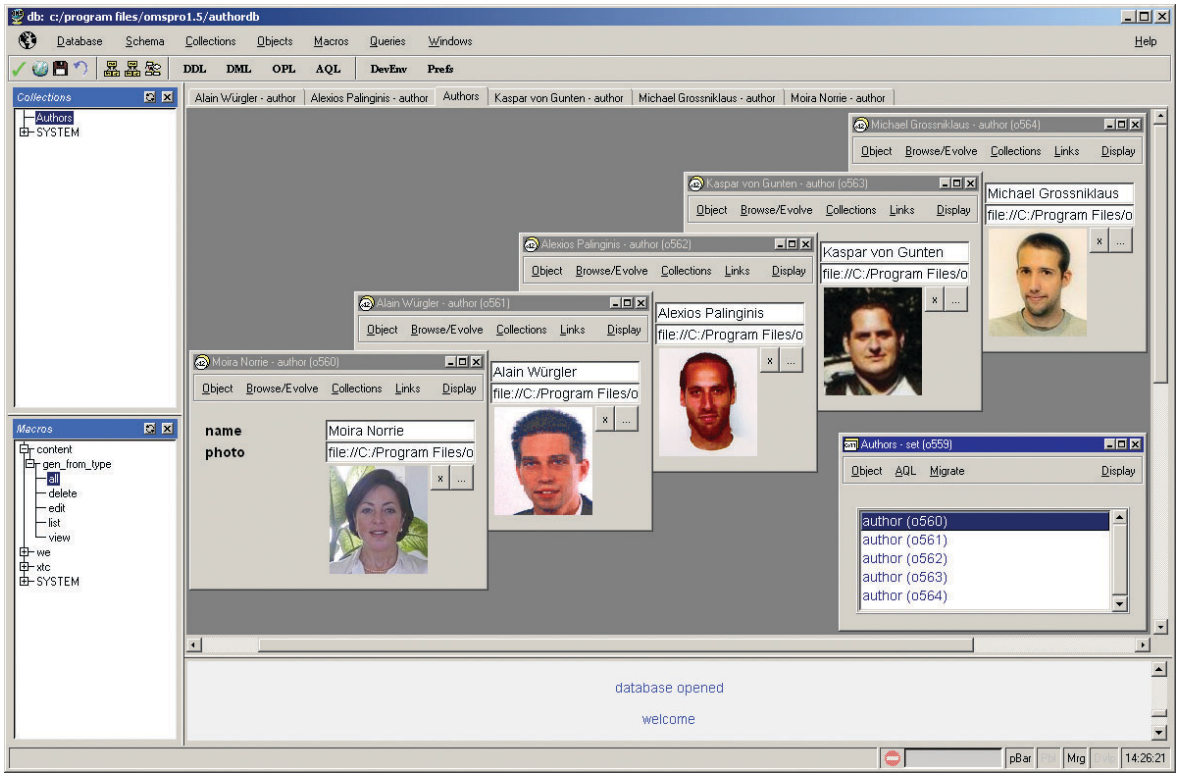
OMS Pro 2.0

Introductory Tutorial



Moira Norrie Alain Würgler Alexios Palinginis
Kaspar von Gunten Michael Grossniklaus

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Institute for Information Systems

globis

Preface

This document is intended as an introduction to the OMS object-oriented database management system. It provides an overview of the underlying OM generic data model and the specific OMS type model together with a tutorial on how to use the OMS system. Such a document cannot provide full details of the model, the query language, the Prolog programming language, in which object methods are specified, and all of the system features. However, we hope that it provides a good starting point for someone who wants to learn about the main features of the system and to start to use the system.

Zurich, March 1997 Moira Norrie, Alain Würgler

Preface to the Second Edition

This document is intended as a tutorial introduction to the object-oriented data management system OMS Pro. The system supports the development of database application systems through rapid prototyping and lies at the centre of the tools and technologies that comprise the OMS database development suite.

The OMS Pro system was developed by Alain Würgler within the Global Information Systems group of the Institute of Information Systems at ETH Zurich.

Through a simple demonstration database, we show how an application is modelled and provide a step-by-step guide to using the system to browse and update the database. We also introduce the query language of OMS, called AQL, and describe how operations in terms of methods, triggers and macros can be defined.

Such a document cannot provide full details of the model, the query language, the Prolog programming language, in which object methods are specified, and all of the system features. However, we hope that it provides a good starting point for someone who wants to learn about the main features of the system and to start to use the system.

An on-line manual for OMS Pro is available at: **`http://www.oms.ethz.ch/omspro`**. Other information on OMS Pro, other OMS systems and also the general OMS database development environment is available through the Web pages of the Global Information Systems group: **`http://www.globis.ethz.ch`**. These pages also provide access to various on-line demonstrations including Web-based access to OMS databases through the Internet OMS and OMS Web systems.

Zurich, March 2000 Moira Norrie

Preface to the Third Edition

In the last three years since the second edition of this tutorial the OMS Pro system has a lot evolved. The most striking difference to the previous version is probably the inclusion of the OMS Interaction Language (OIL). This language has been specially designed to implement operations such as macros, methods and triggers within a database. The documentation of this new programming language together with some example routines has been integrated into this tutorial. Hence, Chapter 6 and Chapter 7 have almost been completely rewritten to separate the notion of working with operations from the actual implementation of operations.

But there have also been more visible changes to the system. The entire user interface has been rewritten in response to the user feedback given to the developers over the past years. OMS Pro now no longer opens a window for every object, thereby cluttering the desktop and making it easy to lose one's overview. Rather the system now works as a single window and objects are displayed as child windows within the main panel of the application. Furthermore usability aspects have been taken under consideration and frequently required functionality is now better accessible through the user interface. Collections and macros have been placed in panels that are visible all the time, allowing these objects to be used with much fewer mouse clicks. Moreover, important commands are now available from toolbars and icons at the top of every OMS window. Chapter 3 covers all changes and improvements of the user interface while explaining at the same time how to browse a database.

Not all new developments, however, are presented and discussed in this tutorial. OMS now features a built-in content management framework called *OMSwe* which uses a concept of web elements to publish data from OMS databases on the web and to build web applications that need to work with a database. Thus, it is very simple to provide web access to databases and allow users to update their content over the web. OMSwe also features device independence and context-dependent deliveries, as it allows the generation of various output formats besides HTML.

Besides these developments at the core of the OMS system, there has also been a lot of work in building applications based on OMS. The *eXtensible Information Management Architecture* or XIMA, for instance, is a system that allows universal client access to OMS databases. Thus, databases can be browsed using arbitrary client devices such as PDAs, mobile phones or even regular phones using voice recognition and synthesis technologies.

Linking digital and paper-based information while allowing the user to go back and forth between these media seamlessly, is the aim of *Paper*$^{++}$, an EU project that also uses technologies of the OMS product family, such as *iServer* dynamic linking.

Although OMSwe provides basic content management features, it has also some shortcomings and limitations. The *XCMServer* content management system therefore aims to cover all requirements of content management. Demands such as multi-format/multi-lingual content, context-dependent information, personalisation and business workflow support are met using a set of simple clearly defined concepts represented in OM.

Sharing information and distributed collaboration has become more and more important in various application domains. The *Universal Information Platform* project uses OMS to build a common layer of information management that will ease the development of such applications.

Our world evolves towards an information society, where access to and control of information become increasingly important. OMS and its applications therefore strive to deliver the right information to the right person at the right time. We believe that new systems will have to be developed to cater for ever changing technologies. Hopefully, the concepts of OMS will be there to stay.

Zurich, March 2003 Michael Grossniklaus

Contents

Chapter 1

Introduction

OMS is a suite of tools and technologies designed to support database development from conceptual design through to implementation [KNW98, NW97]. All stages of the development process are based on the same abstract model of the application thereby ensuring that the user's working model will match that of the initial conceptual design. Rapid prototyping early in the development cycle assists both requirements analysis and database design and is supported by a special prototyping system OMS Pro [Wue00, NW98].

OMS includes its own object model, OM, which specifies constructs and operations for the management of interrelated collections of data objects. The OM model was designed to be suitable for both conceptual modelling and data management and is generic in that it supports both the design and documentation of databases independent of the eventual implementation platform.

OMS Pro allows the full conceptual structure of an OM application model to be implemented and validated directly. The system supports full database functionality in terms of browsing, querying, constraints, methods, triggers and a persistent working space. Additionally, it supports the prototyping process itself, enabling an application model to be revised and extended incrementally without loss of experimental data.

At the end of the prototyping process, a developer may export the schema, and optionally data, either as SQL for a relational system, or for import to another OMS data management system such as OMS Java [KNSG01, KN00, KN99b, KN99a, SKN98].

The OMS database development suite includes tools and technologies to provide Web access to OMS databases [Nor00, EN98]. OMS Web provides access to OMS databases by means of CGI (Common Gateway Interface) programs executing on the server side. In contrast, Internet OMS is a client-server Web-based architecture in which a generic object browser implemented in Java executes on the client machine. Internet OMS also uses a cooperative client-server caching scheme to improve performance [SEN00, ENK98]. OMSwe is a content management system built into to OMS Pro and distributed with every copy of the system. Furthermore, the eXtensible Information Management Architecture or XIMA provides universal client access to OMS databases for different client devices such as PDAs, mobile phone, voice interfaces or even plain paper [NS02, SNGH02, SGN01].

Also included in the OMS suite is OMS Connect [NPW98], an extended version of OMS Pro which supports the modular development of databases through the controlled, networked sharing of OMS schemas and databases. The benefit of OMS Connect is, that it can be used to build larger databases from modular components, as it has been state-of-the-art for software development for some time now [NP02].

In addition to its role as a prototyping system for database development, OMS Pro also provides a valuable platform for research in all aspects of next generation information systems based on object-oriented technologies. Further, it supports the teaching of both general and specialised concepts of database systems and enables students to quickly and easily develop their own object-oriented databases and validate their own designs.

In the remainder of this chapter, we provide some more background to the OMS Pro system by outlining our approach to database development and introducing the OMS framework in terms of the proposed stages of application development and the underlying model and systems.

1.1 Database Development

As stated above, we believe that both the development and use of a database application system should be based on a single abstract model of the application domain. This means that the same basic structural model should support all of the development stages from conceptual modelling through to implementation. It follows that the underlying data model should be:

- semantically expressive for conceptual modelling,
- independent of the implementation platform, since this is often determined after the initial design phase,
- supported on a number of implementation platforms including at least one for rapid prototyping,
- amenable to a refinement process from design to implementation.

Unfortunately, most object-oriented data models fall short in one or more of these requirements. Commercial object-oriented database management systems (OODBMS), such as Objectivity/DB, GemStone and O_2, tend to have a data model closely tied to the implementation and programming environment and have next to no support for the conceptual design phase. For example, while object data models for conceptual modelling are rich in terms of semantic expressiveness, most OODBMS offer little in the way of constraints and also provide rather limited ways of modelling object roles, classification structures and associations. Even the data model proposed by the Object Data Management Group, ODMG [Cat96], falls somewhere in-between in that it is too type model specific to be suitable for conceptual modelling, but on the other hand too general in that it does not correspond exactly to the data models of the OODBMS. For example, the relationship constraints expressed in the ODMG data model are not supported in many commercial OODBMS.

Conceptual modelling systems such as ConceptBase [JGJ+95] and MOSAICO [MT94] place a strong emphasis on prototyping as an integral part of the design and specification process. Their conceptual schemas can be considered as executable specifications and further can be developed incrementally. The underlying models are semantically rich, thereby allowing systems to be modelled easily and quickly. In the case of MOSAICO, there is also a strong theoretical basis upon which a schema validation process is based. However, there is still a large semantic gap between these models and those of the eventual implementation platform.

We advocate an approach of *refinement* rather than *mapping*. At all levels of design and implementation, the information is viewed in terms of the same basic model and schema with detail being added at each stage. This contrasts with mapping approaches where constructs of one model are mapped into another and the final implementation model may bear little resemblance to the original design.

One of the best known general object-oriented modelling techniques is the Unified Modelling Language (UML) [BRJ98], which is also intended to support all stages of application development from

initial design to implementation. However, UML is not intended primarily for database system development, but rather for the development of application systems of which a database may form a part. Thus the emphasis is more on the development of code and application classes and it is often assumed that database functionality will ultimately be provided by a relational storage system. This means that there is a mapping from the object model to a relational model similar to that described for OMT in the book [RBP+91]. Further, a prototyping tool for information system development is not explicitly part of the UML approach. Rather, prototyping is supported implicitly through the general principle of incremental system development supported by object-oriented technologies based on techniques such as subtyping and inheritance. We strongly believe that a database prototyping system is invaluable for validating the basic model before proceeding with method and application program implementation.

We propose a refinement approach to the development of object-oriented databases based on the generic object data model OM and its associated interactive object-oriented database management system OMS Pro which supports rapid prototyping. The OM model is independent of a particular implementation platform in that, unlike most object-oriented data models, it is independent of a particular type model and hence of a particular programming language environment. The generic model has been instantiated in number of environments including platforms based on C++, Java, Python, Oberon [RW92] and Prolog.

1.2 OMS Framework for Database Development

OM is a generic object data model with influences from Extended Entity Relationship models and semantic data models, as well as from object-oriented paradigms. There were two main driving forces in the development of the model. First, we wanted to combine ideas from conceptual modelling with object-oriented concepts. Second, we wanted to provide a model that is both a good framework on which to build a data management system and yet independent of a particular implementation platform.

The key to achieving our desired aims was drawing a clear distinction between the notions of *typing* and *classification*. We model the application domain at the conceptual level by classifying entities into *roles*. How these entities are represented in the database is specified by the *types* of the associated objects. Thus, a database schema is considered as serving a dual purpose in that it is both a description of the application domain and also of its representation in the database. A more detailed discussion of the reasons for and benefits from this distinction can be found in [Nor95].

The strict separation of classification and typing supports the process of database development in various ways: It forms a vertical partition of the problem to be solved in terms of defining an interface between the conceptual application domain and the implementation. The conceptual model using classifications focuses on the roles which the entities may take and on the dependencies among roles. It results in conceptual portability by suppressing structural and behavioural details. Only after having modelled the application domain in terms of a role model, is it important to think about the properties of those entities that are to be stored in the database and decide on the types of the objects representing those entities.

The OMS approach to database development is based on three basic levels as shown in figure 1.1.

The general modelling strategy advocated is to first focus on the entity roles and associations in an application domain and produce an initial graphical OM schema. As will be seen later, OM graphical models are not unlike those of extended Entity-Relationship models and UML object models where only entity/object classes and relationships/associations are represented. We see this as an advantage of our model in that we are able to combine the benefits of Entity-Relationship-style modelling at the conceptual level with those of object-oriented systems at the operational level. Note however,

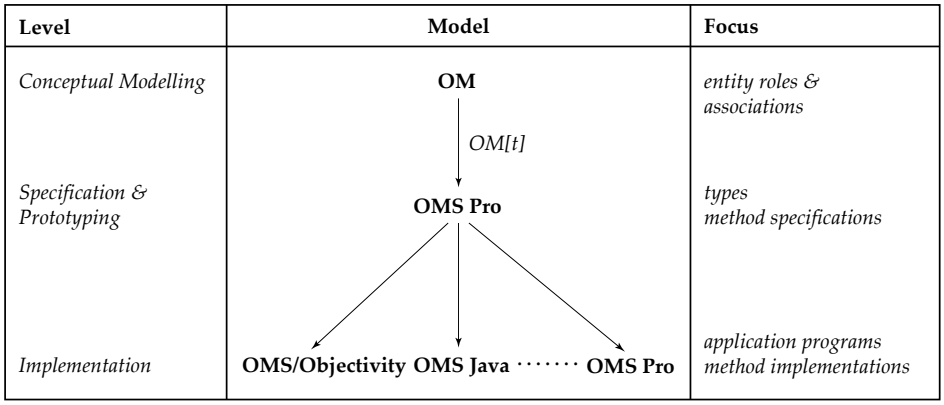


Figure 1.1: OM Framework

that the OM model is not only a structural model, but also an operational model with operations over both roles and associations.

The next stage is to refine the OM role model into a database description that can be used for prototyping. This means that the generic model has to be instantiated with a particular type model t which can be used to specify the representations of entities as database objects. In our case, we use the type model of OMS Pro to specify the properties of objects in terms of attributes and methods.

At this stage of the development process, the problem can be partitioned into the definition of structure and behaviour, without considering the actual implementation of behaviour. By loading the OMS schema definition at this stage, a database is created and data can be entered, browsed and queried. OMS Pro provides a graphical interface for browsing and querying that allows objects to be manipulated directly without the need for any application programming other than a schema definition. In this way, the basic structural design can be prototyped before implementing any of the operational parts of the model.

The description of the database can later be completed by giving the method implementations. Method implementations can be added incrementally and are specified in terms of the OMS Interaction Language (OIL) or Prolog rules using special OMS Pro system-defined predicates for access to data, metadata, system calls and dialogue boxes for user interaction. Thus, the method bodies can be considered as high-level executable specifications. OMS Pro provides a graphical interface for browsing and querying that allows objects to be manipulated directly without the need for any application programming other than a schema definition.

Note that as the schema is refined towards the implementation platform, additional methods and or attributes may have to be added to compensate for the rich set of types and operations which are system-supported in OMS Pro. For example, OMS Pro supports URIs as a base type such that clicking on such a value automatically calls an appropriate application to view the content referenced by the URI. In other systems, such an attribute would have to be replaced by a string attribute for the URI and a method to call the browser with the appropriate address.

The second major refinement step involves taking the schema resulting from the prototyping stage and implementing it on the chosen OODBMS. It would be possible to map the schema to a schema expressed in terms of the data model of the OODBMS. However, as stated earlier, there are disadvantages to such a mapping approach. An alternative is to extend the platform to support constructs of the OM model. In our experiences, comparing the two approaches, the latter produces much more easily understood and flexible systems amenable to future extensions.

Clearly, the amount of work involved in realising the OM model on a particular implementation platform depends on the semantic and functionality gap. In figure 1.1, we show a range of imple-

mentation platforms. Not only do these cover different programming language environments, but also different levels of support and different styles of system. For example, in terms of an actual OODBMS such as Objectivity/DB much of the database functionality such as persistence, transaction management and access methods are already there. In contrast, a system supporting database programming in Oberon has also been developed and, in this case, it was necessary to implement all functionality including persistent storage [Sup99, SN97]. We include OMS Pro at the implementation level since it can be used not only as a prototyping system, but also as an implementation platform for lightweight databases.

OMS Java [KNSG01, KN00, KN99b, SKN98] is both an application development framework and an object data management system. The system implements the OM model within the Java environment and manages collections of objects dynamically built from one or more Java object instances. It is therefore able to support Java application programming, while providing the flexibility of OM information modelling. OMS Java has been designed to support model extensibility for specialist applications such as the management of temporal or spatial data. Further, the core of OMS Java is independent of a particular storage platform and it is possible to use the OMS Java framework to provide higher-level object management on top of a variety of existing database management systems [KN99a].

Chapter 2

Application Modelling

The initial stages of designing a database application system involve constructing an abstract model of the application domain. This model specifies not only *what* information is to be stored in the database but also *how* this information is to be organised. The application model is therefore central in determining not only the functionality of the system, but also its performance in terms of how information is accessed and processed.

It is important to emphasise that a “bad" application model will result in a “bad" application system in terms of functionality and performance — no matter how efficient and sophisticated the underlying database management system. For this reason, we emphasise the importance of these initial design stages in ensuring that the requirements of the system are met and that the information is structured accordingly.

In this chapter, we first introduce the OM model through an example of a database for maintaining information on contacts and then show how the schema for this database is defined in OMS.

We start with a brief description of the application:

> *Contacts may be either persons or organisations and we store information such as phone numbers, fax numbers, Web and email addresses. An organisation may be a company, a university or a society. Further, it may be any organisational unit such as the department of a company. One organisation may therefore be part of another organisation. Associated with persons and organisations are locations which have a physical address in terms of a building, street etc. and possibly a separate postal address. We may also store photographs of locations and maps. For persons who are private contacts, we record additional information such as birthdates and various phone numbers such as a home phone number, a mobile number or that of a holiday home. Additionally, we are interested in the home locations of private contacts. We are also particularly interested in persons currently associated with ETH – including staff and students – and also those who were previously associated with ETH.*

First we introduce a graphical OM model of the contacts database, locating the entities, their roles and their associations. As a refinement, we will determine the properties of these entities in order to derive an OMS schema using the OMS data definition language, DDL.

2.1 OM Graphical Model

The general modelling strategy advocated is to first focus on the entity roles and associations of interest to an application system and model these graphically using OM. We show the OM model for our example contacts database in figure 2.1 and will use this to explain the constructs and graphical notation of the OM model.

