming*, Journal of Virual Languages & Computing, 8 (5/6), pp 641-674, 1997

[Rousset '97] Marie-Christine Rousset, *Verifying the Web: a Position Statement*, Proceedings of the 4th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97), Leuven, Belgium, pp 95-103, 1997

[Rumbaugh '94] James Rumbaugh, *Building boxes: Composite objects*, Journal of Object-Oriented Programming, 7 (7), pp 12-22, 1994

[Rumbaugh '88] James Rumbaugh, *Controlling Propagation of Operations using Attributes of Relations*, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88), San Diego, California, USA, pp 285-296, 1988

[Rumbaugh '95] James Rumbaugh, *OMT: The object model*, Journal of Object-Oriented Programming, 7 (8), pp 21-27, 1995

[Rumbaugh '91] James Rumbaugh, Michael Blaha, William Premerlani, Federick Eddy and William Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall International Editions, 1991

[Rundensteiner '94] E. A. Rundensteiner, *A Classification Algorithm For Supporting Object-Oriented Views*, Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, pp 18 - 25, 1994

[Rundensteiner '92a] Elke A. Rundensteiner, *MultiView: A Methodology for Supporting Multiple View in Object-Oriented Databases*, 18th International Conference on Very Large Data Bases, Vancouver, Canada, pp 187-198, 1992a

[Rundensteiner '92b] Elke A. Rundensteiner and Lubomir Bic, *Automatic View Schema Generation in Object-Oriented Databases*, Department of Information and Computer Science, University of California, Irvine, 92-15, 01/92, 1992b

[Saarenmaa '95] H. Saarenmaa, S. Leppäjärvi, J. Perttunen and J. Saarikko, *Object- oriented taxonomic biodiversity databases on the World Wide Web*, Internet Applications and Electronic Information Resources in Forestry and Environmental Sciences, Workshop at the European Forest Institute, EFI Proceedings 3, Joensuu, Finland, pp 121-128, 1995

[Santos '94] Cassio Souza dos Santos, Serge Abiteboul and Claude Delobel, *Virtual Schemas and Bases*, Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, pp 81-94, 1994

[Schek '91] M.E. Scholl and H.-J. Schek, *Supporting Views in Object-Oriented Databases*, IEEE Database Engineering Bulletin, Special Issue on Foundations of Object-Oriented Database Systems, 14 (2), pp 43-47, 1991

[Scholl '93] M.H. Scholl, C. Laasch, C. Rich, M. Tresch and H.-J. Schek, *The COCOON Object Model*, ETH Zürich, Dept. of Computer Science, Zürich, 193, December 1992, 1993

[Schürr '98] Andy Schürr, Andreas J. Winter and Albert Zündorf, *PROGRES: Language and Environment*, in *Handbook on Graph Grammars: Applications*, 1998

[Shah '89] Ashwin V. Shah, James E. Rumbaugh, Jung H. Hamel and Renee A. Borsari, *DSM: An Object-Relationship Modelling Language*, Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), New Orleans, Louisiana, pp 191-202, 1989

[Stein '89] Lynn Andrea Stein and Stanley B. Zdonik, *Clovers: The Dynamic Behavior of Types and Instances*, Brown University, Providence, USA, CS-89-42, 1989

[Stonebraker '90] M. Stonebraker, A. Jhingran, J. Goh and S. Potamianos, *On rules, procedures, caching and views in database systems*, ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, pp 259-270, 1990

[Straube '90] Dave D. Straube and M. Tamer Ozsu, *Queries and query processing in object-oriented database systems*, ACM Transactions on Information Systems, 8 (4), pp 387-430, 1990

[Stroustrup '91] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, I.S.B.N. 0-201-53992-6, 1991

[Suciu '96] Dan Suciu, *Query Decomposition and View Maintenance for Query Languages for Unstructured Data*, VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India, pp 227-238, 1996

[Sun '95] Sun, *"The Java Programming Language (white paper)*, Sun MicroSystems Inc, 1995

[Tompa '89] Frank WM. Tompa, *A data model for flexible hypertext database systems*, ACM Transactions on Information Systems, 7 (1), pp 85-100, 1989

[Vermeer '95] Mark W. W. Vermeer and Peter M. G. Apers, *Object-Oriented Views of Relational Databases Incorporating Behaviour*, Database Systems for Advanced Applications '95, Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), Singapore, pp 26-35, 1995

[Walter '93] K. S. Walter and M. J. O'Neil, *BG-BASE: Software for botanical gardens and arboreta*, The Public Garden, 34, pp 21-22, 1993

[Watters '90] Carolyn Watters and Michael A. Shepherd, *A Transient Hypergraph-Based Model for Data Access*, ACM Transactions on Information Systems, 8 (2), pp 77-102, 1990

[Webster '01] Webster, *Webster Online Dictionary*, http://www.webster.com/, 2001

[Weiser '89] Stephen P. Weiser and Frederick H. Lochovsky, *OZ+: An Object-Oriented Database System*, in *Object-Oriented Concepts, Databases, and Applications*, pp 309-337, ACM Press, 1989

[White '93] R. J. White, R. Allkin and P. K. Winfield, *Systematic Databases: The BAOBAB Design and the ALICE system*, Advances in computer methods for systematic biology: Artificial Intelligence, databases, computer vision, 1993

[Wieringa '94] R. Wieringa, W. de Jonge and P. Spruit, *Roles and dynamic subclasses: a modal logic approach*, Proceedings of the European Conference on Object-Oriented Programming, Bologna, Italy, pp 32-59, 1994

[Wong '97] Raymond K. Wong, H. Lewis Chau and Frederick H. Lochovsky, *A Data Model and Semantics of Objects with Dynamic Roles*, Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham U.K., pp 402-411, 1997

[Wood '90] P. T. Wood, *Graph Views and Recursive Query Languages*, BNCOD 8, University of York, UK, pp 124-141, 1990

[Zhong '96] Yang Zhong, Sunwon Jung, Sakti Pramanik and John H. Beaman, *Data model and comparison and query methods for interacting classifications in a taxonomic database*, Taxon, 45, pp 223-241, 1996

[Zhuge '98] Yue Zhuge and Hector Garcia-Molina, *Graph Structured Views and Their Incremental Maintenance*, Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, pp 116-125, 1998

Appendix I BNF

This section shows the BNF of two of the languages supported by Prometheus: the query language (POOL, known internally as PQL, Prometheus Query Language), and PCL (Prometheus Constraint Language).

I.1 PQL BNF

PQLProgram := "select" SelectDefinition "from" FromDefinition ("where" WhereDefinition)? (XLinkDefinition)* ("follow" FollowDefinition)? ("order by" MemberList)? | CountQuery | ExistsDefinition

CountQuery := CountDefinition (Sign WhereValue)?

SelectDefinition := CountDefinition | (Distinct)? (UsingSynonyms)? (StarSelectField | DotSelectFieldWithAggregate ("," DotSelectFieldWithAggregate)*)

Distinct := "distinct"

CountDefinition := "count" "(" (StarSelectField | PQLProgram) ")"

ExistsDefinition := "exists" "(" PQLProgram() ")"

StarSelectField := "*"

DotSelectFieldWithAggregate := DotSelectFieldWithAggregateShallow | DotSelectFieldWithAggregateDeep | DotSelectField

DotSelectFieldWithAggregateShallow := "shallow aggregate" DotSelectField

DotSelectFieldWithAggregateDeep := "deep aggregate" DotSelectField

DotSelectField := SelectField ("." SelectField)*

SelectField := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

FromDefinition := FromTypeOrVariable ("," FromTypeOrVariable)*

FromTypeOrVariable := DotFromType ("as")? (FromVariable)? | FromVariable "in" DotFromType

DotFromType := FromType ("." FromType)*

FromType := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

FromVariable := <IDENTIFIER>

WhereDefinition := "(" WhereDefinition ")" (AndOrComparison)* (UsingSynonyms)? | Comparison (AndOrComparison)* (UsingSynonyms)?

UsingSynonyms := "using instance synonyms"

AndOrComparison := "or" OrWhereExpression | "and" AndWhereExpression

OrWhereExpression := WhereDefinition

AndWhereExpression := WhereDefinition

Comparison := DotWhereField (In "(" PQLProgram ")" | NotIn "(" PQLProgram ")" | Sign WhereValue) | CountDefinition Sign WhereValue | "(" PQLProgram ")" Sign WhereValue

DotWhereField := WhereField ("." WhereFieldWithModifier)*

WhereField := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

WherePathField := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

WhereFieldWithModifier := WherePathField (WhereFieldPlusModifier | WhereFieldStarModifier | WhereFieldQuestionModifier) | WhereField | "(" WherePathField ("." WherePathField)* ")" (WhereFieldPlusModifier | WhereFieldStarModifier | WhereFieldQuestionModifier)

WhereFieldPlusModifier := "+"

WhereFieldStarModifier := "*"

WhereFieldQuestionModifier := "?"

Sign := <SIGN>

In := "in"

NotIn := "not in"

WhereValue := WhereString | WhereNumber | WhereBoolean | WhereDate | WhereReference

WhereString := <STRING_LITERAL>

WhereNumber := <NUMBER_LITERAL>

WhereBoolean := <BOOLEAN_LITERAL>

WhereDate := <DATE_LITERAL>

WhereReference := <REFERENCE>

MemberList := OrderType "." OrderAttribute ("asc" | "desc")? (OrderType "." OrderAttribute ("asc" | "desc")?)*

OrderType := <IDENTIFIER>

OrderAttribute := <IDENTIFIER>

FollowDefinition := FollowVariable

FollowVariable := <IDENTIFIER>

XLinkDefinition := ("in context" | “xlink”) ("where" WhereDefinition)?

< **IDENTIFIER**: <LETTER> (<LETTER>|<DIGIT>)* > | < REFERENCE: "PtRef("
<STRING_LITERAL> ")" > | < #LETTER: ["a"-"z", "A"-"Z", "\u00a0"-"\u00ff"] > | < #DIGIT: ["0"-"9"] >

< **NUMBER_LITERAL** : (["0"-"9"])+ | (["0"-"9"])+ (["e","E"]) (["+","-"])? (["0"-"9"])+ | (["0"-"9"])+"." (["0"-"9"])* (["e","E"]) (["+","-"])? (["0"-"9"])+ | "." (["0"-"9"])* (["e","E"]) (["+","-"])? (["0"-"9"])+ >

< **BOOLEAN_LITERAL** : "true" | "false" >

< **DATE_LITERAL**:
"\"" <NUMBER_LITERAL> "/" <NUMBER_LITERAL> "/" <NUMBER_LITERAL> "\"" >

< **STRING_LITERAL**:
"\""
((~["\"","\\","\n","\r"])
)*
"\"" >

<**SIGN**: "=" | "!=" | "<=" | ">=" |"<" | ">" | "==" | "<>">

I.2 PCL BNF

PCLProgram := WhereDefinition (ImpliesDefinition)?

ImpliesDefinition := "implies" WhereDefinition

WhereDefinition := "(" WhereDefinition ")" (AndOrComparison)* | Comparison (AndOrComparison)* | Not

AndOrComparison := "or" OrWhereExpression | "and" AndWhereExpression

OrWhereExpression := WhereDefinition

AndWhereExpression := WhereDefinition

Comparison := DotWhereField Sign (WhereValue | SelfWhereDotNoModifier)

Not := "not" "(" WhereDefinition ")"

DotWhereField := ClassWhereDot | SelfWhereDot

ClassWhereDot := ClassName ("." WhereFieldWithModifier)*

SelfWhereDot := Self ("." WhereFieldWithModifier)*

SelfWhereDotNoModifier := Self ("." WhereField)*

Self := "Self"

ClassName := <IDENTIFIER>

WhereField := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

WherePathField := <IDENTIFIER> "[" <IDENTIFIER> "]" | <IDENTIFIER>

WhereFieldWithModifier :=WherePathField (WhereFieldPlusModifier | WhereFieldStarModifier | WhereFieldQuestionModifier) | WhereField | "(" WherePathField ("." WherePathField)* ")" (WhereFieldPlusModifier | WhereFieldStarModifier | WhereFieldQuestionModifier)

WhereFieldPlusModifier := "+"

WhereFieldStarModifier :="*"

WhereFieldQuestionModifier := "?"

Sign : = <SIGN>

WhereValue := WhereString | WhereNumber | WhereBoolean | WhereDate | WhereReference

WhereString := <STRING_LITERAL>

WhereNumber := <NUMBER_LITERAL>

WhereBoolean := <BOOLEAN_LITERAL>

WhereDate := <DATE_LITERAL>

WhereReference := <REFERENCE>

< **IDENTIFIER**: <LETTER> (<LETTER>|<DIGIT>)* >

< **REFERENCE**: "PtRef(" <STRING_LITERAL> ")" >

< #**LETTER**: ["a"-"z", "A"-"Z"] >

< #**DIGIT**: ["0"-"9"] >

< **NUMBER_LITERAL** : (["0"-"9"])+

| (["0"-"9"])+ (["e","E"]) (["+","-"])? (["0"-"9"])+

| (["0"-"9"])+"." (["0"-"9"])* (["e","E"]) (["+","-"])? (["0"-"9"])+

| "." (["0"-"9"])* (["e","E"]) (["+","-"])? (["0"-"9"])+ >

```
< BOOLEAN_LITERAL : "true"
 | "false" >
< DATE_LITERAL:
    "\"" <NUMBER_LITERAL> "/" <NUMBER_LITERAL> "/" <NUMBER_LITERAL> "\"" >
< STRING_LITERAL:
    "\""
    (   (~["\"","\\","\n","\r"])
     | ("\\"
        ( ["n","t","b","r","f","\\","'","\""]
        | ["0"-"7"] ( ["0"-"7"] )?
        | ["0"-"3"] ["0"-"7"] ["0"-"7"]
        )
      )
    )*
    "\"" >
<SIGN: "=" | "!=" | "<=" | ">=" |"<" | ">" | "==" | "<>">
```

As can be seen, PCL is a subset of PQL. This is due to the fact that PCL needs to implement path expression in order to check the validity of chains of objects starting from the object implementing the rule. As the constraint language is implemented on top of the database, it needs to implement the features necessary to manipulate the structures offered by the model. The best way to do so is to adapt (restrict) the existing query language.

When PCL rules are defined, they are automatically translated into PQL queries. These PQL queries are thereafter evaluated with the PCL rules are fired.

Appendix II Benchmark

This section presents the benchmark results of section 7. The typical taxonomic queries, the POET (standard object-oriented) schema, the Prometheus schema, and the OO7-inspired benchmark results are presented.

II.1 Typical taxonomic queries

A working taxonomist needs access to specific kinds of information. These requests, used often during the process of a revision can be proposed as canned queries. Some of these queries are straightforward and any database system would be able to answer them, given that they store the appropriate information. Here, POOL queries are compared to standard OQL queries. Although OQL is not the most advanced query language, it is a standard, and most commercial database systems adhere to it or to one of its subsets. The comparison will show the advantages in terms of expressiveness and ease of use of POOM/POOL against ODMG/OQL.

Note that when relationship objects have been used in the schema, they are assumed to be pointers in OQL queries for simplification. A vertical line in the margin identifies these queries. Using relationship objects in OQL would require many joins and would complicate reading and comparison of queries.

Example of typical queries:

- Who published a specific name?

  For example: find the taxonomist that published the name Apium.