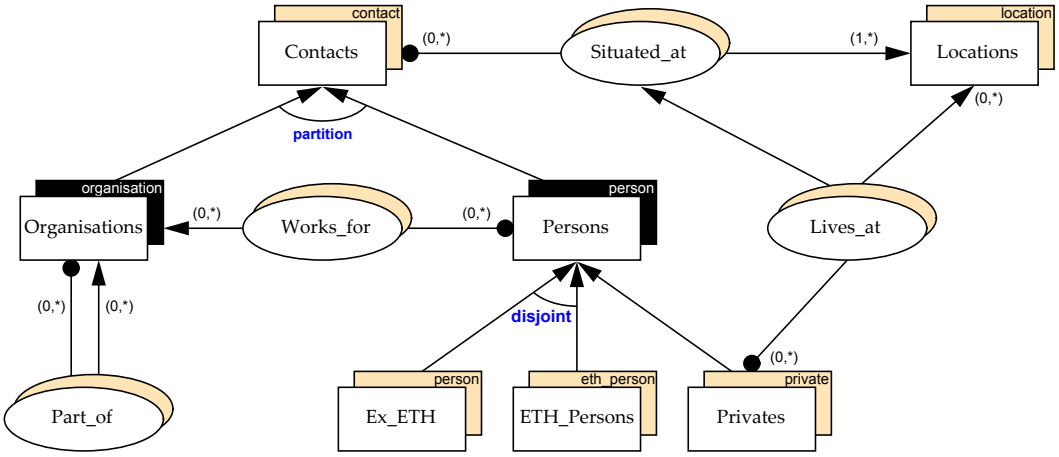


Figure 2.1: Contacts Database

A contact may be either a person or an organisation and so we have the role **`Contacts`** partitioned into the more specialised roles **`Persons`** and **`Organisations`**. Object roles actually correspond to collections of objects of a given type and we represent them graphically as a shaded rectangle with the name of the collection in the unshaded part and the type of the members given in the shaded part.

`Persons` and **`Organisations`** are *subcollections* of **`Contacts`** as shown by the arrows from these collections to **`Contacts`**. We say then that **`Contacts`** is a supercollection of both **`Persons`** and **`Organisations`**. The arc over these arrows labelled **`partition`** specifies a constraint that every member of **`Contacts`** must also be a member of exactly one of **`Persons`** or **`Organisations`**.

The role **`Persons`** is further specialised into **`Privates`**, **`ETH_Persons`** and **`Ex_ETH`** to represent private contacts, ETH associated persons and ex-ETH persons, respectively. Nobody can be an ETH person and an ex-ETH person at the same time, so we specify a **`disjoint`** constraint over the corresponding subcollection relationships. Generally, a disjoint constraint over two or more subcollections specifies that no object of the supercollection may be in more than one of those subcollections.

Another form of constraint over subcollections is the **`cover`** constraint which specifies that every object of the supercollection must be in at least one of the subcollections. A **`partition`** constraint over subcollections is equivalent to having both a **`disjoint`** and **`cover`** constraint. The combination of these constraints means that every object of the supercollection must be in exactly one of the subcollections.

In addition to **`partition`**, **`disjoint`** and **`cover`** constraints, there exists an **`intersect`** constraint over subcollections. The **`intersect`** constraint can be used when a subcollection has two or more supercollections. It is used to specify that any object which is in all of these supercollections must also be in the subcollection.

At this stage, the types are only named and the definitions are left unspecified. Note that an object may be a member of any number of collections and have any number of types. Thus, a person may be classified simultaneously as an ETH person and as a private contact, which means that the

corresponding object will be in both **ETH_Persons** and **Privates**. Then such an object must have at least the types **contact**, **person**, **eth_person** and **private**. It has, however, only one representation and a type specifies that part of an object's representation, i.e. set of properties, which is visible when accessed in a specific context. Thus, if an object is accessed as a member of collection **Contacts**, then only the properties defined for all contact objects will be visible. However, if we choose to explicitly browse that object as an **eth_person** or access it instead through the collection **ETH_Persons**, then the properties defined for ETH persons, together with those inherited from contacts, will be visible. Thus, the member type of a collection may be considered as specifying a view on objects accessed through that collection.

OM has an association construct to represent relationships between objects at the same level of abstraction. For example, **Works_for** in figure 2.1 is an association between **Persons** and **Organisations**. The associated cardinality constraints indicate that the number of persons associated with an organisation, and vice versa, is unrestricted. Generally, association cardinality constraints are represented as a pair (i, j) for each of the associated collections. It means that a member of that collection can be associated to a minimum of i and a maximum of j of the other collection, i.e. it specifies the possible levels of participation in an association. In the case that the level of participation is unrestricted, the cardinality constraint will be specified as **(0,*)**.

An association also corresponds to a collection of values – but in this case it is a collection of pairs of related objects. We therefore distinguish *unary collections*, which contain atomic values and correspond to object roles, from *binary collections*, which correspond to associations. By having a general concept of collection that covers both, we are able to apply our notions of *subcollection*, constraints over subcollections such as *partition* and also operations over collections to both roles and associations. For example, the contacts database schema shows an association **Lives_at** which is a subcollection of **Situated_at**, i.e. it represents a more specialised form of the association, in this case between **Privates** which is a specialisation of **Contacts** and **Locations**.

Associations are directed – which is necessary since OM is also an operational model and hence it is important to know for any pair (x, y) of a binary collection to which collections x and y belong. We say that x belongs to the *source* collection and y to the *target* collection. This is indicated graphically by the direction of the association arrow. Thus, for association **Works_for**, the source collection is **Persons** and the target collection **Organisations**.

The **Part_of** association represents the structure of organisations in that organisations may be part of other organisations. Note that, not only may an organisation have several parts, but also an organisation may be part of more than one organisation. This reflects the fact that enterprises are not always purely hierarchical structures. For example, in a university there may be research centres with staff from various departments and hence they can be considered as being part of many departments.

A contact may be associated with any number of locations. It is possible that many contacts may have the same location. The association between contacts and locations is represented by the **Situated_at** association. In our application, it makes little sense for a location to exist independently of any contact and hence we impose a constraint that each location in the database must be associated with at least one contact.

We may wish to record details of the home address of a private contact and so we also have an association **Lives_at** which associates private contacts to their home locations. Association **Lives_at** is modelled as a specialisation of **Situated_at**. This means that the collection for **Lives_at** is a subcollection of that for **Situated_at** and each pair occurring in **Lives_at** must also occur in **Situated_at**.

While OM does not specify a particular type model, we note here that there are some features of a type model that allow the expressiveness and flexibility of the OM model to be exploited to the full. As described above, we consider that an object may be an instance of many types – which implies that it has the properties associated with all of those types. Each type has a set of explicit structural

and behavioural properties, and further may inherit properties through subtype relationships. These subtype relationships between types defines a type graph which determines property inheritance. While OMS systems support this type model and, hence, a very dynamic and flexible notion of role modelling, other systems are much more restrictive. For example, the type models of most object-oriented programming languages consider that an object has only one type – that of a particular leaf node of the type graph. Thus, a particular type model may restrict the type graph and inheritance mechanisms – but, at the OM level, we do not assume any such restrictions and, in the case where an OM model is mapped to such a system, the model has to be adapted in the mapping process to take such restrictions into account.

2.2 OMS Data Definition Language

Once a graphical model has been developed, the next stage is to refine this into an OMS schema enabling an OMS database to be created. An OMS schema definition is specified in the OMS Data Definition Language, DDL, and consists of three main parts – a type definition part, a collection definition part and a constraint definition part. We describe each of these parts in turn.

First, the required properties of the types named in the OM schema have to be determined and this is the main refinement step in going from an OM graphical model to an OMS schema. Object types consist of a number of attribute properties and a number of method properties. For example, in figure 2.2, we show the type definition for **`contact`** objects.

```
type contact
( name        : string;
  phone       : string;
  fax         : string;
  email       : uri;
  www         : uri;
);
```

Figure 2.2: Type **`contact`**

For each contact, we record a name, a phone number, a fax number, an email address and a World Wide Web (WWW) address. The attribute properties **`name`**, **`phone`** and **`fax`** are of type **`string`**, while the attributes **`email`** and **`www`** are of type **`uri`**. Type **`uri`** is used to store Universal Resource Identifiers (URI) values used for local files, email or WWW addresses (URLs). The URI is input as a string and the values are later displayed according to their type. Double clicking with the right mouse button on a url for example causes the associated page to be displayed using the system's default WWW browser. Whereas doing so on an email address will start the default email client and create a new message to this address.

All attributes of type **`contact`** are single valued attributes, but, as we shall see later, OMS also supports multi-valued attributes through **`set`**, **`bag`**, **`ranking`** and **`sequence`** constructors. A set contains no duplicates and is unordered. A bag is also unordered, but elements may have multiple occurrences. A ranking is ordered and elements may occur only once. A sequence is ordered and elements may have multiple occurrences.

The type definition for **`person`** is shown in figure 2.3. Type **`person`** is declared as a subtype of **`contact`** which means that it inherits all attributes and methods of type **`contact`**. It also has additional attributes for a title (Prof, Dr, Mr, Mrs ...) and the name of a file containing an image for their photo.

Type **`person`** also has a method **`work_places`** which evaluates the set of possible work places of a person by composing the associations between a person and the locations of the organisations for

```
type person subtype of contact
( title              : string;
  photo              : uri;
  work_places        : ( ) -> ( locations: set of location );
);
```

Figure 2.3: Type **person**

which they work. It is a method returning a set of location objects as a result. Details of how methods are implemented are left until a later chapter.

```
struct phone_struct
( description        : string;
  number             : string;
);

type private subtype of person
( birthdate          : date;
  phone              : set of phone_struct;
  music              : uri;
  transport          : uri;
  age                : ( ) -> ( years: integer );
);
```

Figure 2.4: Type **private**

Private person contacts are represented by objects of type **private** as defined in figure 2.4. These persons have a **birthdate** attribute with values of type **date**. Method **age** uses the value of **birthdate** to calculate the age of a person. If the birthdate is unknown, then a null value will be returned. A private contact may also have various phone numbers such as a home phone, a mobile and a holiday apartment. We represent this by defining a multi-valued attribute **phone** with values that are of type **phone_struct**. Type **phone_struct** is a *structure type* with two fields. The first field **description** is used to describe what kind of phone number it is and the second field **number** will be the phone number itself. Both structure fields are of type **string** since the phone number may also contain non-numeric characters. Thus, for a particular private contact, the **phone** attribute could have the value **{(work,'+41-1-6327273'),(mobile,'+41-79-6382614')}**.

Two additional attributes are **music** and **transport** which are both of type **uri**. As briefly mentioned before attributes of type **uri** can be used to reference local files such as image, audio or video files. URI values are always displayed as a text field, where the resource reference can be edited and a set of two buttons to open/close and browse for the resource. Clicking on the **O** (open) button will, according to the resource's type, either start a particular application to view the file or will display it directly within the object. After clicking the open button, the symbol on the button changes to an **X**. Clicking on this close button will collapse the object window to hide the displayed resource. In this way, standard applications and plug-ins can be called to display images, videos, Postscript files and play audio. Thus, **music** can be the person's favourite video and **transport** a video or image of the person's mode of transport, i.e. a car or bicycle.

For persons associated with ETH, we store their office number and also an indication of their expertise through the attribute **activities** which is a multi-valued attribute with sets of strings as values. The definition for type **eth_person** is given in figure 2.5.

Type **organisation** is also a subtype of **contact**. As an additional attribute, it contains a description of the organisation as a text value. The type definition for **organisation** is given in figure 2.6.

```
type eth_person subtype of person
( office              : string;
  activities          : set of string;
);
```

Figure 2.5: Type **eth_person**

```
type organisation subtype of contact
( description         : string;
);
```

Figure 2.6: Type **organisation**

The final type definition **location** is given in figure 2.7. Objects of type **location** have all the address information of a physical location – including possibly a postal address which may differ from the physical address. For example, many organisations use a box number as a postal address.

```
type location
( building       : string;
  street         : string;
  postcode       : string;
  city           : string;
  country        : string;
  postal_address : string;
  photo          : uri;
  map            : uri;
);
```

Figure 2.7: Type **location**

A location may have a photo represented again as an attribute of type **uri**. A map of a location may be included as well as a value of also type **uri**. For example, a location may have the map attribute value **file://map.gif** will be recognised as the format of an image file and displayed either within the object or using an image viewer such as **xv** on Unix systems.

After having declared all types used in the graphical schema shown in figure 2.1 at the beginning of this chapter we go on to specify the collections and associations used in the contacts example. The **Contacts**, **Persons**, **Privates**, **ETH_Persons**, **Ex_ETH**, **Organisations** and **Locations** collection are as declared in DDL as shown in figure 2.8. To define a collection it is necessary to specify its name, its type (**set**, **bag**, **ranking** or **sequence**) and the type of its member objects. Note, that sometimes collections are also called *unary collections* to emphasize that they contain single objects and not tuples of objects.

The contacts schema example also defines the **Part_of**, **Works_for**, **Situated_at** and **Lives_at** association. Associations between unary collections are also said to be *binary collection* as they contain tuples of objects. The DDL statements to declare these four associations are given in figure 2.9. Defining associations however is slightly more complex than defining unary collections as the type of the association's member objects has to be defined explicitly as a *structure type*. The procedure is the same as used for the **phones** attribute in type **private** where a structure type had to be declared to represent phone numbers as a tuple of a description and a number. Associations are directed from one collection to another. Thus, the respective structure type definitions have two fields, one to denote the source and one for the target of the association. As with unary collections

```
collection Contacts      : set of contact;
collection Persons       : set of person;
collection Privates      : set of private;
collection ETH_Persons   : set of eth_person;
collection Ex_ETH        : set of person;
collection Organisations : set of organisation;
collection Locations     : set of location;
```

Figure 2.8: Collection Declarations

it is also possible to define associations of a different type than **set** depending on whether or not members are ordered and may have multiple occurrences.

```
struct part_of_struct
( sub_organisation       : organisation;
  parent_organisation    : organisation;
);
struct works_for_struct
( employee               : person;
  employer               : organisation;
);
struct situated_at_struct
( contact                : contact;
  location               : location;
);
struct lives_at_struct
( person                 : private;
  home                   : location;
);

collection Part_of       : set of part_of_struct;
collection Works_for     : set of works_for_struct;
collection Situated_at   : set of situated_at_struct;
collection Lives_at      : set of lives_at_struct;
```

Figure 2.9: Association Declarations

Finally, the constraint definitions are given in figure 2.10. They impose restrictions over collections in terms of subcollections and associations. First, we show the constraints for associations. Each of the four associations has a constraint detailing the source and target collections which it associates and the cardinality constraints.

There are six subcollection constraints. Collections **Persons** and **Organisations** are both subcollections of **Contacts**. Collection **Privates**, **ETH_Persons** and **Ex_ETH** are all subcollections of **Persons**. The binary collection for **Lives_at** is declared to be a subcollection of that for **Situated_at**.

The third group of constraints specify the partition and disjoint constraints. The first specifies that subcollections **Persons** and **Organisations** form a partition of **Contacts** which means that every contact object must belong to exactly one of these subcollections. The second specifies that **ETH_Persons** and **Ex_ETH** are disjoint subcollections of **Persons** meaning that no person object can simultaneously be a member of both of these subcollections.

The final group of constraints specify that collections **Persons** and **Organisations** are fixed classifications, i.e. they represent fundamental *kinds* of entities, as opposed to *roles* which may be gained

```
constraint Works_for association from Persons (0:*) to Organisations (0:*);
constraint Part_of association from Organisations (0:*) to Organisations (0:*);
constraint Situated_at association from Contacts (0:*) to Locations (1:*);
constraint Lives_at association from Privates (0:*) to Locations (0:*);

constraint Persons subcollection of Contacts;
constraint Privates subcollection of Persons;
constraint ETH_Persons subcollection of Persons;
constraint Ex_ETH subcollection of Persons;
constraint Organisations subcollection of Contacts;
constraint Lives_at subcollection of Situated_at;

constraint (Persons and Organisations) partition Contacts;
constraint (ETH_Persons and Ex_ETH) disjoint Persons;

constraint Persons is kind;
constraint Organisations is kind;
```

Figure 2.10: Constraint Declarations

and lost. In this case, this means that an object cannot be removed from these collections, unless the object is deleted from the database. These constraints are therefore used to give some control over how an object may evolve over time in terms of changing its classification. A person cannot become an organisation, but they may become a private contact and an ETH person may find new employment and become an ex-ETH person.

This completes the description of the OMS schema for the contacts database application. It is sufficient to introduce the structure of the example database to allow a user to work with the database. Note that the full syntax of the OMS data description language DDL is given in Appendix B.

In the next chapter, we will show how to use OMS Pro to browse and update the database. This follows with a chapter introducing the query language and showing how to use OMS Pro to query the database. Later chapters, will describe how you can create your own database and how to implement macros, methods and triggers.

Chapter 3

Browsing and Updating a Database

In this chapter, we give a tutorial introduction to the OMS Pro system by taking you through a session browsing the contacts database system and adding new information to the database. Note that you should first ensure that you have a personal copy of the contacts database. This allows you to freely experiment without fear of corrupting the original demonstration database!

3.1 Starting OMS Pro

After starting the OMS Pro system, the main OMS window will appear as shown in figure 3.1.

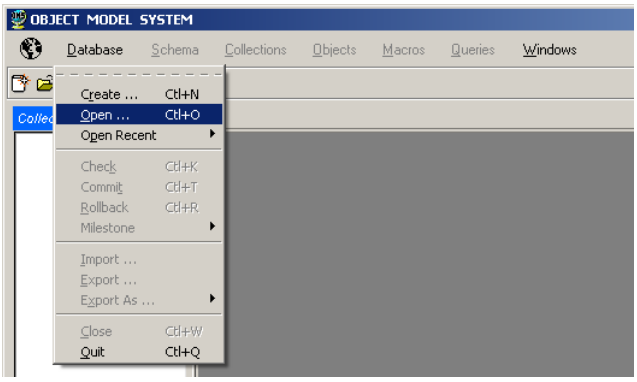


Figure 3.1: Opening a Database

3.2 Opening a Database

To browse the example contacts database, you must first open the database. Do this by first clicking on the **`Database`** menu button with the left mouse button, then holding the button pressed and releasing it on the **`Open`** option of the displayed menu. Then the dialog window shown in figure 3.2 will appear.

Double clicking on directory entries opens that directory. In this way, you can navigate through the directories. Note that double clicking on the entry ".." causes you to move up in the directory structure. The current directory may also be changed by typing in a new directory in the **`Selection`**

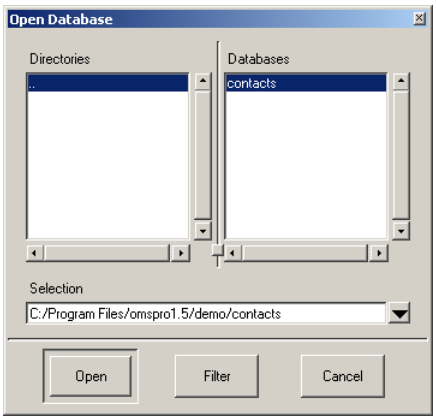


Figure 3.2: Selecting a Database

text field at the bottom of the window and pressing return. If a selected directory contains any OMS databases, these will be listed under **`Databases`**. You should look for your copy of the database named **`contacts`**.

A single click on a database entry enters the name of the database at the top of the window under **`Selection`**. Clicking on the **`Open`** button will then open the database. Alternatively, a double click on a database entry will open the database directly. You should open the **`contacts`** database. A message that the database has been opened will be displayed in the message panel at the lower right of the main window. You are now ready to start browsing the contacts database.

Note that recently opened databases may be opened through the **`Open Recent`** option of the **`Database`** menu of the main OMS panel. A list of recently opened databases will be displayed and you must simply select the appropriate one.

3.3 Finding Objects

Assume you want to find information about the person Moira Norrie. Then we want to locate and display the object of type **`person`** with name value **`Moira Norrie`**. To locate and display individual objects of the database, we use *object matching*. Invoke the **`Objects`** menu of the main window and then select the **`Match...`** item. A list of object types appears as shown in figure 3.3.

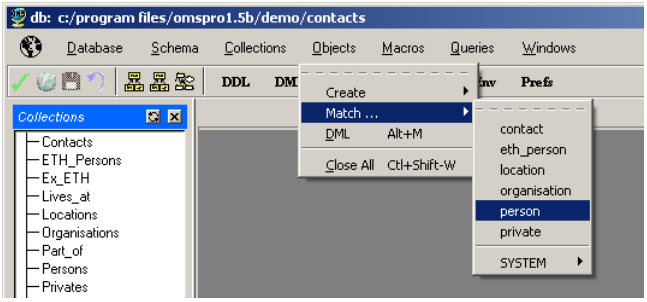


Figure 3.3: Finding Objects

To find a specific person, select the **`person`** menu item and the form shown in figure 3.4 will appear. The entries in such a form will be used to select matching person objects.

Figure 3.4: **`person`** Match

In figure 3.4, we enter **`Moira Norrie`** into the **`name`** field of the form in order to find a person object with that name. No partial matching of attribute values is performed and hence the complete name as appears in the database must be given. Generally, you can enter values in as many or as few fields of the form as required – however, in this case, it is sufficient to specify only the name value.

Click on the **`Match`** button and as a result the matching person object is displayed as shown in figure 3.5.