  In standard OQL:

```
(Q1)  Select a from Name n, n.thePublication.theAuthor a where
      n.theEpithet.theName = "Apium"
```

  In POOL:

```
(Q2)  Select n.thePublication.theAuthor from Name n where
      n.theEpithet.theName = "Apium"
```

  or

```
(Q3)  Select a from Name n, n.thePublication.theAuthor a where
      n.theEpithet.theName = "Apium"
```

  Note that in Prometheus the direction of traversal is only required if there may be an ambiguity. The three relationships used in this query, thePublication, theEpithet, and theAuthor are not recursive relationships, therefore cannot be traversed in the wrong direction.

- What names have been published by a specific taxonomist?

For example: find the names that Linnaeus published

In standard OQL:

```
(Q4)  Select * from Name n where
   n.thePublication.theAuthor.surname = "Linnaeus"
```

In POOL:

```
(Q5)  Select * from Name n where
   n.thePublication.theAuthor.surname = "Linnaeus"
```

- What is the taxonomic type of a name?

For example: find the type of Apium.

In standard OQL:

```
(Q6)  Select n.LinkToType from Name n where
   n.theEpithet.theName = "Apium"
```

In POOL:

```
(Q7)  Select n.LinkToType from Name n where
   n.theEpithet.theName = "Apium"
```

- What specimens have been determined to a specific name?

For example: what specimens have been determined to *Apium graveolens*?

In standard OQL:

```
(Q8)  Select n.LinkToDetInv from Name n where
   n.theEpithet.theName = "Apium graveolens"
```

Note that LinkToDetInv is the inverse reference of the LinkToDet reference. This query therefore assumes that inverse references have been managed by the system (e.g. relationships in ODMG).

In POOL:

```
(Q9)  Select n.LinkToDet.origin from Name n where
   n.theEpithet.theName = "Apium graveolens"
```

Note that this query takes into account the fact that relationships in Prometheus can be traversed in all directions, as long as the traversal is not prohibited by the traversal flag (see section 4.4.3). The "origin" keyword in the query specifies that the traversal of the relationship must be performed from destination to source. This query shows that the availability of relationships as first-class concepts provides easier navigation and does not require manual management of inverse references. This query can also be written from the point of view of specimens:

```
(Q10) Select s from Specimen s where
      s.LinkToDet.theEpithet.theName = "Apium graveolens"
```

Here, the keyword "destination" is not necessary, as the relationship is not recursive.

As can be seen in the previous example, POOL is very close to OQL. This allows an easy port from an existing ODMG compliant database to Prometheus.

Other queries are less straightforward. These queries involve either the recursive exploration of a classification or many classifications, or the extraction of specific classifications (or classifications given a context). The examples given next cannot be computed in other existing taxonomic databases. For example:

- What is a specific taxonomist's view/classification?
  For example: what is the classification created by Linnaeus and published in "Flora Europea"?
  In OQL:
  As ODMG and OQL do not manage classifications, this query cannot be simply expressed in the context of graphs. OQL could only extract nodes that appear in a classification. However, if the taxonomic model shown in section 2.3 were implemented as it is in an ODMG database, the following query would extract named classifications:

```
(Q11) Select ct, ct.CalculatedName from CT ct where
      ct.thePublication.thePublication = "Flora Europea" and
      ct.theAuthor.surname = "Linnaeus"
```

  This query returns tuples containing CTs (classification elements), and NTs (names). It can be seen that this approach has the disadvantage of not returning Specimen objects, as they are not CTs.

  In POOL:

```
(Q12) Select t.origin, t, t.destination from theCircumscription
      t where t.theCircPublication.thePublication = "Flora
      Europea" and t.theCircAuthor.surname = "Linnaeus"
```

  This query extracts all relationships and all nodes involved in the definition of Linnaeus' classification published in "Flora Europea". The selection is performed on the basis of relationships and their attributes (a specific context is selected). The resulting collection contains these objects arranged in triples representing atomic portions (two nodes and an arc) of the classification.

- What are the specimens defined in the system that are not used in a particular classification?
  For example: what are the specimens that do not appear in the classification created by Linnaeus and published in "Flora Europea"?

In OQL:

ODMG/OQL, as most object-oriented models/query languages, do not manage classifications, especially overlapping classifications. This type of query is not expressible in such an environment. However, on a POOM-like schema, the OQL query would be:

```
(Q13) select distinct s1 from theCircumscription t, Specimen
   s1, Specimen s2 where s1=t.destination and
   t.theCicPublication.thePublication = "Flora Europea" and
   t.theCircAuthor.surname = "Linnaeus" and s2 != s1
```

In POOL:

```
(Q14) Select * from Specimen s not in (Select
   t.destination[Specimen] from theCircumscription t where
   t.theCicPublication.thePublication = "Flora Europea" and
   t.theCircAuthor.surname = "Linnaeus")
```

Note that the selective downcast operator has been used to select only specimens in the sub-query.

- What specimens appear in the circumscription of a name in a specific classification, and also in the circumscription of another in another classification?

  For example: what specimens appear in the circumscription of Apium according to Linnaeus and also in the circumscription of Heliosciadium according to Watson?

  Note that the two groups selected do not contain specimens directly, but rather contain other groups that may contain specimens (e.g. Species) or that may contain other groups that contain specimens (e.g. Sections).

  In OQL:

  As this query is recursive and as OQL does not support recursion, it cannot be expressed.

  In POOL:

```
(Q15) Select s from Specimen s where s in (select
   t1.destination[Specimen] from theCircumscription t1 where
   t1.theCircAuthor.surname = "Linnaeus" and
   t1.(theCircumscription.origin)*.theEpithet.theName = "Apium"
   in context) and s in (select t2.destination[Specimen] from
   theCircumscription t2 where t2.theCircAuthor.surname =
   "Watson" and
   t2.(theCircumscription.origin)*.theEpithet.theName =
   "Heliosciadium" in context)
```

The `in context` operator is used here in order to keep the traversal of the graph of objects in the same classification as the first traversed relationship. Note also the use of the selective downcast operator to select objects whose type is Circumscription to only Specimen.

- What specimens appear in the circumscription of a name in a specific classification, but not in the circumscription of another in another classification?
  For example: what specimens appear in the circumscription of Apium according to Linnaeus but not in the circumscription of Heliosciadium according to Watson?
  In POOL:
  This query is similar to (Q15).

```
(Q16) Select s from Specimen s where s in (select
   t1.destination[Specimen] from theCircumscription t1 where
   t1.theCircAuthor.surname = "Linnaeus" and
   t1.(theCircumscription.origin)*.theEpithet.theName = "Apium"
   in context) and s not in (select t2.destination[Specimen]
   from theCircumscription t2 where t2.theCircAuthor.surname =
   "Watson" and
   t2.(theCircumscription.origin)*.theEpithet.theName =
   "Heliosciadium" in context)
```

- What are the synonyms based on types of a specific group in a specific classification?
  For example: which names at rank Genus are synonyms of Apium (also at rank Genus) according to Linnaeus, based on the occurrence of taxonomic types?
  In OQL:
  Absence of recursion in OQL prevents this kind of query.

  In POOL:

```
(Q17) Select * from Name n where
   n.(LinkToType[Name])*.LinkToType[Specimen] in (select
   t1.destination[Specimen] from theCircumscription t1 where
   t1.theCircAuthor.surname = "Linnaeus" and
   t1.(theCircumscription.origin)*.theEpithettheName= "Apium"
   in context) and n.theRank.theName = "Genus"
```

The extensive use of the selective downcast operator simplifies the query and allows recursive querying on subtypes. Without that operator, the query would have been more complex, as shown in query (Q16). In effect, POOL extracts all specimens that belong to a specific classification (sub-query) and then recurses down the type hierarchy until Specimen objects are found (repetition of traversal of Name types until Specimen types are found), and only retains those that appear in the sub-query classification.

- What are the synonyms based on specimens to a specific group in a specific classification?
  For example: which names at rank Genus are synonyms of Apium (also at rank Genus) according to Linnaeus based on specimens?
  In OQL:
  Absence of recursion in OQL forbids this kind of query.

  In POOL:

```
(Q18) Select * from Name n where
   n.(theCircumscription[Name])*.theCircumscrition[Specimen] in
   (select t1.destination[Specimen] from theCircumscription t1
   where t1.theCircAuthor.surname = "Linnaeus" and
   t1.(theCircumscription.origin)*.theEpithettheName= "Apium"
   in context) and n.theRank.theName = "Genus" in context
```

  The `in context` operator is used on the main query because it is essential to stay in one specific classification once it has been entered. It would not make sense to switch between classifications at any time.

II.2 Performance test

II.2.1 POET schema

The following schema (Figure 47) has been implemented in POET to serve as a reference point benchmark.

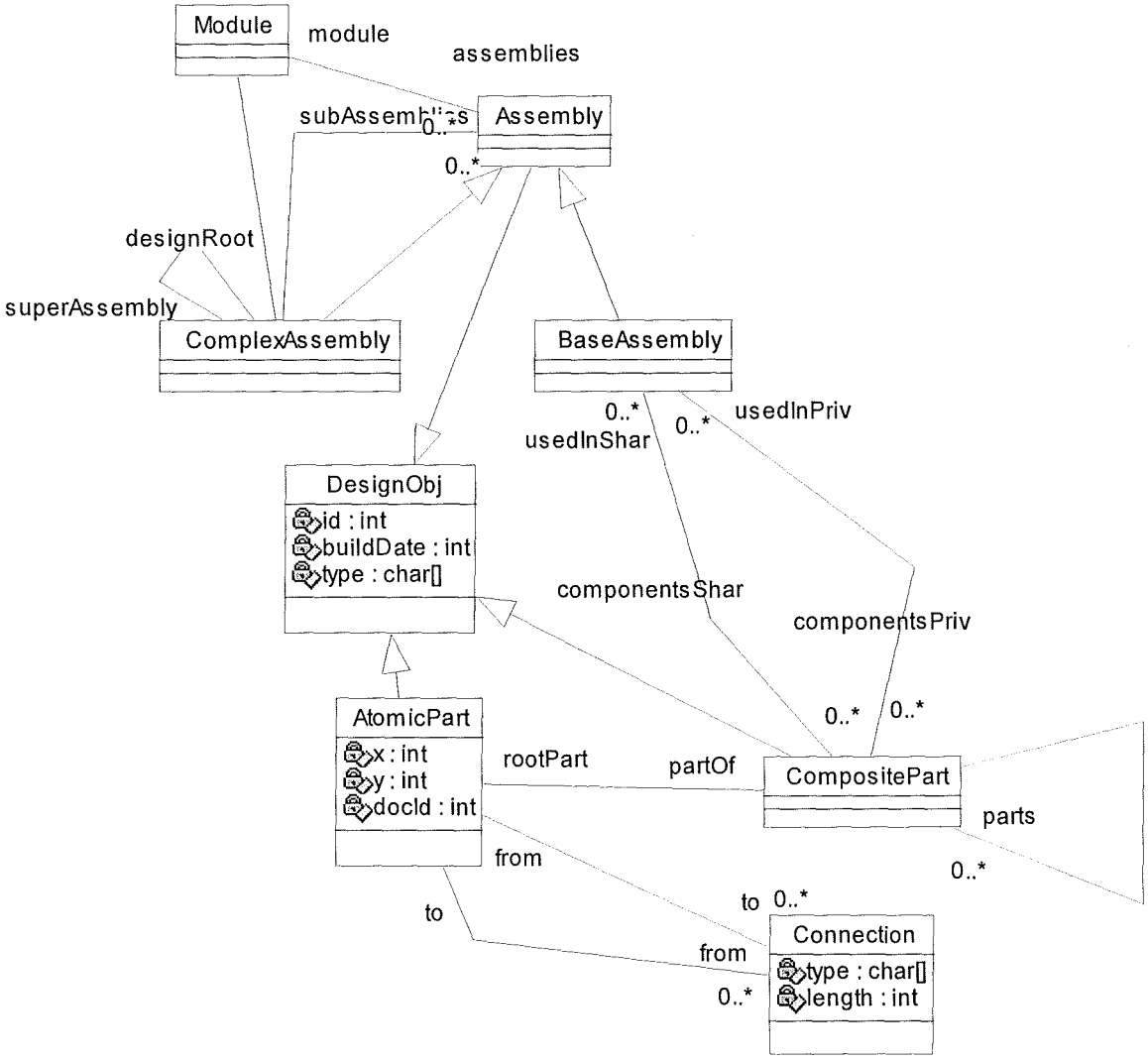


Figure 47: POET benchmark schema

Note on this schema: the semantics of relationships have not been represented. For example, `componentsPriv` should be an aggregation relationship with non-sharing behaviour. However, as these semantics are not expressible in POET, as in most object-oriented databases, they are implemented as pointers and it is left to the implementation code to maintain consistency.

II.2.2 Prometheus Schema

The following schema (Figure 48) has been implemented using Prometheus' features in order to test the performance of the system.