If several objects match, all objects will be returned and these objects are placed into a temporary collection which is displayed on the screen. Double clicking on an individual object entry in this collection will then cause the object to be displayed.

Exercise

1. Find the organisation "University of Glasgow".

2. Perform a match to find all person objects with the title **`Prof`**.

Note: Alternatively, if a match operation returns multiple objects, these objects may also be displayed as windows within the main panel of the OMS application window instead of generating a temporary collection as a holder for these objects. This option can be specified using the **`Results`** menu at the top of the object match window. The **`Default`** is to display single results directly as objects. **`On Desktop`** specifies that all results should always be displayed directly within the main panel. Remember to set this option with caution as it can be very time-consuming when the number of results is very large. **`In Collection`** specifies that all match operations should generate a temporary collection to hold the results — even if the match returns a single or no object.

3.4 Executing Operations on Objects

Operations on objects are made available as *object methods*. The methods of an object are represented by arrow buttons placed alongside the method names as shown in the person object of figure 3.5.

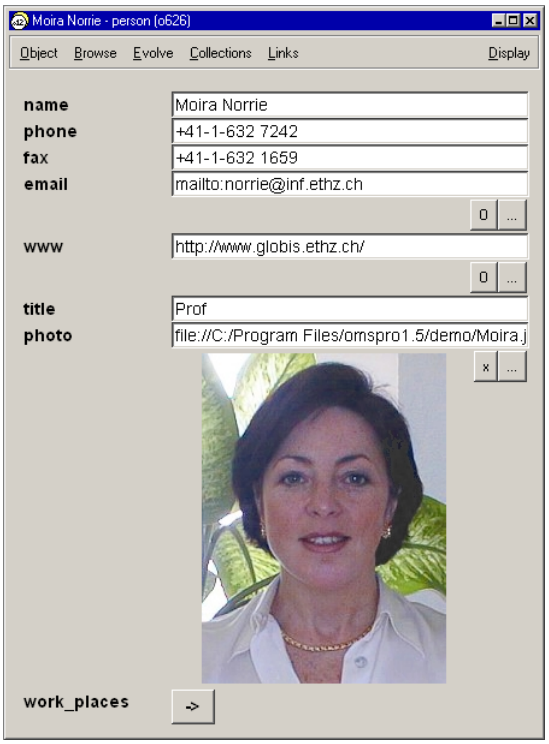


Figure 3.5: **person** Object

The method **work_places** returns the set of location objects of the organisations for which the person works. Clicking on that method button therefore displays a set of location objects as shown in figure 3.6. Double clicking on one of the location objects causes the object to be displayed.

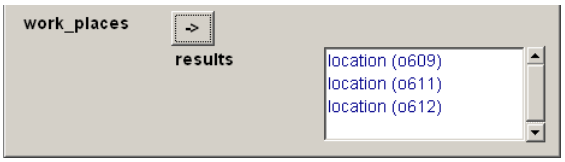


Figure 3.6: Using Methods

3.5 Accessing URIs

As mentioned in Chapter 2, attributes of type **uri** are displayed as a standard text field together with a pair of buttons to open/close and browse for resources. Clicking on the **O** (open) button with the left mouse button will either cause the resource to be displayed directly within the object, if possible, or will call an external application such as a Web browser to view the URI content. Whenever a resource is being displayed the symbol on this button changes into an **X**. To hide the resource viewed within the object window click on the **X** (close) button with the left mouse button. If however the resource is viewed outside the object window, the close button has no effect on the viewer application, i.e. it cannot be used to close the external programme. If it should be required to force OMS to use an external application to view URI content, this can always be done by double clicking with the right mouse button on the text field of the **uri** attribute.

For example, clicking on the **O** button for **www** in the object of figure 3.5 will cause the specified Web page for Moira to be displayed. Similarly, the email attribute is also of type **uri** and clicking on its open button will cause a send email window to be displayed.

3.6 Navigating by Links

The **Links** menu at the top of an object window allows easy navigation to associated objects. For example, from a person object you can find associated location and organisation objects through the **Situated_at** and **Works_for** associations as shown in figure 3.7.

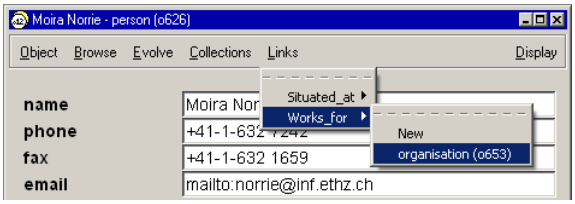


Figure 3.7: Navigating by Links

Thus, in the case of Moira, there is a single organisation object **Works_for**. Selecting that item from the **Links** menu, causes the organisation object to be displayed.

Exercise

1. Find the locations at which Moira is situated.
2. Find someone else who works for the same organisation as Moira.

3.7 Closing Object Windows

At any stage, any object window can be closed by selecting the **Close** option from the **File** menu displayed at the top of the window. To close all current object displays, select **Close All** from the **Objects** menu of the control panel.

3.8 Navigating through Collections

Instead of using object matching to locate entry points into the database, it is possible to start by displaying collections. If the collections panel is displayed at the upper left of the OMS application window, double click for example on the **Contacts** entry to display the collection in the main panel. If the collections panel is currently not displayed you can either open it from the **Windows** menu by selecting the appropriate entry in the **Panels** menu item or you can open the **Contacts** collection by selecting it from the **Collections** menu as shown in figure 3.8. This menu can always be used to display a full list of all collection currently defined in the database.

Double clicking on an object in a collection causes the object to be displayed. For example, scrolling down the collection **Contacts** and double clicking on the entry **contact(o626)** of collection **Contacts** will display the object shown in figure 3.10.

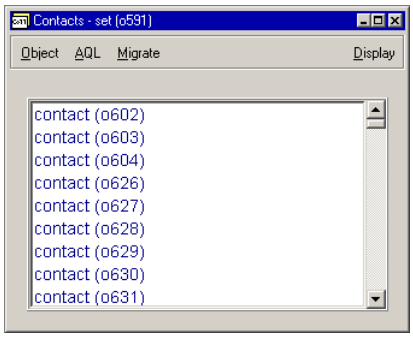


Figure 3.8: Collection **`Contacts`**

3.9 Viewing Collections as Objects

Since collections are themselves objects, they could be displayed in the same style as any other object with attributes and their values. The default display of collection objects is simply a more convenient form in which the member objects are displayed as a list of clickable entries. Selecting the **`Object View`** entry of the **`Display`** menu allows you to view the collection using the regular object presentation as shown in figure 3.9. Here we see that a collection has three attributes – a name, a type for its members and an extension which gives the objects in the collection.

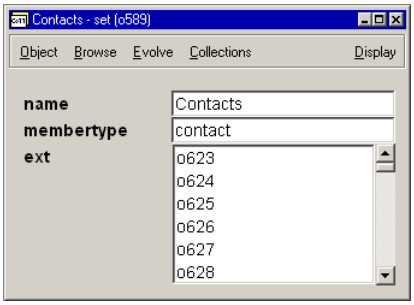


Figure 3.9: Object View of a Collection

Return to the special collection style of presentation by selecting **`Collection View`** from the **`Display`** menu.

3.10 Browsing Different Views of Objects

The object view in figure 3.10 is that of a **`contact`** object with only those attributes of type **`contact`**. Generally, an object is always viewed in the context of some type. In particular, clicking on an entry in a collection causes the object to be viewed in terms of the specified member type of that collection.

If the object has other types, it is possible to browse it as an object of another type by changing the presentation and possibly its behaviour. To do this, select the **`Browse`** menu and then select from the list of possible types. For example, in figure 3.11, we show how we could choose to view Moira as a **`private`** object rather than simply as a **`contact`** object.

Figure 3.10: **contact** object

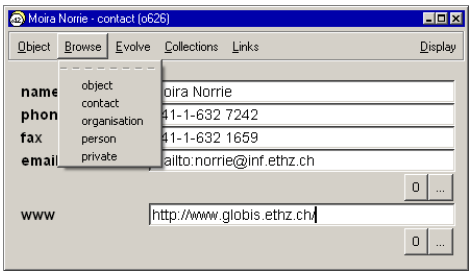


Figure 3.11: Browsing an object

Exercise

1. Browse the object for Moira as a **contact**, **person**, **private** and **eth_person** object. Note which attributes and methods are displayed in each case. What do you think is the general rule for the object attributes and methods displayed in a particular context?

2. As you browse the object for Moira under the various types, note also the various navigation links available under the **Links** menu. What do you think is the general rule as to which links are displayed in a particular context?

3.11 Changing Objects by Evolution

In the **Evolve** menu of objects are the evolution operations **Dress Me** and **Strip Me**. Note that these operations actually change the object and not just its presentation! The **Dress Me** option allows an object to be specialised by gaining an additional type and its attributes. The **Strip Me** option causes an object to lose some of its more specialised types and hence some of its attributes.

Exercise

1. Strip the object for Moira to type **contact**. Then dress Moira with type **person**. What has happened to the object?

2. The original version of the object can be recovered by selecting the **Rollback** option of the **Object** menu.

3.12 Migrating Objects between Collections

An object may change its roles over time and this corresponds to an object changing its membership of collections. Migration of an object within a classification structure can be performed through the **Migrate** menu of the collection from which the object is to be migrated. For example, assume that an ETH person leaves ETH. Then we would wish to migrate the corresponding object from **ETH_Persons** to **Ex_ETH**. This can be done by displaying the collection **ETH_Persons** and selecting the object to be migrated by clicking on it once with the left mouse button. Then use the **Migrate** menu of the collection to obtain a list of collections to which the object may migrate as shown in figure 3.12.

By selecting the option to migrate to collection **Ex_ETH**, the object will automatically be deleted from the collection **ETH_Persons**, added to collection **Ex_ETH** and have any necessary strip and dress

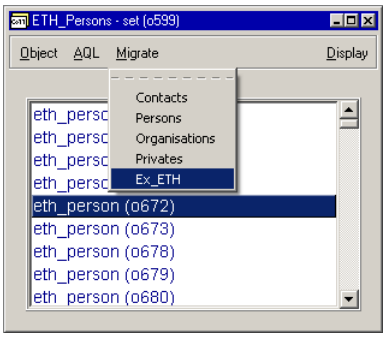


Figure 3.12: Migration from **ETH_Persons**

operations performed in accordance with the member types of the collections. In this example, a strip would be performed on the object to type **person**, thereby causing it to lose all of its **eth_person** properties.

Exercise

1. To reflect the fact that Niklaus Wirth retired in March 1999, migrate him from the collection **ETH_Persons** to **Ex_ETH** and note what happens.

2. Migrate some person objects from **Persons** to **Privates** and note what happens.

3.13 Browsing Associations

Associations are also represented as collections — in this case binary collections in which the members are object pairs representing a specific relationship between two objects. For example, returning to the main window and selecting **Works_for** from the **Collections** menu displays the collection shown in figure 3.13. **person** and **organisation** objects displayed on the same line are related through the **Works_for** association.

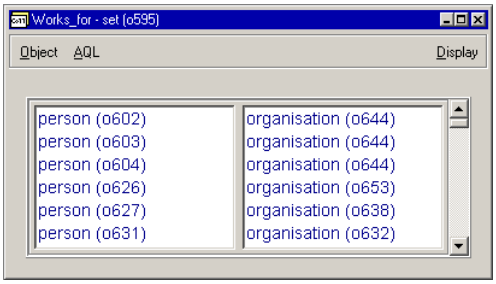


Figure 3.13: Binary Collection for **Works_for** Association

Again, individual objects can be displayed by doubling clicking on the collection entries with the left mouse button. However, double clicking on entries with the middle mouse button causes collection elements to be displayed which, in this case, are object pairs as shown in figure 3.14.

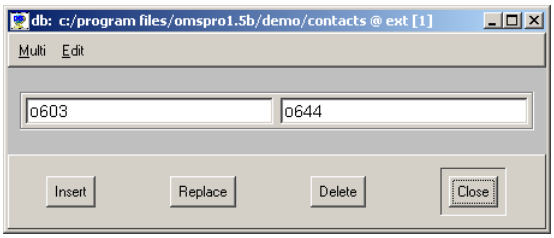


Figure 3.14: Element of **`Works_for`** Association

3.14 Creating Objects

We now consider how to add objects to the database. Under the **`Objects`** menu of the main panel, select **`Create`** and then the type **`person`**. An empty object is displayed the fields of which can be edited with the attribute values. For example, we show a new person object which has been edited in figure 3.15.

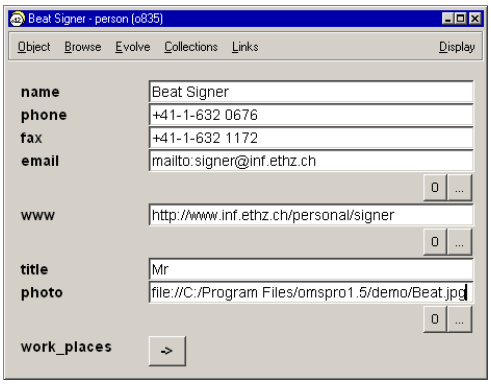


Figure 3.15: Creating a **`person`** Object

Figure 3.16: Adding an Object to Collections

There are a couple of things to note in creating objects. Originally, the entries of the object display indicate the types of the values. If a value is unknown and therefore no value is supplied, the entry can be left as in the original displayed object. If, however, you type a value by mistake and later wish to reset it to indicate "value unknown", delete the value in the entry field and instead type **`null`**. Alternatively it is also possible to position the text cursor within the appropriate field and hit the keys **`Ctrl`** and **`N`**. When the object is displayed in the future, it will be shown with only the type indication.

Attribute **`photo`** has the name of a file as a value. To enter the file name, click on the **`...`** (browse) button. A file dialog that can be used to select a local image file from your file system will be displayed. After having selected a file click **`OK`** and the path to the file will be displayed in the text field of attribute **`photo`**. It is, of course, also possible to simply type in the path to the image file. But remember that since the attribute's type is **`uri`** a path to a local file has to start with the **`file://`**

prefix. Once the name of the file has been entered, click on the **O** (open) button to now display the photo.

The values of attributes **email** and **www** which are also of type **uri** are entered directly in the text field of these attributes. Again these are **uri** values which have to start with a prefix that indicates to OMS which protocol is being used. WWW addresses should always start with the **http://** and email addresses with **mailto:** prefix. To launch the Web browser or email client to work with the entered values, either click the **O** button or double click with the right mouse button on the text field.

Note that date values, such as specified for attribute **birthdate** in objects of type **private**, can take several forms for input and display. The current form is specified in the system preferences. Click on the "globe" at the top left hand corner of the main panel and then select **Preferences**. Select the **Dates** option at the top of the **Preferences** window to see the current format of date values and select an alternative format if required. The internal representation of dates is always the same and these different formats only control the external representation for input and display. Therefore, the date preferences can be changed at any time.

3.15 Adding Objects to Collections

Once an object has been created, it can be added to one or more collections by selecting the **Collections** option at the top of the object window. The system will display a list of possible collections according to the type of the object as is shown in figure 3.16.

Note that adding a person object to the **Persons** collection will automatically add it also to its supercollection **Contacts**. This behaviour of the system is an option specified in **Preferences** under **Extra** preferences. It allows the user to specify that collection insertions and deletions should be propagated automatically in accordance with subcollection constraints between collections. Note further that such propagation applies only to unary collections, i.e. collections of objects and not to association collections.

Exercise

1. Create an object for yourself of type **private** and add it to the **Privates** collection. Check that the object also belongs to collections **Contacts** and **Persons**.

2. Go to the "globe" at the top left of the OMS panel and select the **Preferences** menu option. The system will automatically display the **Dates** preferences. Check the current preference setting for the date format. Experiment by setting different preferences for date values and check what happens with the **birthdate** value of your **private** object as you change the formats. Note that to see the effect of changes, you must first change the dates preference, click on **Save** which closes the preferences window and then refresh the object by selecting the **Browse** option of the same type, i.e. **private**.

3.16 Creating Associations between Objects

To record the organisation for which a person works and its location, we first create the associated **organisation** and **location** objects as shown in figure 3.17. The **organisation** object is added to the **Contacts** and **Organisations** collections and the **location** object to the **Locations** collection.

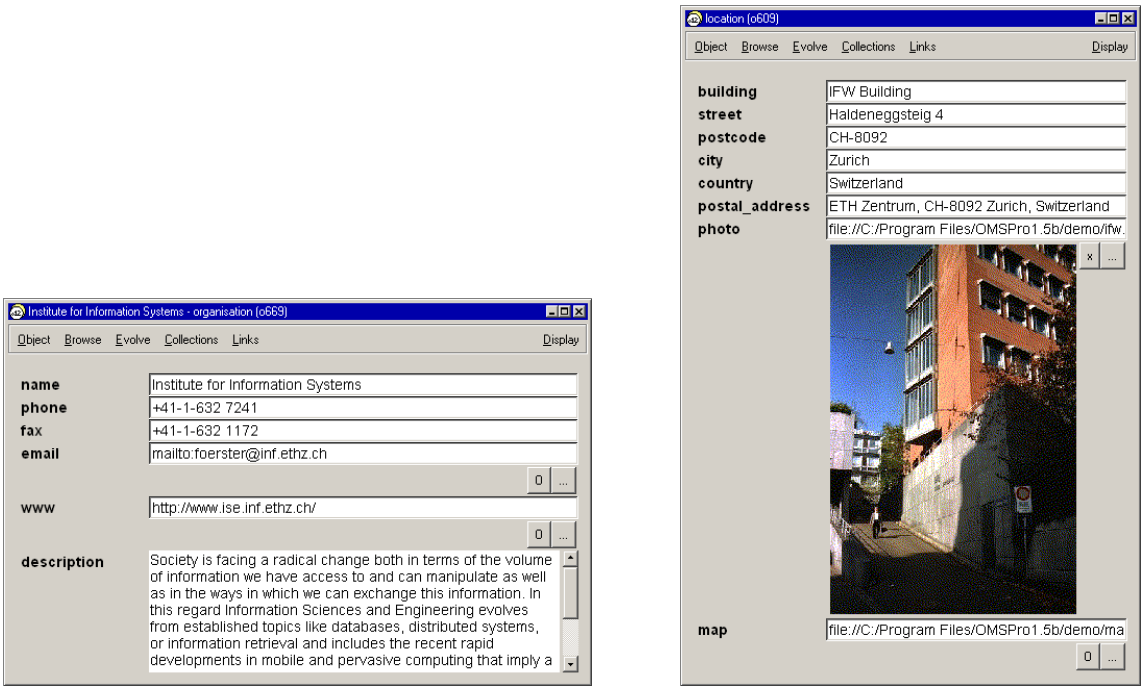


Figure 3.17: Creating **`organisation`** and **`location`** objects

The next stage is to create the associations between the **`person`** object and the **`organisation`** object and between the **`organisation`** object and the **`location`** object. We will describe how to add the association between the **`person`** object and the **`organisation`** object to the **`Works_for`** association.

First select the **`Works_for`** association collection from the **`Collections`** menu of the main panel. In figure 3.18, we show the two objects and the association collection involved in the association along with an association instance.

To create a new entry in the **`Works_for`** collection, double click with the middle mouse button somewhere in the entries display panel of the **`Works_for`** collection to generate an association instance as shown in the bottom right of figure 3.18. This is however only one of two possible representations of association instances in OMS. The other is the default representation as shown in figure 3.14 where the objects are displayed as text fields. To toggle between these two representations hold down the **`Ctrl`** key and click on the object with the right mouse button.

The actual values displayed in the association instance at this stage are not important as we will now update them to create a new instance. This is done by an operation referred to as “object fishing". First double click with the middle mouse button on the **`person`** object in the association instance to indicate that you want to update it. Then you go “fishing" for the object with the cursor by moving over the other displayed objects. Note that the colour of the message panel at the lower right of the main OMS window will change to guide you towards legal fishing! If the panel is red, it means that the cursor is not positioned over a valid object for the update. If the panel is green, it indicates that the cursor is positioned over a valid object. Moving the cursor over the object for Beat on the left of figure 3.18 will cause the panel to change to green. Then click with the middle mouse button to complete the “fishing" process and cause the **`person`** object in the association instance to be updated.

In a similar manner, update the required **`organisation`** object of the displayed association instance. Now that the required instance has been created, it can be added to the **`Works_for`** collection by selecting **`Insert`** from the options on the instance.

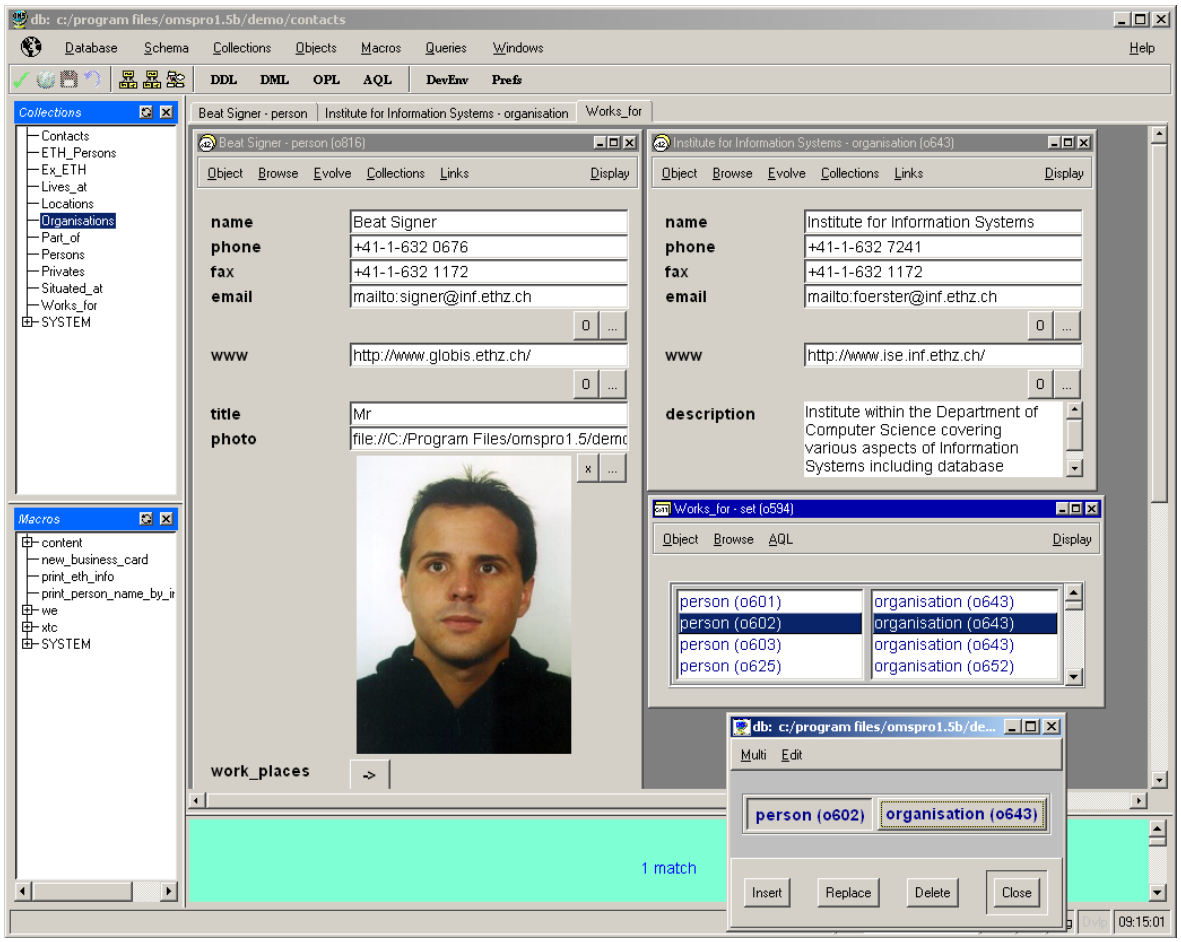


Figure 3.18: Creating an Association

Note: Instead of using the "object fishing" operation it is also possible to type in the objects manually when working with the default representation of an association instance. To simplify this process and to make sure that only legal objects are entered into the association, OMS provides *auto-complete* functionality for these text fields. To use this feature, first clear one of the text fields and then put the text cursor into the blank field. Pressing the **`Ctrl`** and **`S`** key will display a list of matching objects that can be used in this field. Simply choose one of the objects and click on the **`OK`** button. The object will then appear in the association instance window.

Exercise

1. Complete your entry in the database by recording the organisation you work for, its location and also where you live.

2. Assume Beat Signer has obtained his doctoral degree and has a new Professor position in the Department of Computer Science at the University Edinburgh. Make the necessary updates to the database to reflect these facts.

In the case where an association collection is empty, double clicking on the entries display panel will cause an association instance with a pair of null objects of the corresponding types to be displayed. To create the new association instance, these objects can be updated with the required object instances in the same way as described above. Then click on **`Insert`** to insert this pair in the association.

Note that this method of inserting elements in a collection applies to all forms of collections. Thus, to insert a **person** object in the **Persons** collection, you could double click on the entries part of the collection to display an element instance, then update it accordingly and insert it. Clearly, this method is more laborious than simply selecting from the **Collections** menu of an object to add it or delete it from the set of possible collections. However, it is the only means of adding an element to an ordered collection so that it is in the required position. Recall that ordered collections are those defined as either a **ranking** or a **sequence** (see Chapter 2). In this case, the same procedure is followed as above but, instead of clicking on the **Insert** button to insert the object, the **Edit** menu of the instance is used to specify exactly what action is to be performed e.g. to insert the object at the beginning, the end or a selected position of the collection.

3.17 Undoing Updates by Rollback

Changes to the database can be undone using a **Rollback** operation. **Rollback** operations are available at either the object level or the database level as shown in figures 3.19 and 3.20, respectively.

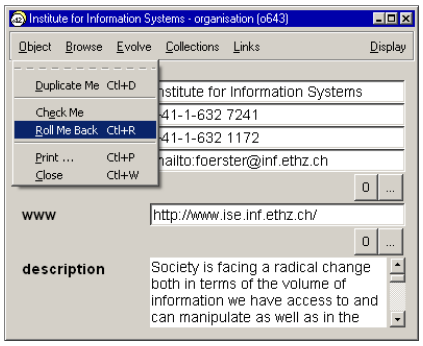


Figure 3.19: Object Rollback

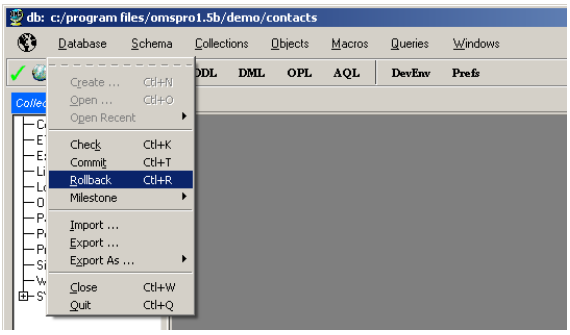


Figure 3.20: Database Rollback

Any changes performed on a database can be undone using the **Rollback** operation of the **Database** menu of the OMS panel.

Caution: The effect of such a **Rollback** operation is to restore the database to its state at the time of the last **Commit** operation.

3.18 Saving Updates by Commit

All changes to the database as stored in the current workspace may be made permanent by using the **Commit** operation available through the **Database** menu of the OMS panel. The operation first checks whether the current state of the workspace satisfies all the constraints specified in the schema. If not, the commit operation will fail and the user can request a **Debug** to find where the constraints are violated. If the database consistency check is satisfied, the system proceeds with the commit operation and writes all changes stored in the workspace to the database and clears the workspace.

Note that rollback information is also part of the workspace. Once a database has been committed, the current workspace is cleared and the user therefore cannot undo any changes that occurred before the commit using **Rollback**.

In OMS, **all** values entered at any stage are automatically made persistent as part of a current workspace. This means that even if you quit a session or the system fails, the values will still be

there the next time that the database is opened. Therefore, it is not necessary to **Commit** at the end of a session. Nor is it necessary to **Commit** for fear of losing work due to system crashes!

3.19 Checking Objects

Normally, all constraint checking is performed at the time of a **Commit** operation. This is the case since, during updates to a database, temporary inconsistencies may arise. For example, during the process of entering a new person in the contacts database, we may create a new location object and add it to **Locations** before adding a new entry to **Situated_at**. Then the cardinality constraints associated with **Situated_at** would be temporarily inconsistent since every object in **Locations** must have at least one association in **Situated_at**.

During our work with a database, we may sometimes wish to check for consistency without actually performing a commit. This can be done on an object level using the **Check** item of an object's **File** menu.

3.20 Setting Milestones

The use of **Rollback** to undo changes is very useful — especially in the case of experimentation with a database as occurs in prototyping. However, there are times when we would like to perform experiments in stages — allowing us to rollback only to a previous state rather than to the last committed state. This can be done by setting *milestones* which record the state of a database at a particular date and time.

Creating a milestone is done by going to the **Database** menu of the main control panel, selecting **Milestone** and then **Create**. Note that no constraint checking is done, since the database state saved does not need to be consistent.

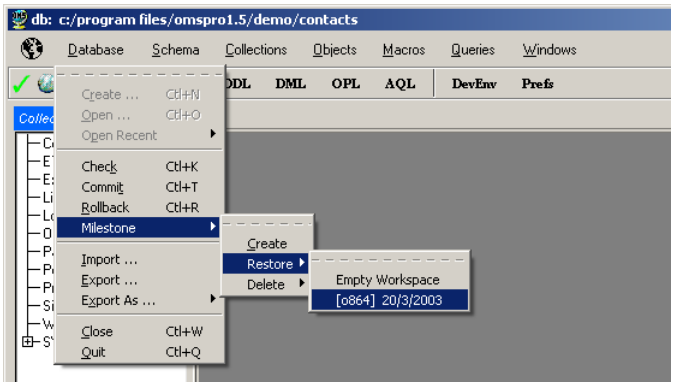


Figure 3.21: Restoring a Milestone

Later, one can go back to a previously recorded database state through the **Restore** entry of the **Milestone** menu as shown in figure 3.21. A list of all milestones is given in terms of their milestone object id and date and the user must select the one they wish to restore. Restoring the **Empty Workspace** is effectively returning to the state of the database at the time of the last commit — however, it differs from a **Rollback** operation in that the set of milestones is not deleted.

Milestones can be deleted through the **Delete** entry of the **Milestone** menu. They will also be deleted when a **Commit** operation is performed.

3.21 Browsing the Schema

The schema information of a database is available in various ways. The most direct way is to view the schema either textually or graphically through the **`Schema`** menu of the main panel. Selecting the **`Definitions`** item from the **`Show`** submenu displays a textual representation of the current schema. As shown in figure 3.22, the item **`Schema Graph`** provides options to display a graphical representation of the base types, types or collections of the schema. The same functionality can also be accessed through the three icons of the second group in the toolbar of the main OMS window.

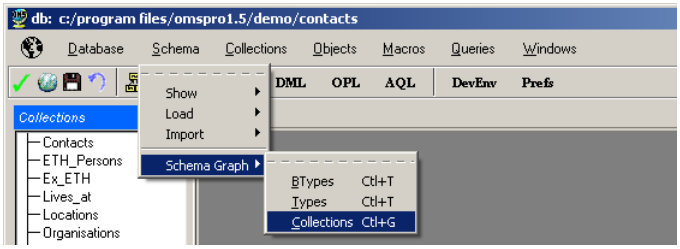


Figure 3.22: Schema Graph Menu

The collection graph for the contacts database is shown in figure 3.23.

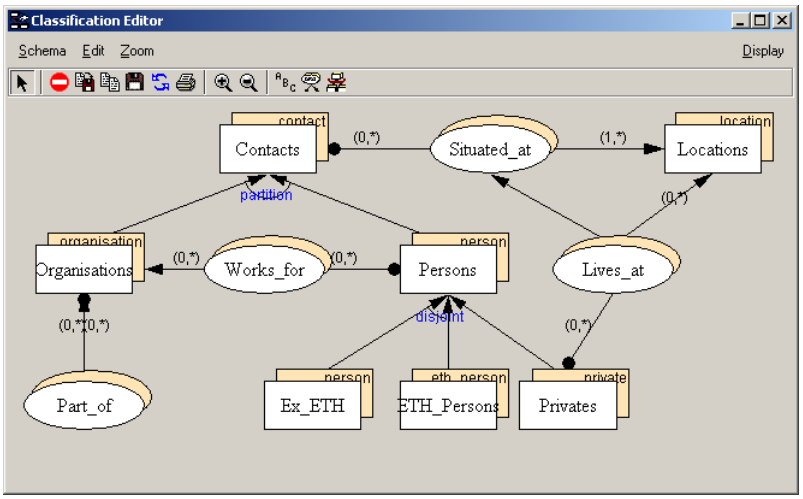


Figure 3.23: Collection Graph

The **`Display`** menu at the top right corner of the window provides various options for the information displayed in the graph. The most important of these is the **`Associations`** option. When this option is **not** set, only collections are displayed and not the links between them as represented by associations. In effect, setting the **`Associations`** option provides you with a full graphical display of the application model in terms of collections, subcollections and associations, together with any constraints.

Initially, the collection layout is generated automatically by the system. The user can then adjust the layout by dragging the various collection and association boxes. When a desired layout has been achieved, clicking on the disk icon in the tool bar at the top of the window will save that layout.

The schema can also be edited through this graphical interface and this is described in Chapter 5 which deals with creating a database.

Double clicking on the various components of the collection graph causes the corresponding objects to be displayed. Thus, double clicking on the collection box **Contacts** will cause that collection to be displayed. Note that this also applies to the special system objects which represent types, constraints etc. as described in the following section. For example, double clicking on the shaded part of the **Contacts** collection which corresponds to the membertype **contact** will cause the object representing that type to be displayed.

Through the **Schema Graph** option there is also access to the type graph as shown in figure 3.24.

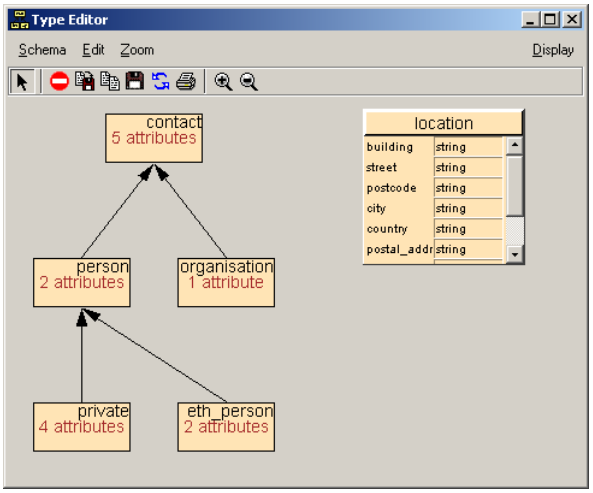


Figure 3.24: Type Graph

Again, the layout may be changed by dragging the type boxes and the new layout saved by clicking on the disk icon of the toolbar. Double clicking on the type boxes will display the corresponding type objects, while double clicking on the arrows connecting type boxes will display the corresponding subtype, i.e. **tisa**, objects.

On the right hand side of the type graph, we show the type **location** in expanded form showing the attributes of the type. The expanded form of a type representation is reached by clicking the right mouse button over the type box and selecting the **Show Attributes** option of the pop up menu. Types descriptions in expanded form enable the user to directly update a type definition by editing the attribute entries or adding new attributes. This is described in more detail in Chapter 5.

3.22 Browsing System Objects

All information in the OMS system is stored as objects of the system. Thus, both model and schema information can be browsed (and updated) in the same way as database information. This can be seen by selecting the **Match** item of the **Objects** option of the main panel and then the **SYSTEM** item to obtain the menu shown in figure 3.25.

As an example, we will consider browsing objects which represent types defined in the schema such as **contact**, **person** and **private**. These user-defined types are represented as object instances of the special system-defined type **ptype** (persistent type).

As an example, assume we want to browse the object representing the type **private**. Then select **ptype** (persistent type) from the **SYSTEM** menu shown in figure 3.25 to perform a match on the ptype

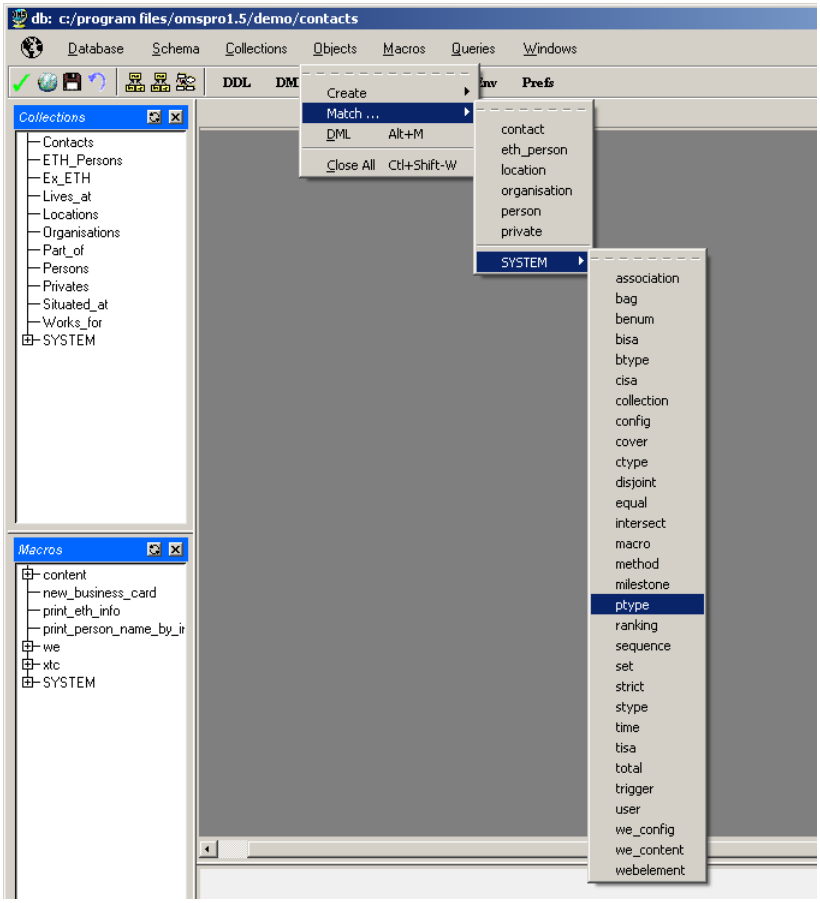


Figure 3.25: System Object Types

object with name **`private`**. The result of the match operation is to display the system object which describes the type **`private`** as shown in figure 3.26.

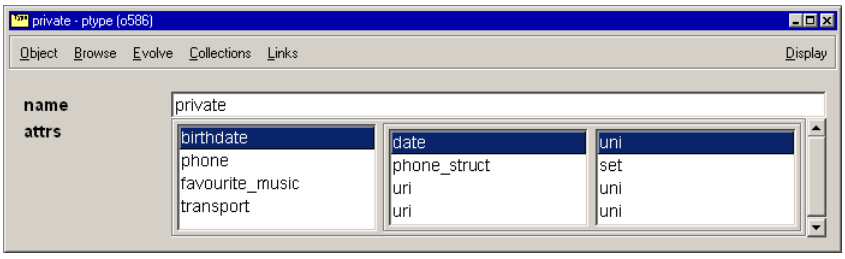


Figure 3.26: System Object for Type **`private`**

Generally, a type is described in terms of its name and its attributes. An attribute has a name, a type and a bulk. If an attribute is single-valued, the bulk is specified as **`uni`**. If it is multi-valued, the bulk is specified as **`set`**, **`bag`**, **`ranking`** or **`sequence`** depending on whether or not it may contain duplicates and is ordered.

Exercise

1. Browse other system objects such as those for collections, subcollections (**cisa** objects) and constraints such as **disjoint**.
2. How is the **partition** constraint over subcollections **Organisations** and **Persons** represented?
3. How are associations such as **Works_for** represented?

3.23 Closing a Session

A database is closed by selecting the **Close** option of the **Database** panel. The **Quit** option causes the OMS system to quit. Note that a **Quit** operation automatically closes any open database before quitting.

Chapter 4

Querying a Database

In this chapter, we describe how a database can be queried. Queries can be used to process the contents of a database – calculating new values or presenting selected values in a convenient form. Often querying may be combined with browsing in that a user first uses a query to locate objects of interest and then browses the database using these objects as a starting point.

OMS has a full query language, AQL, which is based on a general set of operations over collections referred to as a collection algebra. Here we introduce these operations and the query language through some example queries for the contacts database.

A feature of AQL is its generality in supporting operations over not only collections, but also objects and other values such as integers. This generality stems from the orthogonality of the OMS system in that every item within the system is handled as an object – be it an integer, a collection, an object of a user-defined type such as **`person`** or indeed the type **`person`** itself. The results of all queries are displayed as objects.

We first describe how queries can be entered and evaluated in the OMS system and then go on to describe the various operations and how they are expressed in AQL.

4.1 Entering and Evaluating Queries

The AQL interface can be accessed through the **`AQL`** button of the toolbar in the OMS main window. The AQL query panel is as shown in figure 4.1. The query text is entered and the **`Run`** button clicked to evaluate the query. The result of a query is displayed as an object value.

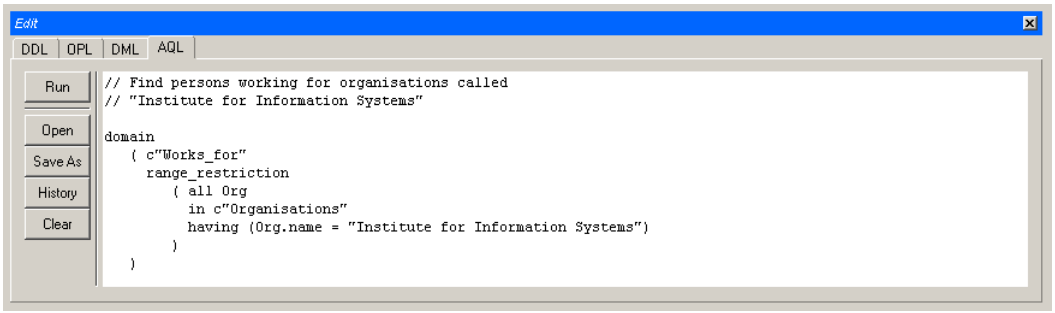


Figure 4.1: AQL Window

If the result of a query is a collection, then a new temporary collection will be generated. The names of temporary collections will be of the form **Temps1**, **Temps2**,... Note that the result of a query may cause new temporary types to be created as well as temporary collections.