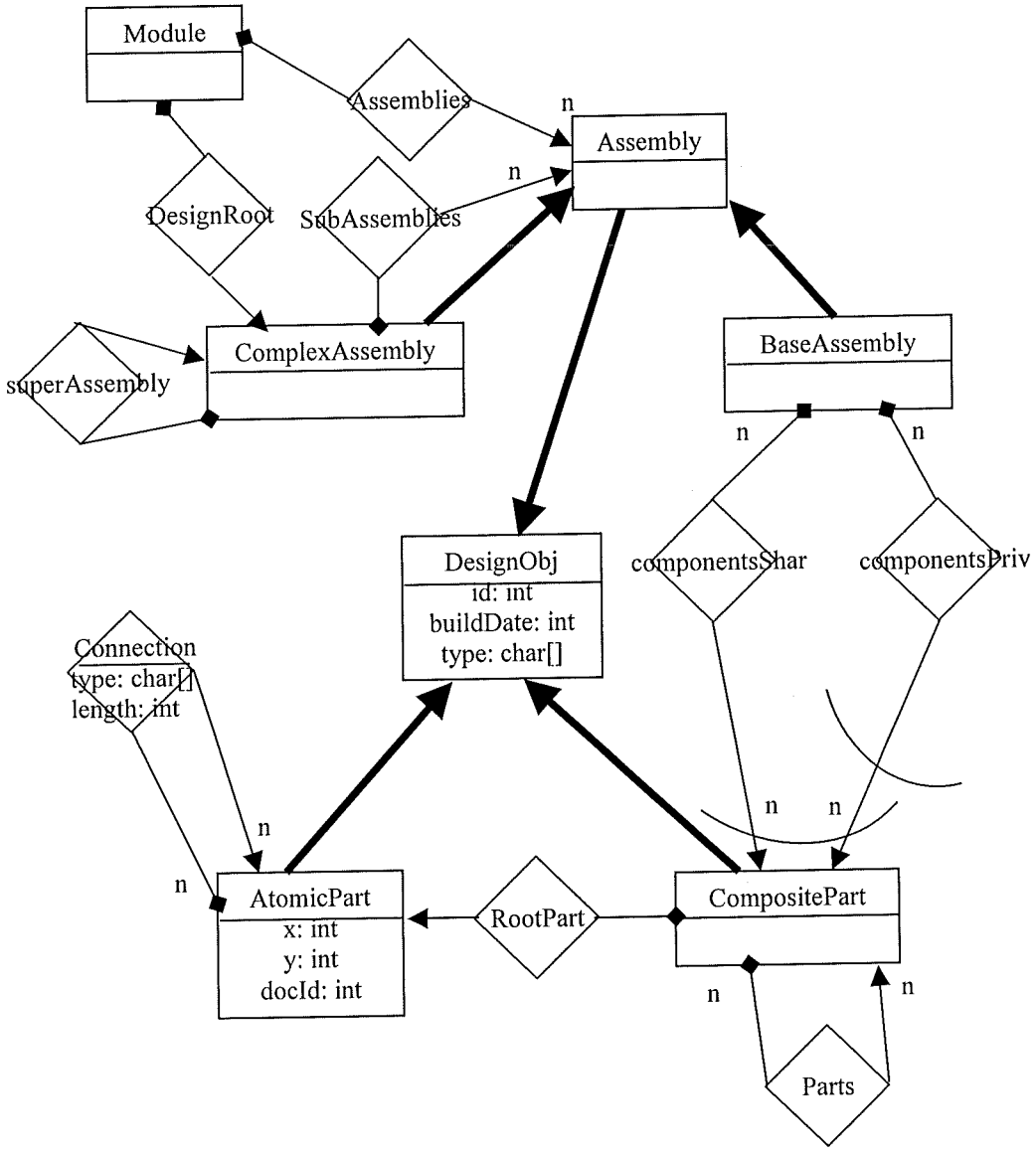
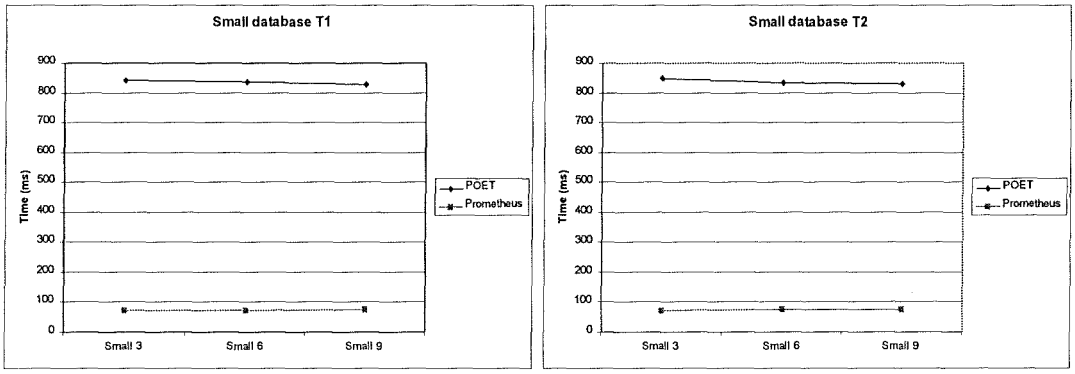


Figure 48: Prometheus benchmark schema

II.2.3 Results

Tests T1 & T2

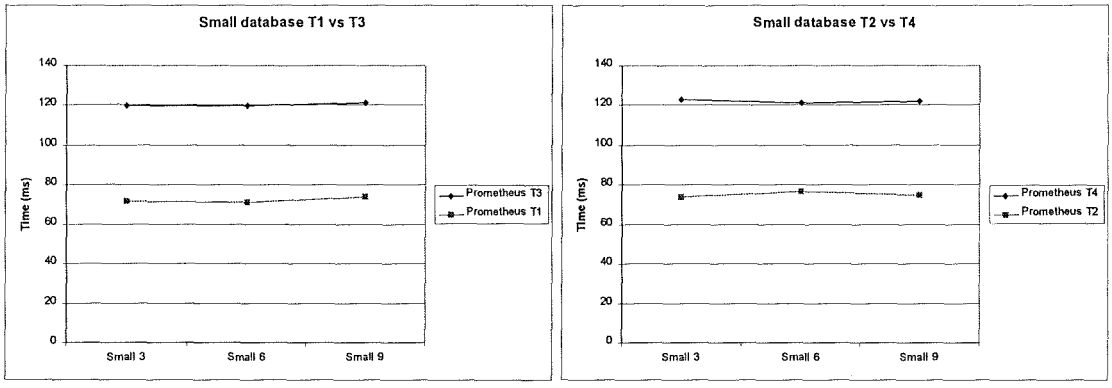
These two tests show the time necessary to load an object in the case of a new empty transaction with empty object cache (cold) and in the case of an existing transaction (hot). The graphs show that the time necessary to load an atomic part from the database is less expensive with Prometheus than with POET, whether the database is hot or cold. This surprising result (Prometheus uses POET, therefore should not be faster) can be explained in part by the fact that objects in Prometheus are generally more numerous (relationships are explicitly represented), but smaller than in POET (see II.2.4.4).



The comparison T1 versus T2 also shows that the object cache provided by POET is not efficient and does not reduces the cost of performing the operation. In fact, as POET is a multi-user database system, each time an object is resolved (which includes each time its attributes are accessed), POET verifies the locking of these objects therefore goes to the database file and reads this information.

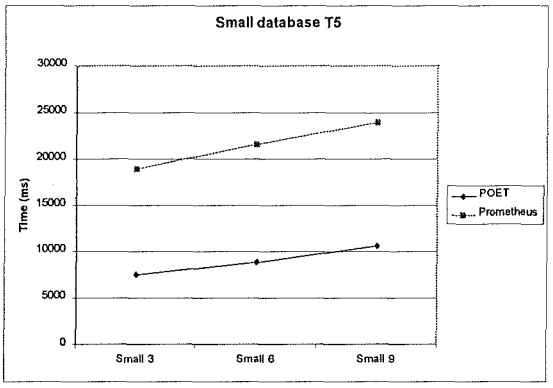
Tests T3 & T4

Tests T3 and T4 allow the comparison between the cost of loading "normal" objects and the cost of loading relationship objects. The graphs show that loading a relationship is constantly slower (about 50 ms) than loading a "normal" object (also with Prometheus). This is in part due to the size of objects and the existence of indexes for relationship objects.



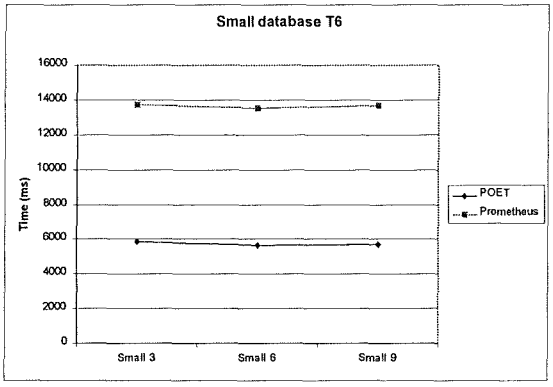
Test T5

Test T5 explores two hierarchies of objects: the assembly hierarchy and the parts hierarchy. It shows the cost of traversing simple graphs (assembly hierarchy) and complex graphs (the highly connected part hierarchy). The graph shows that because Prometheus uses relationship objects instead of references, the traversal of hierarchies is constantly more expensive than in POET.



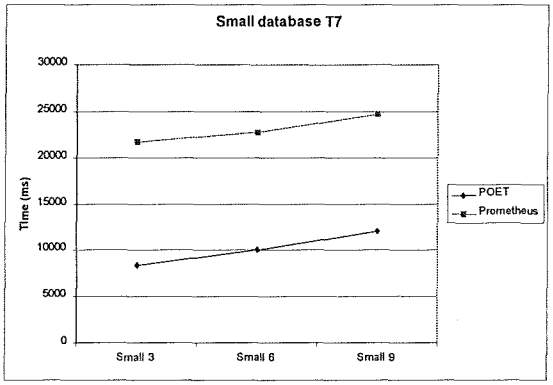
Test T6

Like T5, T6 performs a graph traversal. However, instead of traversing both the assembly and the parts graphs, it only traverses the assembly hierarchy and retrieves the root part of the last base assembly object. The test shows how expensive the traversal of the complex part graph is (over 30% of the whole task in the case of Prometheus) and it shows that traversing relationships adds a constant cost to the operation.