After a query has been evaluated, the text of the query will no longer appear in the query window. You can access it again by clicking on **History**. This is useful for three reasons. First, you can easily edit and re-evaluate a query, if you make a mistake. Second, we advise developing complex queries incrementally – especially if you are an AQL novice. You can start with a central part of the query, evaluate it and check the answer. If you are satisfied, you can then extend the query and so on until you have the whole query. Third, if you get the query right, you can study the AQL expression again and take a copy or save it by clicking the **Save As** button!

Clicking on **Clear** will remove any text from the query window. The query panel is closed by clicking on **X** button in the upper right hand corner.

In the following sections, we describe various operations of AQL through some example queries. We propose that you evaluate these queries to see the exact form of the results produced.

4.2 Operations on Values

AQL includes operations over values in terms of arithmetic and Boolean expressions.

For example, the query

```
i"1" = i"2"
```

will display the Boolean value **false** and the query

```
i"1" + i"2"
```

will display the integer value **3**.

The relational operators of Boolean expressions are **=**,**<**, **>**, **<=**, **>=** and, for inequality, **<>**. Complex Boolean expressions may be constructed using the Boolean operators **not**, **and**, **or** and **xor**.

Arithmetic expressions can include the following binary operators: **+**, **-**, *****, **/**, **mod** and, for exponentials, **^**. In addition, there are a number of unary operators: **+**, **-**, **abs**, **floor**, **sin**, **cos**, **atan**, **sqrt**, **log** and **exp**. There is also an operator **float** to convert an integer to a real and an operator **integer** to convert a real to an integer.

AQL can therefore be used, not only to perform database retrieval and update operations, but also to perform general arithmetic calculations, e.g.

```
sqrt(r"3.141" * i"6")/i"3"
```

Note: Although it is recommended to use *type strings* in AQL queries such as **i"9"**, **r"1.414"**, **l"false"** or **d"1976/06/22/12/33/45"**, it is also possible as a shorthand to omit the type information and use the value directly. In this tutorial however we will always use *typed strings* to guarantee unambiguous queries.

4.3 Operations on Objects

AQl also has operations on individual objects which enable the user to retrieve objects as instances of a specified type, to return attribute values of objects and to evaluate methods on objects.

The query

```
t"person"(o"o672")
```

will display the object with object id **o672** as an object of type **person**. The reference to explicit object ids is generally not used, but it shows here how individual objects can be operated on as part of a query.

Alternatively, the above query could be expressed as

```
o"o672":t"person"
```

where ":" is the general type casting operator. In OMS Pro, an object may have many types and the context in which an object is accessed defines a particular view of that object in terms of an associated type. Type casting allows this view to be changed to any other of the object's types.

Type casting can be performed over collections as well as individual objects. For example, the query

```
c"Persons":t"contact"
```

will return a collection of all objects in **Persons**, but viewed as objects of type **contact**. In other words, the resulting collection will be equal to **Persons** in terms of object membership, but differ in terms of membertype.

Attributes of objects can be accessed by means of the "." operator. For example, the query

```
t"eth_person"(o"o672").office
```

returns the string value which specifies the office location of the associated **eth_person** object.

If an attribute is of a *structure type* the fields of the structure can be accessed using the "@" operator. For example, the query

```
t"private"(o"o672").phone@number
```

would return a list of the phone numbers that are stored in object **o672**.

Like attributes, object methods can be evaluated using the "." operator. For example, the query

```
t"private"(o"o672").age()
```

calculates and displays the person's age by calling the method **age** of type **private**. As method **age** takes no input parameters, there is no argument enclosed within the brackets.

Note that these two operations for accessing object properties can also be applied to collections as well as to individual objects. Above we saw an example of "." used to return the office of an individual **eth_person** object. The operator can also be applied to the collection **ETH_Persons** of **eth_person** objects, i.e.

```
c"ETH_Persons".office
```

to return a collection of office values of all members of **ETH_Persons**.

Similarly, methods can be applied not only to individual objects, but also to collections of objects. The query

```
c"Privates".age()
```

will therefore apply the **age** method to all objects in **Privates** and return a collection of result objects each of which is an object with an integer property **years**. Note that if you want to have a list of ages rather than a list of result objects, then you have to extract the years property by using the query

```
(c"Privates".age()).years
```

Note: Applying methods to collections of objects may have unexpected effects. For example, in the above case, a message will also be displayed if the birthdate for some person is unknown! Also, the result would then include the age null since this is the value returned for those with the birthdate unknown.

4.4 Projection of Attribute Values

The projection operator is used to obtain attribute values of individual objects or collections of objects. For example,

```
project (name, office) of t"eth_person"(o"o672")
```

will return an object with the **name** and **office** attribute values of **eth_person(o"o379")**. The resulting object will be of a new system-generated temporary type having only those two attributes.

The query

```
project (name, office) of c"ETH_Persons"
```

will return a collection of objects – also of a temporary type with attributes **name** and **office** – one for each object in **ETH_Persons**.

Exercise

Write a query to find the name of **person(o672)** using a project operator. What is the difference between the result of this query and a query to find the name of **person(o672)** using the "**.**" operator?

How useful is such a project operation? Project operations are widely used in relational database systems to extract columns of interest in tables of information. Many object-oriented systems have adopted some form of project operation – but you might like to consider how useful such an operation is in the context of object-oriented databases.

4.5 Extracting Objects from Collections

There are a number of operators which can be used to extract individual elements from collections based on an ordering. In the case of set collections, these operators select items based on a natural ordering.

For example, the query

```
first c"Persons".name
```

will form a collection of the names of the objects in **Persons** and then select the first according to the natural alphabetical ordering on string values.

Similarly, the query

```
last c"Persons".name
```

would return the name that occurs last in the alphabetical ordering of person names and

```
the i"3" in c"Persons".name
```

the name that appeared third.

In the case of the query

```
first c"Persons"
```

the object in **Persons** with the lowest object id is returned since this is assumed to be the natural ordering of objects.

4.6 Basic Operations over Collections

So far, we have shown how to operate on values and objects within the query language. We now show some examples involving operations over collections. The simplest form of collection query is a collection name. For example, the query

```
c"Persons"
```

will return the collection **Persons**.

The usual so-called set operations **union**, **intersect** and **difference** are available over collections. For example, the query

```
c"Persons" union c"Organisations"
```

will return a collection of objects which belong to either collection **Persons** or collection **Organisations**. What should be the member type of the resulting collection? All those objects in the resulting collection will have properties that are common to the member types of the operand collections. In this case, the set of common properties is equal to that defined by the type **contact** and therefore the result collection contains objects of type **contact**.

In the case that there is no defined type which has exactly the set of common properties, the system defines a corresponding temporary type. For example, the query

```
c"Persons" union c"Locations"
```

returns a collection of objects of a temporary type which has properties common to types **person** and **location** – namely, attribute **photo**. The name of this type will be something like **tmp1**.

In the examples above, the `union` operator is used. Other set-based operations are intersection and difference which may be specified using the keywords **intersection** and **minus**, respectively. Alternative and abbreviated forms are specified in the AQL syntax which is given in Appendix D of this document.

Note that these operations are also defined for collections which have non-set behaviour. For example, two collections which have bag behaviour can be "unioned" thereby generating the collection defined in terms of the "bag-union" of the members of the two collections. Here, we describe operations only in terms of set collections as appear, for example, in the contacts database.

4.7 Counting Objects in Collections

The **count** operator is used to count the number of elements in a collection. For example, the query

```
count c"Persons"
```

returns an integer value giving the number of values in the collection **Persons**.

4.8 Selecting Objects from Collections

A selection operation is provided to select objects from a collection satisfying some specified condition. For example, we might wish to form a collection of professors by selecting all those objects in **Persons** with the title **Prof**. This selection can be expressed by the query

```
all P in c"Persons" having (P.title = "Prof")
```

The selection condition is specified in terms of a predicate and a variable **P** which refers to a member of the **Persons** collection. The predicate is specified as **(P.title = "Prof")**

4.9 Operations over Associations

The collection operators described so far apply to all forms of collections and hence to both unary and binary collections.

In addition, there are a number of operations which only apply to binary collections. These operations utilise the fact that the members of the collections are pair values. We shall introduce these operations by means of an example query.

Assume we want to find those persons working for organisations called "Institute for Information Systems". A query for this example is given in figure 4.2.

```
domain (
    c"Works_for" range_restriction (
        all Org
        in c"Organisations"
        having (Org.name = "Institute for Information Systems")
    )
)
```

Figure 4.2: AQL Query

This query contains two operations – **domain** and **range_restriction** – which are specific to binary collections.

The **range_restriction** operation (which may also be specified by **rr**) performs a selection on a binary collection by restricting the membership of the target collection. For example, in figure 4.2, a selection on the target collection **Organisations** results in a collection of **organisation** objects which have the name **"Institute for Information Systems"**. The **range_restriction** operation then produces a binary collection generated by restricting membership to those pair value members which have second elements in the selected collection of **organisation** objects.

The operation **domain** is applied to the resulting restricted binary collection to obtain the collection of **person** objects which occur as first elements of the member pair values. Thus, the query finds those persons associated through **Works_for** to organisations called "Institute for Information Systems".

Generally, the operation **domain** forms a collection of the left elements of pair values in a binary collection and, correspondingly, an operation **range** is available to form a collection of right elements. Thus, operations **domain** and **range** find the *source* and *target* elements, respectively, of an association.

Note that the **domain** and **range** operations also apply over association instances, i.e. pair values. For example,

```
range(first c"Works_for")
```

returns the organisation object that appears as the second element of the first association instance of **Works_for**.

Corresponding to the operation **range_restriction**, there is an operation **domain_restriction** to perform a selection on a binary collection based on a restriction of the source collection.

To perform a domain or range restriction operation over an association, you may wish to specify the restriction in terms of a constructed set. For example, we may wish to restrict the **Works_for** association to the persons o286 and o287. This can be done by first forming a set collection consisting of these two objects and restricting the domain to this set. The resulting query would be

```
c"Works_for" dr (set [o"o286", o"o287"]:t"person")
```

The expression

```
set [o"o286", o"o287"]:t"person"
```

constructs a set collection with membertype **person**. Note that here the abbreviated form **dr** has been used to specify the domain restriction operation.

These operations over associations prove to be very powerful in querying a database. Frequently, the key to writing a complex query is to focus on the associations rather than the unary collections. Much of the processing in terms of finding related objects and performing selections can be performed on associations.

4.10 Composing Associations

To find associated objects in the database, it may be necessary to compose associations. For example, the query

```
c"Works_for" compose c"Situated_at"
```

will produce a binary collection that associates **person** objects with **location** objects representing the relationship between people and the location of the organisations for which they work.

AQL also includes a **closure** operation for binary collections. For example, the query

```
closure c"Part_of"
```

evaluates the transitive closure of the relation on **Organisations** defined by the **Part_of** association. Thus, it associates an organisation object to all organisation objects within an enterprise of which it is subsidiary and not only those of which it is a direct subsidiary. It is therefore equivalent to repeatedly composing a collection with itself. Note that this only makes sense when the association has similar types of objects in its source and target collections.

4.11 Product and Division

AQL has a product operation **x** which is used to generate binary collections by forming all possible associations of the members of two operand collections. Thus, the query

```
c"Persons" x c"Locations"
```

would return a binary collection with all possible pairings of elements of **Persons** and **Locations**.

The product operator can also be used to construct association instances between values. For example,

```
i"3" x i"4"
```

would generate a pair value with 3 as the domain and 4 as the range. Note that evaluating this query directly in OMS Pro will cause a tuple object to be displayed that has a temporary structured type which is generated on the fly.

The division operator **div** is used to find domain objects of an association that are associated with all elements of a target collection. For example, the query

```
c"Situated_at" div c"Locations"
```

would return the contacts which are situated at all locations in the database.

4.12 Nesting and Unnesting Associations

The **nest** operator forms a nested structure from a binary collection based on grouping the range elements associated with domain elements. For example, the query

```
nest c"Situated_at"
```

would form a binary collection in which the domain elements were contacts and the range elements were sets of location objects. Thus, each member of the resulting collection would be a pair formed from a contact object and the associated set of location objects as determined by the **Situated_at** association.

The **unnest** operation performs the reverse operation by flattening out such a nested structure.

4.13 Map and Reduction over Collections

The **map** operation can be used to apply an operation to each member of a collection. For example,

```
map P in c"Persons" by (P.name)
```

maps each member of the **Persons** collection to its **name** attribute value – forming a collection of the results. It is therefore equivalent to the query

```
c"Persons".name
```

The query

```
map P in c"Persons" by (P.name x P)
```

would construct a collection where each element consists of the name of a person paired with the associated person object.

A **reduce** operator allows aggregate values to be calculated over collections. For example, a query to calculate the sum of all ages of private person contacts is shown in figure 4.3.

```
reduce P in c"Privates"
aggregate A by ((P.age()).years + A)
default 0
```

Figure 4.3: Reduce Query

The reduce operator can be thought of as an accumulator which starts with a default value and, for each element in the collection, calculates a new value for the accumulator based on the previous value and the element.

For example, in the query of figure 4.3, the variable **P** is used to refer to elements of the collection **Privates** and variable **A** to the accumulator. The line

```
aggregate A by ((P.age()).years + A)
```

specifies how the new value for the accumulator **A** is calculated at each stage based on the old value for **A** and the current element **P**. In this case, the new value for **A** is the sum of the age of **P** and the old value of **A**. The default is specified as 0, which means that the initial value of **A** will be 0 and the final value of **A** will be the sum of all ages.

Note that if some birthdates of persons are unknown, then the age value will be calculated as null and an error will occur. To test for the case where a birthdate is unknown, a conditional statement of the form

```
if (P.birthdate=null) then (A) else ((P.age()).years + A)
```

could be used for the aggregate expression of the reduce statement. In this case, the sum of all *known* ages will be calculated. Of course, one must question whether this is what the user really expects.

4.14 Query Expressions

So far we have introduced the main query operators of the OMS query language AQL. Note that while we have introduced the main operators here, we have not provided a complete list. Nor have we described the effect of these operators when not only set collections, but also the other forms of collections – bag, ranking and sequence – are involved. Further details and examples are given under the OMS Pro **Help** information and in the OMS Pro manual. The full AQL syntax is given in Appendix D and a complete definition of AQL is given in [Wue00].

General tips for constructing complex query expressions are to start by focussing on the associations and to work in stages. Queries may be formed from subqueries by introducing variables.

Assume we want to find those persons who work for ETH Zurich. The **Works_for** association is central to this query since it relates persons to organisations. Therefore, we want to restrict its range to ETH Zurich and then take the domain of the resulting collection to obtain the desired persons.

We will therefore perform a range restriction on **Works_for** where the restriction is the collection of objects corresponding to ETH Zurich. We can form this collection with the expression

```
ETH_Zurich := all O in c"Organisations" having (O.name = "ETH Zurich");
```

Here we assign the result to the variable **ETH_Zurich** which can be referred to in further query expressions. Note that the above query expression is terminated with a semi-colon "**;**" since it actually forms the first of a sequence of expressions.

We can then complete the query with the expression

```
dom(c"Works_for" rr ETH_Zurich)
```

Note that this query may not provide the result that we expected. It will only find those persons in the database who are directly associated with ETH Zurich through the **Works_for** association. What we perhaps really want is the set of persons recorded as working either directly for ETH Zurich or for any of its parts. We would therefore have to find not only the organisation for ETH Zurich, but also its suborganisations.

4.15 AQL on Collections

Some operations of the AQL query language are available directly through menu options on collection objects. For example, in figure 4.4, we show the **AQL** menu available on the collection **Contacts**. The operation **count** returns the cardinality of the collection, i.e. how many members it has. The operations **union**, **intersect** and **minus** are the usual so-called set-based operations over collections.

The system generates a list of likely collections to which the various operations may be applied. For example, we can select **ETH_Persons** under **minus** to form the difference between **Contacts** and **ETH_Persons**.

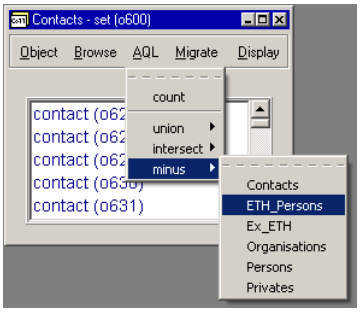


Figure 4.4: Queries on Collections

Note: Whether temporary collections appear in collection menus or not depends on user set **Preferences** as described in Chapter 8. The user can also specify whether they prefer temporary collections to be deleted as soon as they are closed.

4.16 Query Exercise

Write AQL expressions for the following queries on the contacts database.

1. How many people are working for organisations called Computer Science?
2. Find the organisations for which private contacts work.
3. Find all those people who have their home locations recorded in the database.
4. Find the persons who work in the same organisation as Markus Meier.
5. How many people work in the Institute for Information Systems?
6. Find all the people in the Institute for Information Systems who are more than 28 years old.
7. Find ETH Professors living in Zurich.
8. Find ETH Professors who do not work in Zurich.
9. Find those organisations not part of ETH.
10. Find the youngest person in the database.
11. Find the persons in the database who have the same birthdates.
12. Find the total number of people working for ETH Zurich.

13. Find the people who work in the same office as Beat Signer.
14. Find the average age of persons working for ETH Zurich for whom the birthdate is known.
15. Find the organisation with the maximum number of employees.

Chapter 5

Creating a Database

In this chapter, we describe how to create a new OMS database by first creating the database and then defining the schema either by loading data definition statements from a file or working through the graphical schema editor.

Since OMS Pro must also support the prototyping process itself, it is important that a schema can easily be extended or changed, without loss of experimental data. OMS Pro therefore also provides as part of its data definition language DDL, statements for schema updates and these are also described in this chapter.

Note that we deal at this stage only with the structure of a schema and not the operational part in terms of methods, triggers and macros. This is because we believe that you should first create and prototype the database structure before implementing the operations.

5.1 Creating a Database

To create a database, first select the **`Create`** item of the **`Database`** menu of the OMS main window as shown in figure 5.1.

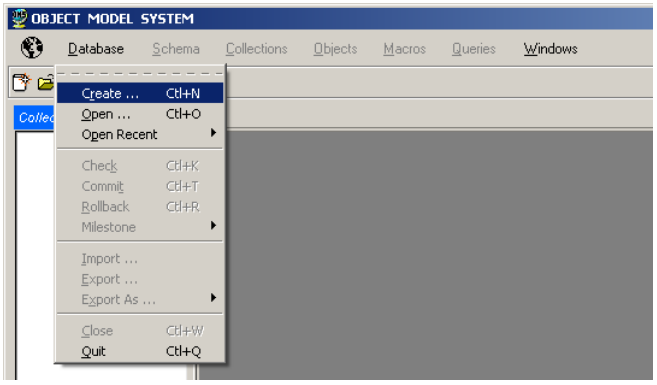


Figure 5.1: Create a Database

Note that if another database is currently open, it must first be closed before creating or opening another database.

A create database dialog window will be opened and you should type the name of your database in the **Database** field as shown in figure 5.2.

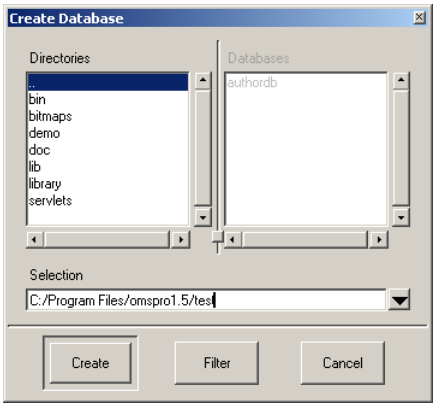


Figure 5.2: Naming a Database

Then click on the **Create** button.

5.2 Loading the Schema

Once the database has been created, the schema can be defined using the OMS schema definition language. A schema definition can either be loaded directly, or first edited in a file and then imported.

To load schema definitions directly, go to the **Schema** menu of the OMS panel and select first **Load** and then **Definitions** as shown in figure 5.3.

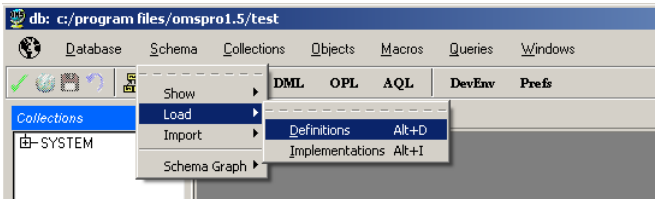


Figure 5.3: Loading a Schema

Figure 5.4 shows the OMS language panel.

Once a schema definition has been typed, it can be entered by clicking on **Run**. The system will display a message to show whether or not the schema definitions are valid and have been loaded. If there are errors, the schema definitions can be displayed again using **History**, then edited and once more loaded.

Note that it is not necessary to load the entire schema at one time. Schemas can be loaded incrementally. If you are unsure about the correctness of your schema, it is a good idea to type the definitions in a file and cut and paste a few definitions at a time into the schema definition window. In this way, it is easier for you to detect where errors lie in your schema. It may also help you to prototype parts of your design. To load a previously stored DDL file, click the **Open** button and select the file from

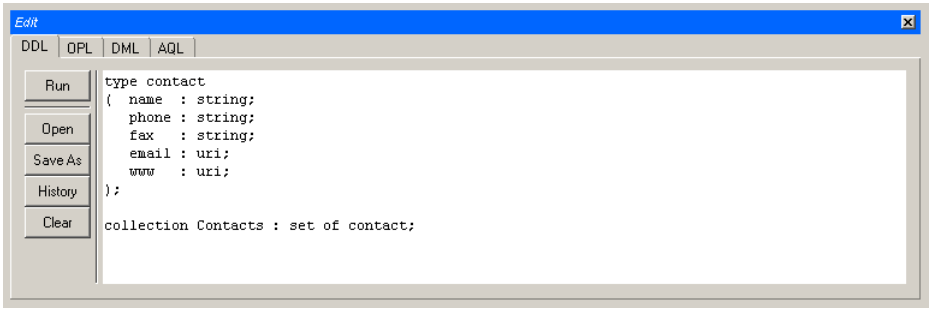


Figure 5.4: Adding Schema Definitions

the file dialog. Even easier is to create and edit the schema directly through the schema graphical editors as described in the next section.

Once a schema has been successfully created, objects can be added to the database as described in the chapter on browsing.

Note that it is possible to specify certain preferences associated with a database to customise not only the presentation, but also certain operational features of the database. There are many different forms of preferences available and we describe them in detail in Chapter 8.

5.3 Schema Graphical Editors

The entire process of creating a database schema can be performed through the graphical schema editors for types and collections. Before creating collections, the appropriate membertype objects must first be created and this can be done through the schema type editor.

By clicking the right mouse button on the type editor panel, a popup menu will appear which gives options to browse, edit and create types. Selection of the **`New Type`** option will cause the user to be prompted for a type name and then the type will be created. A newly created type will be represented by a rectangle with the name of the type in the top right hand corner and the label 'no attribute' in the centre as shown for example in figure 5.5. In this figure, we see that a single type **`person`** has been created, but as yet it has no attributes.

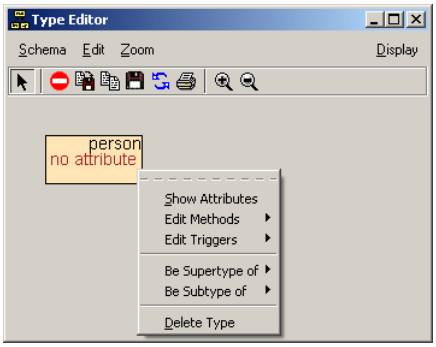


Figure 5.5: Displaying Type Attributes

To define the attributes of a type, you must first expand the type representation to show its attributes. This is done by clicking the right mouse button on the type box to obtain the menu of options shown in figure 5.5. If we select the **`Show Attributes`** option, the type representation is expanded to show the attribute definitions as shown in figure 5.6.

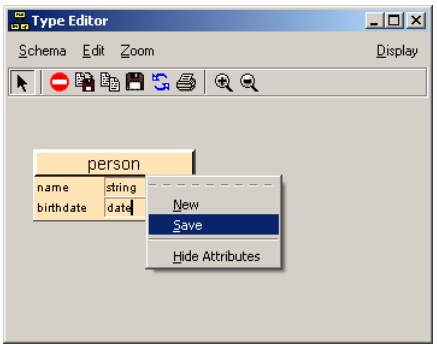


Figure 5.6: Saving Type Attributes

Clicking with the right mouse button on the expanded type box will display the option menu shown in figure 5.6. Selecting the **`New`** item adds new attributes to the type and by simply editing the left and right fields, you can specify the attribute name and type, respectively.

Note that you must select the option **`Save`** if the changes you made to the type definition are to take effect. Then selecting the **`Hide Attributes`** option will return the type representation to its original form – but this time with the new number of attributes specified.

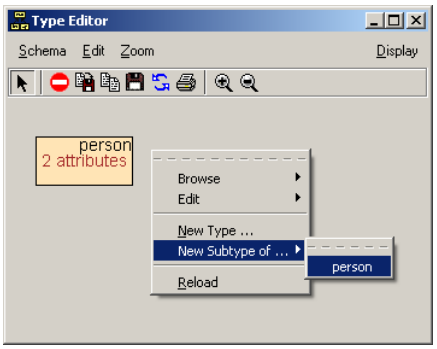


Figure 5.7: Creating a Subtype

Creation of a new subtype can be done through the general type editor menu as shown in figure 5.7. The system prompts with a list of the existing types and the user must select the one for which a subtype is to be created. The system will create a type as before and the associated attributes can be added in the same way. The resulting type graph might look like that given in figure 5.8.

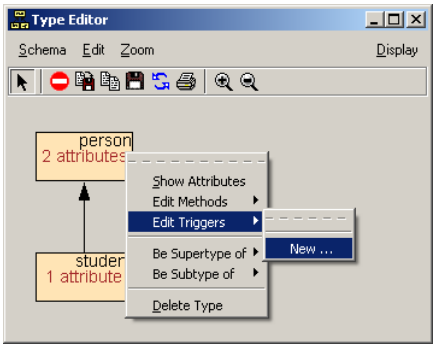


Figure 5.8: Creating a Trigger

So far we have discussed only the structural aspects of a schema. However, types may also have operations associated with them in terms of methods and triggers and these are discussed in detail in the next chapters. However, we note here that these may also be added and edited through the type graph as shown in figure 5.8.

Note: Every type and subtype that you create should have some special associated properties – whether these be attributes, methods or triggers. If you create a subtype and do not define any attributes or operations for it, the objects created for this type may not be handled correctly.

As was shown in figure 5.7, it is also possible to browse and edit types through the popup menu of the type panel. The **`Edit`** option actually is equivalent to performing a **`Show Attributes`** operation on the specified type, thereby enabling the type attributes to be edited as described above. In contrast, the **`Browse`** option provides access to the system object representing the type, i.e. the appropriate **`ptype`** object. The definition of a type could actually be performed by editing the properties of this object directly. However, we advise that it is generally better to edit type definitions either through the graphical type editor described here or through the schema evolution operations of the DDL as described in the next section. These two options provide controlled forms of update and ensure that any existing data is neither lost nor corrupted.

Collections – including associations – can be created through the collections schema graph. As described in Chapter 3, you can select a **`Display`** option so that associations are displayed and not simply collections.

As shown in figure 5.9, collections can be browsed and created by options in the main popup menu of the panel, again obtained by clicking with the right mouse button.

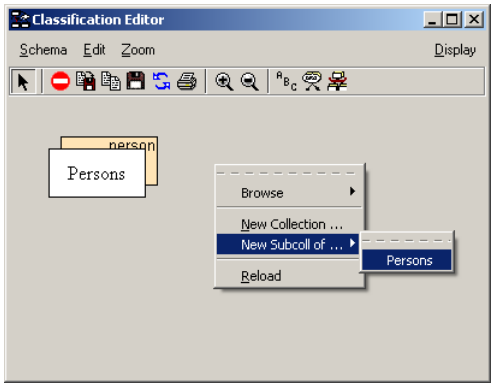


Figure 5.9: Creating a Subcollection

Figure 5.9 actually shows how to create a subcollection of an existing collection **`Persons`**. When a collection is created the user is asked to supply the name and membertype of the collection. In the case of creating a subcollection, only the name of the collection is requested and the membertype is automatically set to be the same as that of the supercollection.

If it is required to create a subcollection of a different membertype, first create the collection by selecting the **`New Collection...`** menu item that can be seen in figure 5.9. Create the collection by specifying a name and a type in the displayed dialog. Note that the specified type has to be a subtype of the collection that will be this collection's supercollection. After confirming the entered data by clicking the **`OK`** button the collection appears in the graphical editor. Clicking on the newly created collection with the right mouse button will display the menu shown in figure 5.10. Selecting the menu item **`Persons`** will convert the selected collection into a subcollection of collection **`Persons`**. The graphical schema editor now displays an arrow from the subcollection to the supercollection as can be seen in figure 5.11.

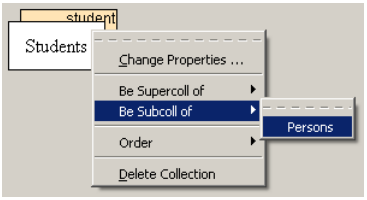


Figure 5.10: Converting a Collection into a Subcollection

To change the name of a collection, there is a **`Change Properties...`** option in the popup menu obtained by clicking with the right mouse button on the collection box. The system will display a dialog as shown in figure 5.11 that prompts for a new collection name. Alternatively the name of a collection can also be changed by clicking on the **`ABC`** icon in the toolbar and then selecting the collection name that should be changed.

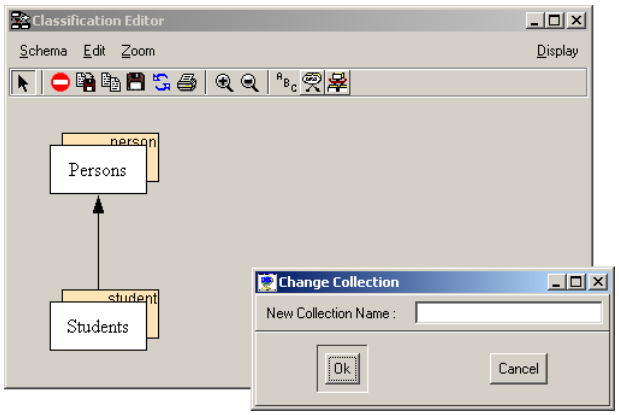


Figure 5.11: Changing Collection Properties

Associations between collections can be created by clicking on the association icon (second from right) of the tool bar at the top of the window. With the left mouse button, you should click on the source collection and then perform a drag to the desired target collection as shown in figure 5.12.

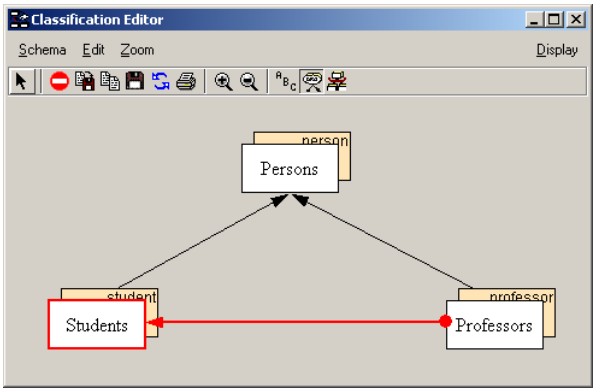


Figure 5.12: Creating Associations

The system will automatically generate a collection name from the composition of the source and target names. This name can be changed by clicking on the **`ABC`** icon of the tool bar and then clicking on the name to be changed. The system will then provide an editable text field as shown in figure 5.13.

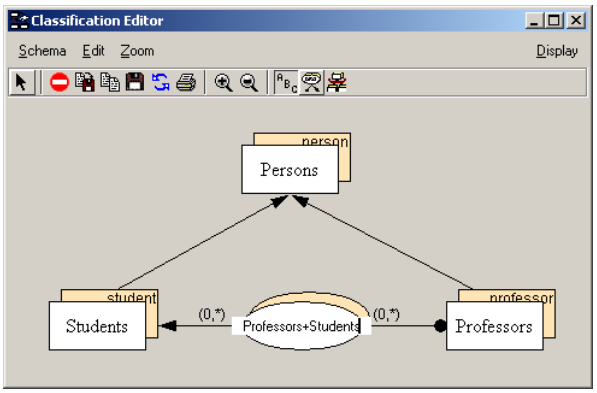


Figure 5.13: Editing Association Name

When an association is created, it is also given the most unrestricted cardinalities **`(0,*)`**. To change these, you can double click on the association links to display the corresponding association object. The cardinalities can then be edited directly.

Through the collection editor, it is also possible to create constraints over subcollections. Consider the case of creating a partition constraint over the subcollections from **`Students`** to **`Persons`** and **`Professors`** to **`Persons`** as shown in figure 5.14.

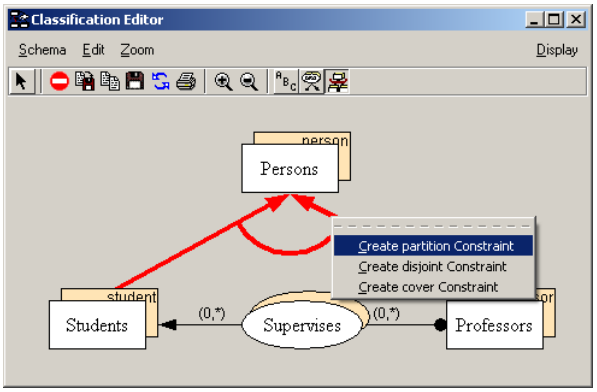


Figure 5.14: Creating a Partition Constraint

To create such a constraint, you must first click on the constraint icon at the right of the tool bar. You then select the subcollection links over which you which to place the constraint by clicking on each of these in turn. As you select each subcollection, it will be highlighted in red and a menu will appear with constraint selections. Note that you will always have available in this menu only the constraints that would make sense. Ignore this menu until you have selected the last subcollection over which you want to place the constraint. In this case, when the menu appears, select the appropriate constraint option.

This completes our description of how to create, browse and edit a schema through the schema type and collection graphs. In the next section, we go on to describe schema evolution operations available in the data definition language DDL.

5.4 Changing a Schema

It is useful to be able to change a schema once a database has been created. This is particularly important during prototyping when one may wish to add or delete attributes, rename them or change their types.

It is always easy in OMS to add new types and collections to the system. Changes to type definitions – and hence to the representation of objects – is more difficult, especially if existing objects are not to be corrupted. OMS therefore provides schema definition commands to change type definitions. These commands to change attributes can be added in the same way as usual schema definitions i.e. by importing them from a file or loading them directly from the schema definition window.

As an example, a new attribute **finger** could be added to type **person** with the schema definition statement

```
type person add finger:string;
```

To later delete this attribute, we could add the schema definition statement

```
type person delete finger;
```

Attributes can also be redefined – changing their name, the type of their values and also whether they are single or multi-valued. For example, if we wanted to change the database so that it is possible to store several Web addresses for a contact, we could change the **www** attribute of type **contact** by

```
type contact change www to www:set of uri;
```

The order of the attributes given in a type definition determines the order in which the attributes are displayed when browsing objects. Sometimes we may therefore wish to re-order attributes in a definition. To avoid any corruption of existing data, it is important that such re-orderings are done through a special **swap** operation of the DDL rather than trying to do it by editing directly the corresponding **ptype** object.

The order of the attributes **name** and **www** in type **contact** could be exchanged using

```
type contact swap name and www;
```

Another change of type definition that can have serious consequences in terms of possible data corruption is the renaming of a type. Again, this should be done through the specially provided operation. To change the name of type **person** to type **human**, the following statement would be used

```
rename type person to human;
```

If we wanted to delete a type, we can use the special **delete** statement. For example, the type **location** could be deleted by

```
delete type location;
```

There are also DDL statements to create, delete, rename and change collections. We do not describe all of these here, but the complete syntax of DDL is given in Appendix B.

5.5 Design Exercise

Develop an OMS database system that could be used as the basis for an Internet store for a furniture company such as IKEA. The database should store information on the various categories of products, customers and their orders. The company also offers a special club (cf. IKEA Family Members) which customers may join.

The database should provide access to the same sort of information on products as can be found in an IKEA catalogue. (You can assume that photographs of the various products would be made available.)

Note that, at this stage, we consider only the data to be stored and not the operations on that data in terms of methods or queries. However, you may wish to consider what methods would be required in such a system.

Develop your system in the following stages:

1. Design an OM graphical schema which identifies the various objects, object roles and associations in your system.
2. Decide on the properties of the various objects to be stored in the database thereby deciding on the object type definitions.
3. Write your OMS schema definition and use this to create an initial database. Enter some data and, if necessary, go back and revise your design until your are satisfied with it.

Chapter 6

Operations in OMS

As briefly mentioned in Chapter 3, OMS supports a set of database operations to perform certain tasks programmatically or to compute new values from the stored data. There are three different kinds of operations, namely *Macros*, *Methods* and *Triggers*. In this chapter we will shortly discuss the properties of each kind of operation and show how they can be accessed within OMS Pro. The details of the implementation of such operations will be given in Chapter 7 where two different languages supported by OMS are presented together with examples.

6.1 Macros

Macros are small applications on top of the database and independent of a particular context. They perform operations, such as assembling collections, accessing meta structures, query and administrate the database, performing database wide consistency checks etc.

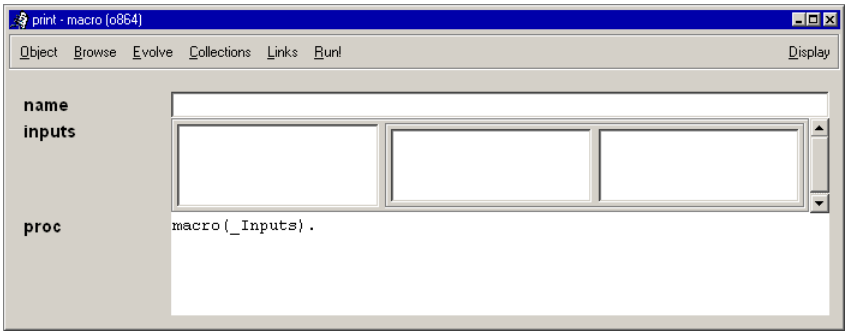


Figure 6.1: OMS Macro Object

As shown in figure 6.1, OMS Pro macros are represented by usual objects in the database as instances of a type **`macro`**. A macro object has a name, a set of input parameters and a macro procedure body. The name of the macro will later be used to display the macro in the **`Macros`** menu and, if displayed, in the macro panel at the lower left corner of the OMS main window. Depending on the list of input parameters that is specified for a macro, OMS will automatically generate a dialog to input all these values whenever the macro is called. Details about the implementation of the macro body are given in Chapter 7.

New macros are created using the **`New...`** option of the **`Macros`** menu. To execute or edit an already defined macro select it from the **`Macros`** menu. Depending on what setting is specified for

the **Macros** option in the **Extra** panel of **Preferences** described in Chapter 8, the macro will either be executed directly or an **Edit** button that allows editing of the macro will be displayed. If however the macros panel is displayed in the OMS main window, executing and modifying macros is even easier. Select the macro to work with from the list. Double clicking with the left mouse button will open the macro object and enable editing, whereas clicking with the left mouse button will execute the selected macro.

6.2 Methods

In contrast to macros that perform operations over the entire database, methods are bound to an object type and therefore are called upon an object instance. Generally, there are three sorts of methods commonly found. Some methods specify operations associated with objects, for example a method to *print* a document object. Methods are also used to compute values associated with an object. For example, a method to calculate the age of a person. Often the computed values are derived from other stored attribute values. For example, the age of a person may be calculated from a stored birthdate value. Methods are also frequently used as an aid to navigation. They may be used to generate a set of objects directly or indirectly associated with an object.

For example, in the contacts database, the method **work_places** of type **person** returns the set of possible work places of a person based on the associations between persons and organisations, and organisations and locations. Rather than navigating through the database from persons to organisations and then organisations to locations, the user can directly go from persons to possible work locations. Note that the system generated links between objects in OMS reduces the need for methods to aid navigation by providing direct access to associated objects.

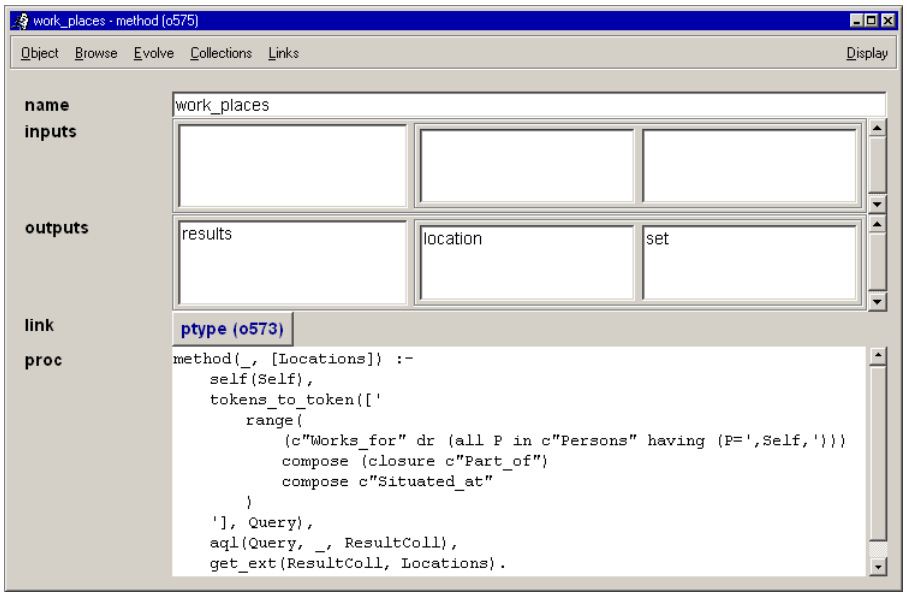


Figure 6.2: Method Object

A method is specified by an object instance of type **method** as shown in figure 6.2. Such a method object stores all information about a method associated with a type. It therefore contains the name of a method **name**, the format of any input parameters **inputs**, the format of any result parameters **outputs**, the link to the associated type **link** and also the method body **method**. Note that the method body is given as a piece of OMS Prolog in this example.