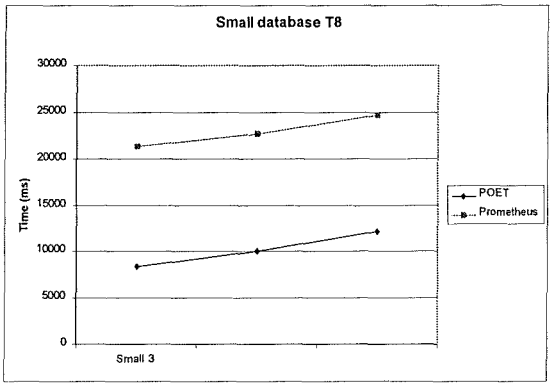
Test T7

In addition to a graph traversal as in T5, T7 performs updates on the objects it encounters (one update per composite part). This shows the cost of updating many objects. Once again, the graph shows that Prometheus is constantly more expensive.



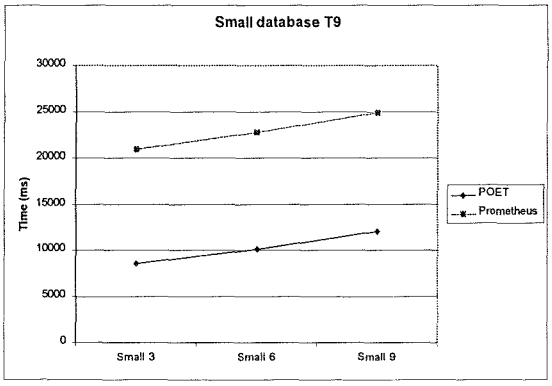
Test T8

Test T8 performs the same operations than T7, but instead of updating one atomic part per composite part, every atomic part encountered is updated. The graph shows that the price of updating a large number of atomic parts is not noticeably more expensive than updating only one atomic part per composite part.



Test T9

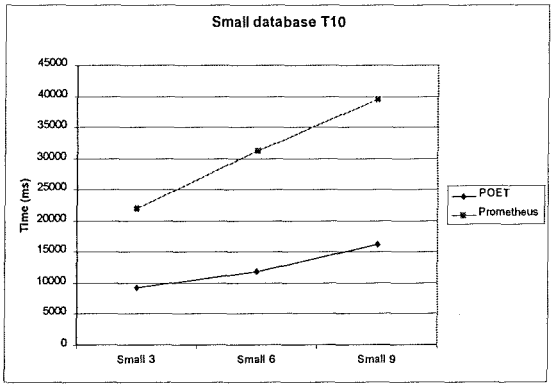
Test T9 goes further than T8 and updates every encountered atomic part four times. The graph shows that the cost of four updates to the same object is not more expensive than a single update. This is due to the fact that objects are only committed once at the end of the transaction.



Note that tests T5 to T9 show a parallel increase in the cost of traversal and updates between POET and Prometheus. This shows a similar ability to scale for access and updates.

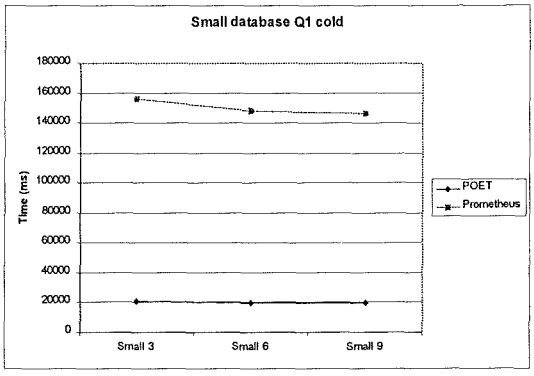
Test T10

T10 compares the cost of updating a "fake" relationship object in POET (an object used as a relationship) and a relationship object in Prometheus. The graph shows that updating relationships in Prometheus is more expensive than "relationship" objects in POET and that the costs increase generated by updating relationships goes up faster than the cost of updating "fake" relationships. This may prove a problem when Prometheus needs to scale up.

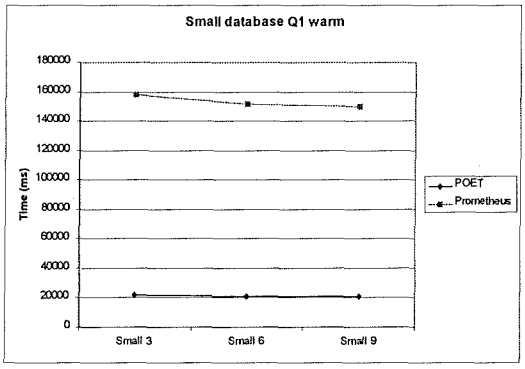


Test Q1

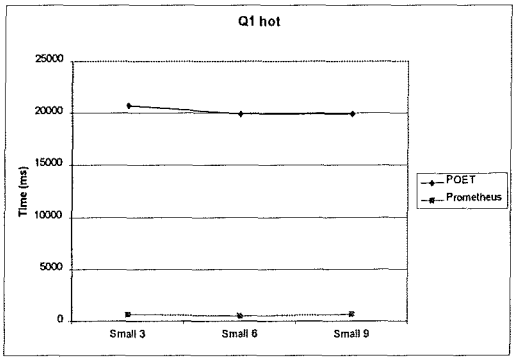
The purpose of Q1 is to compare the speed of retrieval of an atomic part from the database. The graph shows that Prometheus is significantly slower than POET for the evaluation of simple path expressions (the traversal of a single attribute). The cost of the operation is nearly constant across the tests in both Prometheus and POET because the number of connections has no influence on Prometheus' resolution of objects and POET does not resolve objects while it evaluates queries (see II.2.4.6).



Note that with POET (unlike Prometheus), it is necessary to commit the transaction in which objects have been created before they can be queried.



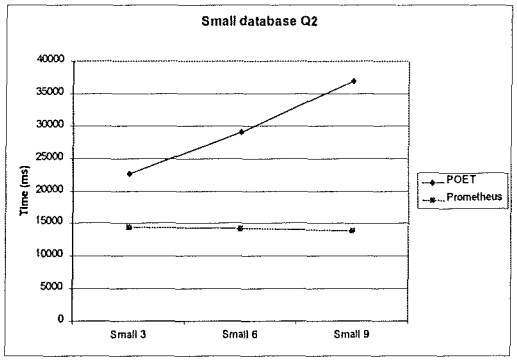
The warm run shows that POET is not affected by the objects present in the Transaction (POET does not query on transient objects). Prometheus is only slightly affected by those objects, as the results are only a few seconds slower than a cold run.



The hot run shows the advantages of the built-in Prometheus indexes. As a query is run the path that was traversed and the objects contained in that path are indexed, which allows a faster later traversal of the same path (possible as part of a different query).

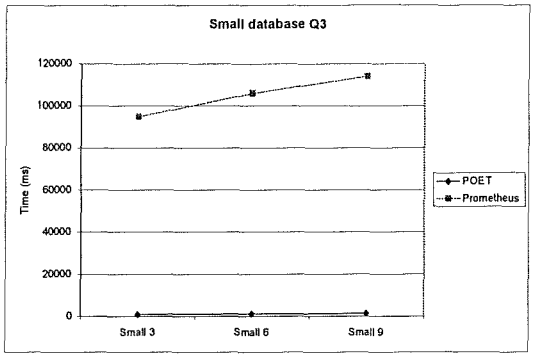
Test Q2

Test Q2 measures the time necessary to scan the atomic parts collection (extent). Combined with Q1, Q2 shows the cost of exploring a very simple path expression independently of the time necessary to scan a collection. It can be seen that Prometheus is slower than POET for a query, but faster for scanning a collection. Combined with T1, it can be deduced that this is partly because of the size of objects in Prometheus is generally smaller than in POET (see results of test T1 and section II.2.4.4). The fact that the cost of object resolution increases in a linear manner with POET whereas it is constant in Prometheus can be explained by the fact that when the atomic parts are resolved, the number of connections the parts are involved in has an impact on the cost of the operation in POET (in POET close objects are resolved as well).