Methods can be created by opening the graphical type editor window as described in Chapter 5. Clicking with the right mouse button on a type box will the display the menu shown in figure 6.3. Selecting the **`New...`** option will create a blank method object and display it in the main panel of the OMS window.

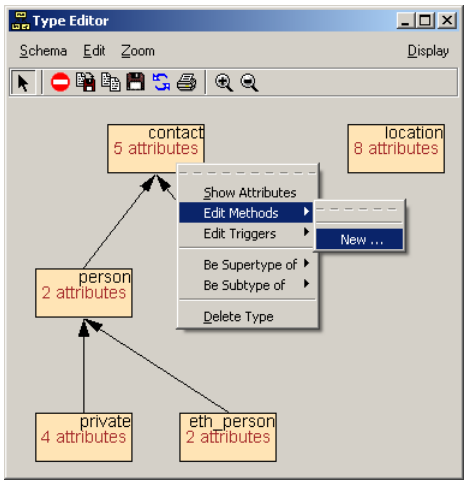


Figure 6.3: Creating a Method Object

The easiest way to locate a previously created method object is through an object of the associated type. As described in Chapter 3, clicking on the "**`->`**" button alongside the method name with the left mouse button causes the method to be evaluated. However, clicking on the method button with the right mouse button displays the corresponding method object.

If there are currently no objects of the type associated with the method, methods can also be accessed through the same menu in the graphical type editor that was used to create the method. If methods are defined for a type the menu shown in figure 6.3 also displays a list of defined methods. Simply choose the name of the method that should be edited from this list to display the method in the main panel of the OMS window. Alternatively, the method object could also be located through object matching as described previously.

As with other objects in OMS, all values of a method object can be changed dynamically at runtime. This means that the implementation of a method can easily be changed and tested without any recompilation and this is beneficial for prototyping.

6.3 Triggers

A trigger associates a command with some event. The command will be evaluated whenever the specified event occurs in an object identified by the scope parameter. The scope argument therefore determines to which objects – or more exactly, which object components – the binding applies. This means, that only a part of an object receives the event.

Here, it is important to understand the dynamic type system of OMS and to take a physical view of the objects within the system: An object is represented by a set of *information units*. In a logical view, an instance of a type is composed at runtime by several information units appropriate to the given type hierarchy. Each of these information units is represented by a triple. It contains the object identifier, a link to a type object and a set of values which corresponds to the set of attributes introduced by that type object.

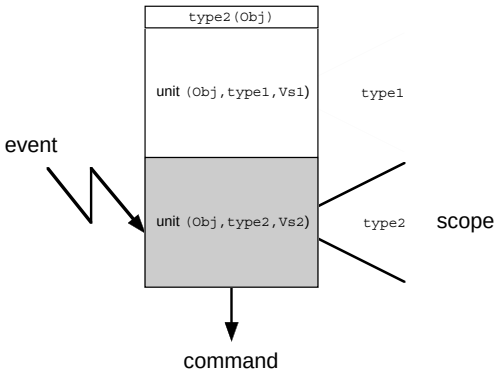


Figure 6.4: Trigger Invocation

In this context, the scope argument of a trigger is defined by a type object. This means, that, similarly to methods, triggers are linked to a type and determine those information units that are bound to the event. This is shown in figure 6.4.

Generally, there are four kinds of events upon which the system may react and fire a trigger: *create*, *update*, *delete* and *commit* events.

- A *create* event is invoked *after* an information unit has been created. It can be used to perform some initialisation operation, such as putting an object in default collections or for maintaining initial consistency.
- An *update* event is invoked *before* an information unit is going to be updated. This could be used for checking input values or for update propagations.
- A *delete* event is sent to an information unit *before* it is actually deleted. This could be used to prevent deletion in the case that the command associated to that event fails, or to perform certain termination actions and cleanup operations before deletion.
- A *commit* event is invoked *before any* information unit is going to be committed. This is used by the system itself for constraint checking and consistency maintenance.

Triggers are, like macros and methods, stored in objects within the OMS system. The graphical representation of such an object of type **`trigger`** is given in figure 6.5.

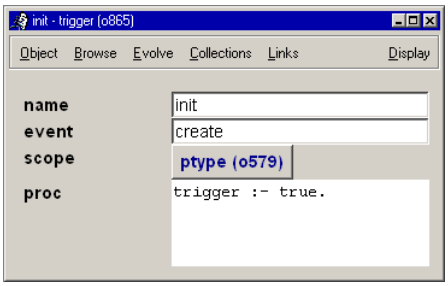


Figure 6.5: Trigger Object

The creation of triggers has already been described in Chapter 5. As mentioned there triggers are created using the type popup menu within the graphical type editor as shown in figure 5.8. Selecting the **`New...`** option from the popup menu will prompt the user to input a name and an event for the new trigger. After this information has been entered, a new trigger object is created and displayed in the main panel of the OMS window.

Trigger objects store a name, an event, a link to a type defining the scope and the implementation of the trigger. Because triggers are invoked by the system rather than the user there is no need for input or output parameters since there is no user interaction. Triggers can however write messages to the message panel or to log files as will be seen later on.

As with macros and methods, triggers can be changed and updated at any point in time without having to store or compile them. Upon the next event the new version is fired and executed by OMS. To browse a previously defined trigger the graphical type editor can be used in much the same way as it is described for methods above. Of course, it is also possible to use object matching to display the triggers defined in a given database. More details on the implementation of triggers will be given in Chapter 7.

6.4 Accessing Operation Definitions

The implementation of an operation can not only be specified and browsed within an operation object, but also through the *schema interface* which provides convenient textual access for both viewing and loading multiple operation definitions. For example, to view all operation definitions go to the **`Schema`** menu of the main OMS window and select first **`Show`** and then **`Implementations`** as shown in figure 6.6.

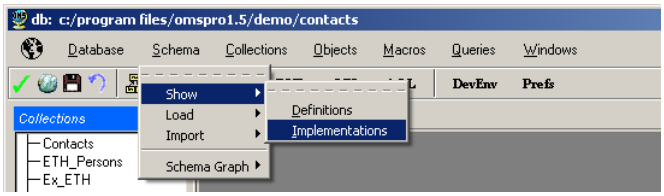


Figure 6.6: Show Method Implementations

The result is to display current method definitions in the operation specification language (OPL) as shown in figure 6.7.

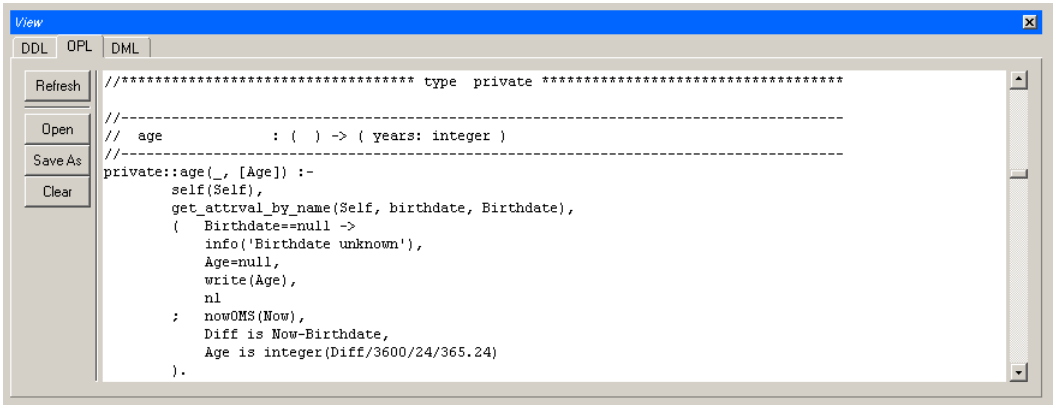


Figure 6.7: Implementations Panel

Method definitions can be loaded interactively in a textual form through the **`Schema`** menu of the OMS main window by selecting **`Load`** and then **`Implementations`**. Alternatively, they can be imported in textual format from an existing file by selecting **`Import`** and then **`Implementations`**. Details of the language OPL are given in Appendix C.

Chapter 7

Implementing Operations

In Chapter 6 we described the three different kinds of operations, *macros*, *methods* and *triggers* in terms of their properties and how they can be created, modified or browsed. In this chapter we will present two different ways to actually implement operations in OMS. Operations in OMS are either implemented in the OMS Interaction Language (OIL) or in OMS Prolog. Since it is beyond the scope of this document to describe those languages in full, we simply aim to give the reader an idea of how operations are specified.

OMS Pro recognises from the format of the implementation whether it is specified in OIL or in Prolog. OIL and Prolog implementations can be mixed freely within the same database. Since in both OIL and in Prolog methods, it is possible to call other methods and macros, it is easy to call operations implemented in Prolog from OIL implementations and vice versa.

Before we get into the specific properties of these two languages we would like to note that it is possible to define utility or helper functions in both OIL and Prolog externally, to take over much of the complex or general processing tasks. Additional OIL procedures can be added to the system using OIL's own library concept to extend the set of available libraries with your own packages. It is however far beyond the scope of this introductory tutorial to discuss the implementation of user-defined OIL libraries. Incorporating external Prolog predicates is very simply as any Prolog files located in the directory associated with an OMS Pro database will be loaded automatically when the corresponding database is opened. This means that you may write your own Prolog predicates to be used in methods of the database. Simply create a file with extension **`.pl`** with your Prolog predicates and save it in the directory of the associated database. For example, for a database called **`contacts`** there will be a directory **`contacts.oms`** and your Prolog file should be stored in this directory.

Note: When working with operation objects in OMS, pressing the keys **`Ctrl`** and **`L`** together will display a small dialog that facilitates locating a position given by line and character number in the source code of the operation. This is particularly helpful in the case of OIL errors where the line number of the offending command is always included in the error message.

7.1 OMS Interaction Language

In this section, we present the OMS Interaction Language (OIL) that can be used to specify operations within OMS Pro. OIL was developed to be a simple, interpreted language that can be used on all OMS platforms — including OMS Pro and OMS Java. Previously, although all OMS systems were based on the same OM object model and used the same languages DDL, DML and AQL for data definition, data manipulation and querying, respectively, the language used to implement methods,

triggers and macros was that of the implementation platform. Thus, method implementations in OMS Pro were written in Prolog and those in OMS Java were written in Java. As a result, migrating a database from one platform to another required re-implementation of all operations.

As a language developed specifically for OMS systems, OIL combines general characteristics of a programming language such as Java with features specifically oriented to working with OMS databases and hence with constructs of the OM object model.

OIL is an imperative language with a Java-like syntax. To keep it as simple as possible, only the essential control constructs are provided. Further, OIL was designed to be strongly based on a library concept, enabling the functionality to be extended easily. A basic set of libraries is included in the distribution of OMS and users can easily add their own. In addition, while OIL is intended as a platform independent implementation language, we also wanted application developers to be able to specify complex functions on the system level using the API of the native language. We therefore provide call-transparency that enables routines of the native language to be called easily from within OIL.

It is beyond the scope of this tutorial to describe all features of the OIL language and also the system libraries. Further details can be found under the **Help** menu of the OMS Pro system. Select the **OMS-Pro** option under the **Quick API** menu item. The option **oms:lang:oil:olib** will take you to documentation of the OIL system libraries.

Macro, method and trigger definitions with OIL implementations can also be included in OPL implementation files and loaded into the database. Note that in this case there is a special syntax required to indicate OIL implementations. The general forms of OPL statements are shown in figure 7.1.

```
oil::meth(type-name,'OIL-method-implementation').
oil::trig(type-name,'OIL-trigger-implementation').
oil::macro('OIL-macro-implementation').
```

Figure 7.1: OPL for OIL Implementations

The entire text of the OIL implementation should be enclosed in single quotes and presented as an argument. Possible single quotes within the OIL procedure must be escaped with the backslash character (\). The method and trigger implementations are specified by two arguments—the name of the type to which they are attached and the OIL implementation text. In the case of macros, there is only one argument—the OIL implementation text.

OIL Routines

OIL is a language specifically developed to implement macros, methods and triggers in OMS and therefore "programs" consist of only one routine. Although it is possible to call other routines from within a routine and also to implement library routines, there is no possibility to assemble larger programs from multiple OIL routines. Essentially, the "structure" of the application system comes from the OMS database and its objects. Generally, a method implementation in OIL is of the form shown in figure 7.2.

Macros can have zero or more input parameters, whereas a method additionally can have zero or more output parameters. Triggers have neither input nor output parameters. Each parameter is specified in terms of a type and a name e.g. **person result** to indicate that the parameter has name **result** and will be of type **person**.

For example, in the case of the **employees** method of figure 7.8, the first line of the implementation is given as

```
operation-ident ( input-param-list ) ( output-param-list )
[ using library-list ]
{
   body
}
```

Figure 7.2: General Form of an Operation Implementation in OIL

```
employees()(person-set result)
```

specifying that the method has no input parameters and a single output parameter **`result`** of type **`person-set`**.

For those familiar with other programming languages such as Java or Oberon, the fact that all methods may return any number of results, is likely to be the most distinguishing feature of OIL. Although this generality is an integral part of the operational framework within the OMS Pro system, it also means that OIL methods cannot be called as functions and have there result value directly assigned to a variable. There is however one exception if a method has *exactly* one return value it can be called inline as a function by omitting the second pair of parentheses. These two approaches are compared in figure 7.3. Assume that variable **`p`** is a reference to an object of type **`private`**.

```
integer age;                    integer age := p.age();
p.age()(age);
```

Figure 7.3: Two Possibilities to Call a Method

The optional **`using`** statement specifies libraries to be imported. System libraries are provided to support basic functions such as access to objects and collections, query processing and output handling. Import names can also be abbreviated by specifying an alias with the **`as`** keyword. For instance, the statement

```
using myStuff.myLibs.Networking as net
```

will enable routines from the **`myStuff.myLibs.Networking`** namespace to be used with the much shorter **`net`** prefix. Examples of the use of libraries are shown in the methods examples later in this section.

The body implementation contains statements terminated by semicolons. The basic language statements of OIL are presented in figure 7.4. Note that expressions used in **`if`**, **`while`** and **`for`** statements can optionally be placed in brackets to enhance readability, but this is not required.

```
VarDecl ::= Type Var [ := Exp ]
Assign  ::= Var := Exp | Exp := Exp
If      ::= if BoolExp { Stmts } [ else { Stmts } ]
While   ::= while BoolExp { Stmts }
For     ::= for [ integer ] Var [ := IntExp ] to IntExp [ step IntExp ] { Stmts }
ForEach ::= foreach [ Type ] Var in ( BulkExp | ListExp ) { Stmts }
Call    ::= [ ( Lib | this | macro ) . ] OpName ( [ InArgs ] ) ( [ OutArgs ] )
```

Figure 7.4: OIL Statements

OIL supports all base types of OMS, as described under the **Values** option of the **OMS Pro Quick Help** menu. To assign values within OIL code, *typed strings* are used. A typed string is a character followed by a string enclosed in quotes. For instance, the typed string **c"Persons"** denotes the **Persons** collection in OIL. Note, that typed strings have already been presented and used in Chapter 4. Similar to AQL, the use of typed strings is optional for values of type **boolean**, **integer**, **real**, **string** or **value**.

Since OMS also supports the notion of *structure types* as presented in Chapter 2, OIL further offers language constructs to work with values of these types. To access for instance elements of binary collections or associations it is required to work with values of the according structure type. Structure types can either be defined in the schema of the database using the **struct** syntax or they can be declared *inline* in the implementation body of an OIL operation. The syntax to define a variable of structure type inline is given below in figure 7.5.

```
StructureVarDecl ::= < FieldDecl , FieldDecl { , FieldDecl } > Var
FieldDecl        ::= fieldType [ fieldName ]
```

Figure 7.5: Inline Declaration of Structure Types

If a variable is declared to be of a structure type, access and assignment to individual values is handled by specifying the field name or the field position after the "**@**" operator. If no field name has been specified in the declaration OIL will automatically label the fields with their position starting with **0**. Values of a structure type are of the form

```
<: Value , Value { , Value } :>
```

It is also possible to nest structured types as the examples in figure 7.6 shows. Note also how such values can be accessed.

```
< <string, integer> street, <integer, string> city > address;
address := <: <: "Princes Street", 927 :>, <: 93000, "Edinburgh" :> :>;
<integer, string> city_struct := address@city;
string street_name := address@street@0;
```

Figure 7.6: Working with Structure Types

Finally a brief description of operators used in OIL is given in figure 7.7.

Evaluating Queries

After having described the syntactic features unique to OIL, we will focus in the remaining part of this section on the interaction between operations and the OMS system. We begin by describing how operations can be implemented in OIL using AQL queries. Using queries in operations provides a very simple way of retrieving data from the system without having to write a lot of code. Queries however have to be parsed and interpreted by the system which can be time-consuming and little efficient. Implementing operations by accessing objects directly on the other hand is more complex, but the performance of the execution increases. To illustrate these two approaches to implementing methods we will present the same method implemented in both ways. Suppose, a method to retrieve all employees of an organisation was to be added to type **organisation**. The query that would compute this set of employees is very straightforward.

```
dom(c"Works_for" rr (all O in c"Organisations" having (O=self)))
```

	Operator		Result	Description
Number	> < >= <=	*Number*	*Boolean*	number comparison
Any	== !=	*Any*	*Boolean*	equal, not-equal
Boolean	| &	*Boolean*	*Boolean*	boolean operators
Text	^	*Any*	*Text*	string concatenation
Number	+ - * div mod	*Number*	*Number*	arithmetic operations
ObjRef	.	*Attr/Meth*	*Any*	attribute or method selector
Struct	@	*Name/Pos*	*Any*	structure field selector
CollRef/List	[*Index*]		*Any*	collection or list member selector
Any	:	*Type*	*Any*	type cast
Any	instanceof	*Type*	*Boolean*	type check
Any	in	*BulkRef/List*	*Boolean*	membership test

Figure 7.7: OIL Operators

Of course, during execution time, the word **self** must be replaced with a reference to the organisation object on which the method is executed. In figure 7.8 the OIL implementation of the **employees** method using query evaluation is shown.

```
employees()(person-set result)
using Aql
{
   Aql.evalQuery("dom(c\"Works_for\" rr
      (all O in c\"Organisations\" having (O=self)))", this)(result);
}
```

Figure 7.8: Implementation of the **employees** Method

The first line of code is the method's header which is used to specify the name and any input or output parameters. As can be seen the method does not declare any input parameters, but has an output parameter of type **person-set**. In OIL the suffixes **-set**, **-bag**, **-ranking** or **-seq** can be appended to any type to that the declared variable holds a collection of the specified bulk type. After the header of the method the **using** statement is used to import the **Aql** library. This is required since the intention is to compute the set of employees using a query, which is done by the only actual line of the method. Calling the procedure **evalQuery** from the library **Aql** evaluates the query and retrieves the appropriate employees from the system.

This procedure **evalQuery** has the general form

evalQuery(*queryString*, *selfReference*)(*resultVar*)

The first input parameter specifies the AQL query to be evaluated as a string value. As we have seen, we may want to include a reference to the object on which the method is being executed within a query string. In OIL, this is done by using **self** within the query string to indicate self references. The second input parameter of **evalQuery** specifies what should be substituted for these self references at evaluation time. If a reference to the current object is to be substituted, this is specified by the parameter value **this** as shown in the example of figure 7.13. If no self references are included in the query text, the second input parameter should simply be **null**. Note that since the substitution of self references within queries is based on an explicit substitution parameter, any other reference variable could be supplied to substitute arbitrary objects and values for occurrences of **self** within a query string.

Accessing Collection Members

As mentioned before it is also possible and more efficient to access the collections and objects of the database directly, instead of using queries. To illustrate the differences and introduce further features of OIL we will show here how the **employees** method could be implemented alternatively. The complete code of this second implementation is given in figure 7.9.

```
employees()(person-set result)
using Collection as coll
{
   result := temp person-set;
   foreach works_for_struct wf_elem in c"Works_for":works_for_struct-set {
      if (wf_elem@employer == this) {
         coll.add(wf_elem@employee, result)();
      }
   }
}
```

Figure 7.9: Accessing Collections Directly

Accessing collections directly is more low-level than letting the system evaluate a declarative query. The main difference between a query and an explicit implementation is that the query describes *what* should be computed, whereas the explicit implementation expresses *how* it should be computed. Hence, when implementing such an operation the programmer rather than the query engine has to make the decision what operations to use and in which order to apply the necessary operators. While this leaves a lot of possibility for optimisation, it also means that the programmer has to work at the level of collection elements. Thus, iterations over collections for instance have to be implemented explicitly using the **foreach** statement of the OIL language as shown in the example. Further, this method implementation also shows how to work with binary types and values through the use of structure types such as **works_for_struct**.