Test Q3

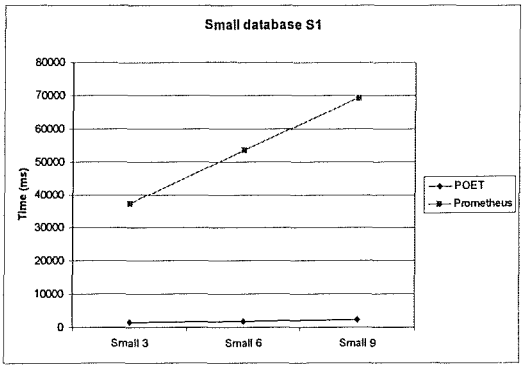
Test Q3 shows the difference in cost between writing a purpose driven recursive program with POET and a generic recursive query with Prometheus. The test shows that what is gained in simplicity for expressing queries (the POET program can be expressed with only one Prometheus query) is lost in performance, as processing queries recursively makes it harder to exploit specific features in the data (e.g. the OO7 data is highly connected in the area where test Q3 is made, which makes evaluation of recursive queries inefficient).



Note that although the graph does not show it clearly because of the difference in scale between POET and Prometheus results, the increase in time from one test to the next is the same (about 10%).

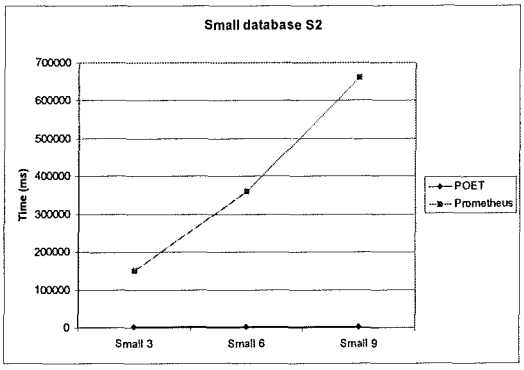
Test S1

Test S1 shows the cost of creating a large number of objects, including a significant part of relationships. It can be seen that Prometheus is much slower to create these objects than POET and the cost increases faster than similar operations in POET. This is because Prometheus uses relationships objects, therefore not only manages a larger number of objects, but these objects are more expensive to manage (see results of test T10).



Test S2

Test S2 allows the comparison of the cost of creating objects to the cost of deleting objects. The graph shows that deleting an object is less expensive than creating one in POET, whereas it is much more expensive to delete an object than create it in Prometheus. This is partially due to the fact that events are detected and fired when objects are deleted. This is also due to the fact that Prometheus has to maintain indexes that need to be updated after deletion. As Prometheus' indexes are objects, updating an index requires fetching additional objects and updating them. This adds to the cost of deletion.



II.2.4 Analysis

II.2.4.1 The package problem

As a design choice in this system, the ability to port the mechanisms provided to new underlying database models was selected. This means that whatever mechanisms are implemented, they need to function as a user application, not as a part of the database. This has two major consequences: the non-availability of indexes that are handled by the underlying database system and are specific to such systems, and the necessity to instantiate any object that is accessed at any time. This can be seen in tests such as (Q1), where the absence of object indexes and the necessity to resolve every traversed object handicaps Prometheus versus POET.

The availability of system indexes to user applications is a consequence of the design of the underlying database system. Some databases provide access to these indexes and others (as it is the case of the database system chosen for this project) do not. The inability to access indexes implies that it is not possible for user applications (Prometheus is one, as it is simply a package added to a given system) to gain access to fast means of finding specific objects in the system. This means that two approaches can be considered: the implementation of surrogate indexes by the user application and the necessity to perform a full access (including object and pointer resolutions) each time an object is handled. The first solution is only applicable if the underlying system provides extent event notification, i.e. only if it is possible to know that an object has been created, changed, or deleted so that the system can take the necessary action to maintain indexes. In addition, as they are additional (redundant) indexes, they apply an overhead to all system operations, as possibly two distinct, but equivalent indexes, may be maintained concurrently by two parts of the system. The second approach has the inconvenience of forcing the system to load fully and to resolve pointers of objects that are accessed, even though they might not be really interesting for the current task. For example, a query that searches for a specific object with a specific attribute value would have to load all the objects that might be of interest in order to find only one (i.e. a full scan of a class' instances in order to find only one instance).

The necessity to instantiate all objects accessed by the system is an important drawback of the package approach. As object resolution and pointer resolution is an expensive task, especially for large objects, the system should try to perform them as little as possible. However, being a package independent of the underlying database implies that any object access passes through the underlying system and fully resolves the accessed objects. This overhead is especially important in queries, where optimisations can reduce the search space (e.g. by evaluating queries in reverse because it is known that targeted sets of objects are smaller than targeting sets). If looking for a single object in a collection (e.g. extent) implies a complete scan of the collection and the instantiation of all the objects contained therein, the query will be extremely expensive compared to the equivalent query without instantiation.

Furthermore, multi-user access implies checking regularly in the database the locking status of the objects accessed. Indeed, when an object is accessed during a read-only transaction, it can be accessed by many other systems that access it in read only too. Therefore, even when objects stay in memory after the end of a transaction (as may happen as some other objects might reference them between transactions), accesses to the database are necessary occasionally, which slows down access to objects. Likewise, between two transactions in a single application, objects may have been changed by another application, therefore need to be checked.

Prometheus is a package implemented on top of an existing object-oriented database. As that database does not give access to its internal indexes (they only exist in the innermost parts of the system), Prometheus needs to instantiate all accessed objects at a given moment. This implies slow querying.

Integrating Prometheus more deeply in an existing database system may provide a better approach from the performance point of view. This would allow access to indexing structures, it would avoid the necessity to instantiate all the objects that are handled, and it would allow the implementation of more efficient indexes (that do not need to manipulate persistent objects managed by the system itself).

II.2.4.2 Indexing

The indexing technique used in order to speed up query evaluation has an overhead on the functioning of the system. Indeed, it requires that each time an object that might invalidate the result of a query is changed, created, or deleted, the index of that query should be discarded and fully re-evaluated the next time it is requested by the user. (S1), (S2), (T7), (T8), and (T9) show this phenomenon: the creation and deletion of relationships require the indexing or de-indexing of relationships, and possibly invalidation of query indexes. The update of "normal" objects in Prometheus also requires more resources than in POET due to changes made to indexes (e.g. path indexes).

The independence of classified objects from the classification also has an impact on the performance of the system. An argued (but debatable) advantage of object-oriented systems over relational systems is their ability to follow pointers economically. Indeed, following a pointer only implies resolving OIDs and possibly loading objects from disk to memory. Although this is expensive when large portions of instances of a single class are accessed (the whole class is not necessarily loaded in memory as a page or a table might be in relational systems), this is economical in some circumstances, for example compared to relational joins. In Prometheus however, as first-class relationships are supported, traversing a graph of objects is not a straightforward traversal of a series of pointers. The economical cost of traversing pointers versus joins is true in the case of Prometheus. Traversing a relationship implies at least one join (to find the relationships that start in the right object) and pointer following (to go to the next object via the destination attribute of the relationship). One could argue that this overhead could be avoided by maintaining reverse pointers in objects that are targeted by relationships, but this option is not allowed by one of the requirements of the system: classified objects (therefore targeted objects) should not be designed in order to be classified, i.e. if these objects do not maintain reverse pointers it is not possible to modify them.

A naive evaluation of this process would imply two scans of (possibly) large extents (the starting object class' extent and the relationship class' extent) and one pointer following. In Prometheus, an index has been created in order to avoid unnecessary scans of relationship extents (see 6.1.5.2). The advantage of these indexes is partially shown in (Q2) where reverse references are managed via indexes in Prometheus, whereas they are managed as duplicate references in POET: in Prometheus, the resolution of "normal" objects does not involve the resolution of connected objects/relationships. However, looking up a hash table is more expensive that simply following a pointer. It also depends on the implementation of the stored structures and their scalability. This can be seen in (T5) where the cost of traversal is higher than traversing references.

II.2.4.3 Active system

The implementation of active rules and constraints in Prometheus also participates in the performance overhead of the system. Indeed, active system implies event detection, execution of rules, and actions taken as a consequence of the execution of these rules. Active database systems are notoriously slower than non-active systems. In Prometheus, tests (S1) and (S2) show the cost of these features compared to the cost of simply creating and deleting objects in POET (note that although rule evaluation has been switched off, event notification is still activated as it is fundamental to Prometheus' working).

The modification of an object in Prometheus results in the system detecting the event, then checking whether it is associated with any rule. The search for the rules can be optimised by clustering rules that depend on the same events together. However, searching for these rules generates a non-negligible load on the system that would not exist if it were a passive system. When the necessary rules have been found, Prometheus needs to evaluate the event part of the rule in order to check whether the action (or constraint in Prometheus) part of the rule should be executed. Then the action/constraint part of the rule can be executed or stored in the rule scheduler for later execution, depending on the type of the rule (see section 5.2.2).

When this process occurs each time an object is created, modified, or deleted, the load becomes significant. Other database systems that have been turned into an active system have encountered the same problem (e.g. Adam).

II.2.4.4 The philosophy

Prometheus manages relationships explicitly in order to offer a better modelling ability and more features. As was explained in section 4.8.1, all relationships should be represented by explicit relationships with Prometheus. For this benchmark, all relationships have been implemented in this way. The consequence is that complex "large" objects are broken down into smaller objects (as in a graph-based database), some representing relationships, others representing attributes references via relationships. Therefore, in a Prometheus database, the objects are likely to be a lot smaller than their equivalent in other systems. The cost of loading such objects therefore also goes down, as if the task is not interested in attributes, they are not loaded as part of the necessary object/reference resolution. See tasks (T1) and (T2) for example.

However, when the objects that are referenced by the object of interest are to be loaded, the cost of resolving the web of objects is higher than resolving a few references because this requires loading far more objects than in the case of common databases. Indeed, as well as the object of interest and its attribute objects, relationships linking them need to be loaded and resolved in order to be able to make the connection. (T5) shows that these external relationships increase the cost of traversing graphs of objects, as additional objects must be resolved during the traversal.

II.2.4.5 Programming language

The programming language chosen for this project is Java. This language has the advantage of not requiring recompilation to run on new systems. As taxonomists use many different platforms (PC, Mac, Sun) and different OS (Windows, Mac OS, SunOS, Linux), this was thought a good choice.

However, this language brings its own problems. The main problem that has arisen during the project is its speed. The speed at which Java runs depends on many design choices: it is interpreted, therefore the executed code is not native machine code and needs to be translated first (on the fly or not, depending on the use of runtime compiling technology); it supports garbage collecting, which simplifies development but increases object creation time (therefore instantiation time of persistent objects) and deletion time. There are Java compilers but they were not used in this project to facilitate debugging.

The underlying database query language and persistent mechanism are implemented in C++. It is well known that C++ performs better than Java (at the price of less straightforward portability). POET only provides a Java API that manages transactions and forwards all other calls to the C++ kernel. Comparing Prometheus to POET implies taking into account these differences and limitations. This difference explains partly the performance difference between Prometheus and POET when queries are run (Q1), but is hard to quantify precisely as it is part of a larger process (query evaluation, which implies additional object instantiation on Prometheus).

II.2.4.6 Transient vs. Persistent querying

It appears that object resolution in POET (the underlying database) is an extremely expensive process, especially when performed on large objects. POET overcomes this problem in queries by only performing queries on persistent indexes, i.e. on data that has been committed to the database. This provides a means to access fast efficient search structures (persistent indexes) and avoids object instantiation and object resolution. This can be seen in test (Q1) warm, where Prometheus does not only searches on disk, but also searches in memory for transient objects (or for the latest version of objects, which POET would ignore).

Although querying on persistent indexes is a sensible approach in some rare cases, on most cases this limits the ability to perform transient queries and therefore the ability to implement a constraint/rule mechanism. Indeed, rules should be evaluated when they need to be evaluated (after an event), i.e. they might need to be evaluated before the end of the transaction, therefore on the transient state of the database, not on its persistent state. Prometheus' queries do not take this approach and transient objects can be queried as well as persistent objects. This choice imposes an important overhead on Prometheus' query engine: efficient persistent indexes are useless in most queries, and object resolution slows down the speed at which objects can be traversed.

In addition, this transient querying requires access to the list of objects that have been modified and to their state at various stages of the transaction (e.g. the latest state or the state at the time a rule has been triggered). Not all database systems provide access to this information and the underlying database used in this project does not. In order to have access to these structures, a technique that can be called "class hijacking" has been devised. It exploits a security weakness in Java: Java reads classes in the order they are found in the class path, therefore if a class with the same type as another is found first in the classpath, it will be used instead of the other. A new transaction class has been created with the same interface and behaviour as the underlying database's transaction classes, but with the ability to provide to user applications information about the state of objects in the transaction.

However, this technique has its own performance drawback: when transient objects are queried, their appropriate state must be found. This implies a scan of all objects that have been modified in the transaction and the selection of the state of the appropriate object. This search, performed on all objects that reside in memory, which is very expensive, especially in long transactions where an important number of objects are accessed and modified and especially because the list of modified objects is not ordered (this list is managed by the underlying database). The solution could be the creation of transient indexes that manage objects in transactions.

II.2.4.7 Recursive querying

Recursive querying is an essential feature of a system designed to support taxonomic work. Indeed, in the view of taxonomy that Prometheus supports (but to which it is not limited), specimens are the most important part of a classification and the only objective piece of information available to taxonomists that are not the authors of a specific classification. Most of the processes involved in the taxonomic processes that must be supported by the system imply a recursive exploration of the classifications: the derivation of names, the comparison of classification or branches of classifications, and the extraction of synonymy information. (Q3) shows the price of these features.

The kind of recursive querying that Prometheus supports includes the exploration of (possibly cyclic) graphs of unknown depth. When the depth of the graphs is not known, it is hard to optimise the query evaluation (more specifically the evaluation of the recursive part of the query), as the objects that must be traversed is unknown. In addition, Prometheus does not provide many optimisation features (see section 6.1.5).

In Prometheus, recursive queries are evaluated using recursive calls to an evaluation function. This has the advantage of fitting well with the way recursive queries must be treated, but it has the disadvantage of requiring the intensive use of the stack. Indeed, each time a relationship or a reference is followed in a recursive statement, the state of the system is stored on the stack to allow returning to that point in case of unsuccessful evaluation and exploration of an alternative path. This use of the stack is expensive memory-wise (many objects must be copied on the stack) and performance-wise (many objects must be copied, therefore the garbage collector is used intensively).

This evaluation combined with the package approach to the implementation of the system and the transient programming necessary to support an active system means that many objects must be instantiated and resolved unnecessarily (in case of unsuccessful evaluation of a path), which is time consuming.

Additional index could be used in order to avoid recursivity in the evaluation of queries. For example, indexes of paths could be devised so that when a query requires the repeated traversal of a relationship from a node, all nodes reachable from the stating node could be found immediately, without the costly exploration of the graph. In effect, this index would "flatten" the graph into a set of indexed paths.