As the example shows, collections can not only be iterated. It is also possible to create new collections and to add elements to these collections. The first line of code in the method body for instance illustrates how a collection, or more specifically a set, of person objects can be created in OIL. Note the use of the **temp** keyword to create a temporary collection that will not be inserted into that database and is only used to calculate the result of this operation. To create persistent objects (including collections) the **new** keyword is used in OIL. In the **foreach** loop, the procedure **add** from the **Collection** library is used to insert an element into the collection of person elements. Browse the API documentation, available from the **Quick API** option of the **Help** menu, to display an overview over the OIL libraries distributed with OMS Pro and the procedures they offer.

Accessing Object Properties

To illustrate how object properties such as attributes and methods can be accessed we implement another OIL operation that uses these features of the language. Given the name of an organisation recorded in the database, the macro **print_staff** shown in figure 7.10 will print a list of all employees working for this organisation to the message panel.

The macro has one parameter of type **string** which is the name of the organisation whose employees should be printed. Two libraries are used by the macro — **Object** which is used to retrieve single objects from the database and **Out** providing basic output procedures. The first line of the macro's body is a call to the **getByName** procedure of the **Object** library. This procedure is used to match for an object by specifying the desired type, the name of an attribute and a value for this attribute. If

```
print_staff(string org_name)()
using Object, Out
{
   organisation org;
   Object.getByName(t"organisation", "name", org_name)(org);
   if (org != null) {
      person-set employees;
      org.employees()(employees);
      foreach person e in employees:person-set; {
         Out.log(e.title ^ " " ^ e.name ^ ", " ^ e.phone)();
      }
   } else {
      Out.dialog("Organisation " ^ org_name ^ " not found!")();
   }
}
```

Figure 7.10: Implementation of **print_staff**

an object matching the given criteria is found it will be returned, if not **null** is returned. The consecutive **if** statement checks if the organisation has been found in the database and proceeds only if so. Otherwise the macro uses procedure **dialog** from library **Out** to display a non-modal dialog with an **OK** button that informs the user that the specified name did not match any object.

If however the organisation was found in the database the macro calls the previously defined method **employees** on the organisation to retrieve the set of persons working for the organisation. Generally methods can be invoked on objects using the syntax

ObjRef.MethName({ *input-param-list* })({ *output-param-list* })

As described at the beginning of this section, method with *exactly* one return parameter can also be called inline as functions with only one pair of parenthesis as the call to **org.employees()** illustrates.

The statement in the body of the **foreach** loop demonstrates how attributes of an object can be accessed. The loop iterates through all employees returned by the **employees** method. Using the **writeln** procedure from the **Out** library the title, name and phone attribute is printed to the message panel. Note also that OIL uses the caret symbol (^) to concatenate strings.

Hence, access to attribute values of an object is done using the syntax

Var := ObjRef.AttrName

7.2 OMS Prolog

When programming in OMS Prolog, each operation implementation is specified in terms of a single Prolog rule. The rule can be arbitrarily complex by including *or*-conditions within the rule. Within rules, the programmer may wish to evaluate queries or access attribute and collection values and we give examples to show how this is done through general OMS-defined predicates, hence the name *OMS Prolog*. Further, we show the use of OMS-defined predicates for processing date values and also for generating dialogue boxes through an OMS interface to the scripting language TCL.

As discussed in Chapter 6 operations can be defined using the Operation Specification Language (OPL). In contrast to the OPL definitions given in the previous section, there is a slightly different

syntax when operations are implemented in OMS Prolog. The required statements to specify macro, method and trigger implementations are given in figure 7.11

```
          :: macro-ident ( input-param-list ) :- body .
type-name :: method-ident ( input-param-list, output-param-list ) :- body .
type-name :: trigger-ident :- body .
```

Figure 7.11: OPL for OMS Prolog Implementations

When method and trigger definitions are given in textual OPL form, it is necessary to specify to which object type they belong. The method rule therefore has the extended form including the name of the type before the "`::`" operator.

Generally, an operation rule is of the form shown in figure 7.12. Input and output parameter lists can be specified based on the type of the operation.

```
macro | method | trigger [ ( input-param-list [ , output-param-list ] ) ] :-
    condition-1 ,
    condition-2 ,
    ...
    condition-n .
```

Figure 7.12: General Form of an Operation Rule

Input and output parameters are specified as Prolog lists. In the case that there are no input or output parameters required by the operation, the parameter lists may be specified as the empty list **`[ ]`**. For example, the head of a method rule may be given as

```
method([ ],[ ])
```

which, in effect, means that there are no input or result parameters.

As indicated in figure 7.12, the body of a method rule is specified as a set of conditions each of which must be satisfied. A condition may be a statement expressed in terms of a general Prolog predicate or an OMS system-defined predicate for accessing OMS objects and values. Statements may be combined to form complex conditions using an "**`or`**" operator which is represented as "`;`". For example,

```
( condition-1 ; condition-2 )
```

states that either *condition-1* or *condition-2* must hold.

Having described the general form of operation rules, it is now time to introduce some examples of specific rules and OMS system-defined predicates. We start by showing how a operation can be defined in terms of a query and then go on to show examples where individual OMS objects and values are accessed.

Evaluating Queries

Operations to return objects or values can be specified as queries. One special OMS predicate evaluates query expressions. The form of the predicate is

```
aql(+Query,-Type,-Object)
```

The predicate has three arguments – a query expression, the type of the result and the result object. Recall that collections are also objects and hence a query which returns a collection actually returns a collection object.

In specifying the form of the predicate, we use the convention that a '+' indicates arguments which should be instantiated when a predicate is evaluated and a '-' indicates arguments which should not be instantiated. If an argument is not instantiated, evaluation of the predicate will attempt to find an instantiation for the argument such that the predicate evaluates to **`true`**. Simply, this means you can regard the arguments with a '+' as inputs and those with a '-' as outputs. If an argument is preceded with a '?', it indicates that the argument may or may not be instantiated.

As a simple example, the predicate

```
aql('c"Persons"',_,Result)
```

will evaluate the query

```
c"Persons"
```

and the variable **`Result`** will be instantiated to the resulting collection object. Note that variables always begin with an uppercase letter. Any string atoms which start with an uppercase letter or contain spaces or non-alphabetic characters must be included in single quotes. This means that if we adopt the convention that names of collections begin with an uppercase letter, then they must be enclosed in single quotes when appearing in a method. The underscore in the above example is used to indicate that the actual type of the result object is not of interest.

The predicate

```
aql('all P in c"Persons" having (P.title="Dr")',Type,Result)
```

will evaluate the selection query and the variable **`Result`** will be instantiated to the resulting collection object and the variable **`Type`** to its bulk type.

It is frequently the case that the query expression in a method is parameterised in the sense that, at evaluation time, a particular object or value forms part of the query expression. Typically, this may be a reference to the object on which the method is evaluated. To retrieve the reference to the current object within a method the **`self`** predicate can be used. Evaluating **`self(Self)`** will "assign" the current object to the variable **`Self`**.

For example, consider the **`work_places`** method of type **`person`**. This method should return the set of all location objects associated with the organisations for which a given person works. Thus, the person in question is specified by **`Self`** — a reference to the particular **`person`** object on which the method is evaluated.

The method rule for **`work_places`** is shown in figure 7.13.

The head of the rule specifies that the method has a single result parameter **`Locations`**. The first statement is a predicate used to generate a query expression from a list of string tokens. This is required because the query expression must be generated at evaluation time when the reference to **`Self`** will be replaced by the given **`person`** object. The resulting query expression is given by the variable **`Query`**.

The second statement is an **`aql`** predicate to evaluate the query and the resulting collection of **`location`** objects is given by **`ResultColl`**.

```
method([],[Locations]) :-
    self(Self),
    tokens_to_token(['range((c"Works_for" dr (all P in c"Persons" having
                  (P=',Self,'))) compose c"Situated_at")'],Query),
    aql(Query,_,ResultColl),
    get_ext(ResultColl,Locations).
```

Figure 7.13: Method **work_places** of type **person**

As the result returned by the method, we want to have a set of **location** objects rather than a collection object containing these objects. The third statement is used to access the extent of the result collection which is the required set, rather than the collection itself. The general form of this predicate is

```
get_ext(?Object,?Extent)
```

with arguments for the collection object and the extent which is the members of the collection.

Accessing Collection Members

In the previous section, we described how operations can be implemented using queries. This form of implementation is particularly useful as a high-level specification. When the database system is later implemented on top of its chosen platform, this query specification can be implemented in the appropriate programming language (and/or query language).

However, rather than implementing methods as queries, it is possible to perform processing by accessing collections and their member objects directly. Generally, this is more efficient for processing – but it is also a lower-level of specification. To show an example of how members of collections can be accessed directly, we present an alternative implementation of the **work_places** method in figure 7.14.

```
method([],[Locations]) :-
    self(Self),
    get_coll(Collection1,'Works_for',_,Ext1),
    get_coll(Collection2,'Situated_at',_,Ext2),
    findall(L,(member((Self,O),Ext1),member((O,L),Ext2)),LocationList),
    list_to_set(LocationList,Locations).
```

Figure 7.14: Alternative Implementation of Method **work_places**

The first two statements use the OMS system-defined predicate

```
get_coll(?Obj,?Name,?MemberType,?Ext)
```

to set variables **Ext1** and **Ext2** to the member sets of binary collections with name **Works_for** and **Situated_at**, respectively. Note that the names of the collections are enclosed in quotes so they are not interpreted as variable names.

The Prolog predicate **member** predicate is used to check for membership of items in sets. Both **Ext1** and **Ext2** will be sets of pair values since they are the extensions of binary collections. We can therefore use

```
member((Self,O),Ext1)
```

to find an object id of an associated organisation object **O** and then

```
member((O,L),Ext2)
```

to find an object **L** associated with organisation **O**. However, what we want to find is not a single possible work place, but all possible work places. That means that we have to find all possible instantiations of **O** and **L**. This can be done by using the Prolog predicate **findall** which has the general form

```
findall(?Template,:Goal,?List)
```

and can be read as "find all instances of *Template* such that *Goal* is satisfied and return those instances in *List*". **Template** is often a single variable, but can take more complex forms. **Goal** is a Prolog statement – simple or complex. The ':' is used to indicate an argument which must itself be a Prolog goal or clause.

Consider the **findall** statement of figure 7.14. The template consists of the variable **L** which is to be instantiated to location objects. All possible instantiations such that the goal is satisfied will be returned in the list given by **LocationList**. Note that **findall** may return duplicates since it is possible that a single instantiation of a variable may lead to goal satisfaction in more than one way. The return argument is therefore a list rather than a set. The goal specifies the single condition for a location to be a possible work place of a person as described previously.

The final statement of the method converts the list of locations returned by the **findall** statement to a set, i.e. it removes any duplicates. The resulting set is returned by the method.

Adding and Removing Objects

Of course, it is also possible to add or delete objects from collection. These operations are generally implemented by triggers that are executed on creation or deletion of an object. To add objects to a collection the system defined

```
add_obj_to_coll(+Obj,+Coll)
```

can be used. As shown in figure 7.15, we first get the appropriate collection and then add the object which receives the event – denoted by `Self` – to that collection.

With this configuration, the creation of an instance of type **person** invokes two triggers which add the person object to both collections **Contacts** and **Persons**. This happens because two information units have been created. However, in the case that a contact object which already exists is *dressed* with type organisation, only one command is fired since only one information unit, namely that for **organisation**, has been created and as a result the object is added to collection **Organisations**.

Note that there is also the system defined predicate

```
del_obj_from_coll(+Obj,+Coll)
```

to remove an object from the given collection. This predicate, for instance, could be used in a trigger bound the the *delete* event to remove an object from all collection is has been a member of.

```
contact::init :-
    self(Self),
    get_coll(C,'Contacts',_),
    add_obj_to_coll(Self,C).

person::init :-
    self(Self),
    get_coll(C,'Persons',_),
    add_obj_to_coll(Self,C).

organisation::init :-
    self(Self),
    get_coll(C,'Organisations',_),
    add_obj_to_coll(Self,C).
```

Figure 7.15: Trigger Implementations

Accessing Attributes

We now describe the implementation of method **age** of type **private**. This example shows how to access attribute values of objects and also how to process date values. The method rule is given in figure 7.16.

```
method([],[Age]) :-
    self(Self),
    get_attrval_by_name(Self,birthdate,Birthdate),
    ( Birthdate == 'null' ->
      info('Birthdate unknown'),
      Age = 'null'
    ; nowOMS(Now),
      Diff is Now - Birthdate,
      Age is integer(Diff/3600/24/365.24)
    ).
```

Figure 7.16: Method **age**

The OMS predicate **get_attrval_by_name** is used to access attribute values of objects and has the general form

```
get_attrval_by_name(+Object,+AttrName,?AttrVal)
```

Given an object and an attribute name, **AttrVal** is set to the current value of that attribute. The predicate may also be used to check whether an object attribute has a specified value by instantiating all three arguments. For example, assume that variable **Object** is instantiated to some given object of type **contact**. Then the predicate

```
get_attrval_by_name(Object,name,'ETH')
```

would evaluate to **true**, if that object had the value 'ETH' for attribute **name**.

The first statement of the method rule of figure 7.16 therefore sets **Birthdate** to the value of attribute **birthdate** of the **person** object on which the method is evaluated.

The second condition, which is enclosed in brackets, is a complex condition equivalent to an *if ... then ... else* statement. Either the birthdate value is null in which case a message is printed and the value **null** returned, or, the age is calculated from the current date and the birthdate. This is expressed as a complex condition of the form

```
( condition-1 -> condition-2 ; condition-3 )
```

which can be read as *if condition-1 then condition-2 else condition-3*.

The statement of *condition-1* checks whether **Birthdate** is equal to **null**. If this is true, the system then proceeds to *condition-2*. The first statement of *condition-2* is an OMS predicate for a printing an information message. The second statement of *condition-2* sets the value of variable **Age** to **null**.

If *condition-1* is not satisfied, the system tries to satisfy *condition-3*. This means that the birthdate is not equal to **null** and so the age is calculated. The statement **nowOMS(Now)** reads the current date and sets variable **Now** to this value. Note that while date values have many possible presentation formats, as specified under OMS Preferences, they have a single internal format as a number of seconds. The second statement of *condition-3* calculates the difference between **Now** and **Birthdate** and this is then converted to an integer number of years and **Age** set to this value in the third statement.

Invoking Methods

The operations presented in this chapter so far are relatively simple and they use only a few operations of the OMS Pro Application Programming Interface (API). API reference lists are available through the **Help** menu of the main OMS Pro panel. Through this menu, you can access all of the API or subsets related to particular topics. While the complete list of predicates is very large, it is important to realise that for general method programming a relatively small subset is required and one quickly learns those required for basic operations of creating and accessing objects and their properties.

Another important point to note is that writing method implementations in OMS Pro as single rules is rather limited. For complex methods, the use of many 'or' and 'if' constructs in a rule leads to situations where it is very difficult to understand methods and to ensure that the logic is correct. It is possible to keep to small, well-structured methods by using two general techniques. The first is to remember to think object-oriented and, rather than having large methods, try to distribute functionality by having methods call other methods. This can be done using the **send** predicates of the API.

For example,

```
send(Obj, print, person, [], _)
```

would send the message **print** to the object **Obj** of type **person** with an empty list of input arguments. The method **print** would then be evaluated on object **Obj**.

Generating Output

As a simple example of how to generate output from an operation, we define a macro **print_collection_names** which gets and prints the names of all collections stored in our database.

In figure 7.17, the construct

```
::print_collection_names([]) :-
    \+ (get_coll(_,Name,_),
        log(Name), nl,
        fail
    ).
```

Figure 7.17: Macro Implementation

```
\+ (body, fail)
```

uses backtracking in order to iterate over the whole database. The predicate **log** is used to print output to the OMS message panel. It is however also possible to display simple message boxes to interact with the user. The system defined predicate

```
info(+Message)
```

will generate and display a dialogue box with the given message and an **OK** button. Note that this predicate has already been used in the **age** example shown in figure 7.16.

7.3 Implementation Exercise

Work through the following exercises using either OIL or OMS Prolog.

1. Write a query to evaluate for a given organisation the number of staff members recorded in the contacts database. Then add a method to type **organisation** corresponding to this query. Note that you can add the method to the object by using the graphical schema type editor described in Chapter 6.
2. In the contacts database, change the implementation of the **work_places** method of type **person** to the implementation given in figure 7.14.
3. Add a method to type **organisation** to find the name of all organisations which are directly above it in the *part-of* hierarchy. Implement the method without using a query.
4. The **work_places** method described in this chapter will only find the possible work locations of a person by considering an organisation associated with that person through the **Works_for** association. However, it is possible that we may not have specific information about the location of that part of an organisation, but we do have information about the locations of its parent organisation. How could you rewrite the the **work_places** method so that, in the case that an organisation has no associated locations, it will try to find locations of a parent organisation?
5. Add the triggers described above to the contacts database and try them out!
6. Write triggers to maintain consistency when objects are deleted i.e. you must remove the object from any collections in which it resides and remove any associations in which it participates.
7. Add a trigger to the contacts database which will generate a log file listing the names of all updated persons and the times when the updates were performed.
8. Write a macro that creates a person contact and associates an organisation object given the name of that organisation. The attributes of the person and the name of the organisation should be provided as input parameters.

Chapter 8

Setting User Preferences

One of the features of the OMS Pro system is that it allows the user to customise certain settings of the system through preferences. This includes not only presentation details such as fonts and date formats, but also some operational features such as the handling of temporary collections and the automatic propagation of certain collection updates according to the collection constraints. In this chapter, we give an overview of the various forms of preferences and their settings.

The preference options are available through the special “globe" menu at the top left of the OMS main window as shown in figure 8.1.

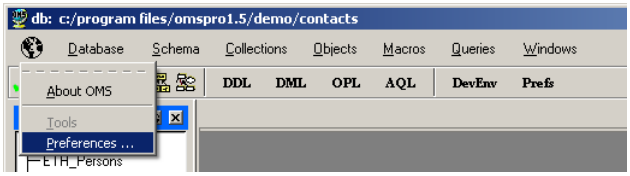


Figure 8.1: Accessing Preferences

The option **`About OMS`** provides information about the OMS Pro system version. Note that selecting this option and then enlarging the message panel will provide further details of the particular OMS Pro installation. If a database is open, it will also provide some information about that database.

The **`Tools`** option provides certain facilities for database administration. Since these tend to be for advanced users, we will not describe these here. However, note that one option is to force an unlock on a database. This is generally a dangerous option and should only be used to recover from situations where some kind of failure has resulted in an OMS process continuing to run in the background, thereby retaining a lock on the database. Under Unix, this can be detected in the OMS Pro database information displayed when you attempt to open a database and the operation fails due to the database being locked. If you increase the size of the display panel, you will see the machine and process id. By logging in on that machine, you will then be able to halt the process and re-open the database. Under Windows this must be done by using the **`Tools`** option and then selecting the **`Unlock Database`** option under **`Database`**. The system will ask for user and password identification. The user here will be the creator of the database, which by default is identified with name **`user`** and password **`null`**. Note that **`user`** is a special system object used in multi-user versions of OMS. It can also be used by any applications that require the notion of application users.

The **`Preferences`** option displays the Preferences window as shown in figure 8.2. On the left hand side is a column of buttons giving access to the various forms of preferences. We will discuss each of these in turn.

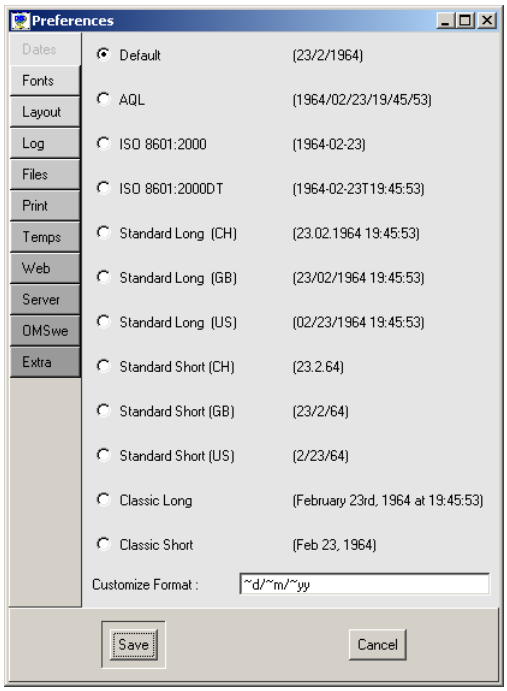


Figure 8.2: Date Preferences

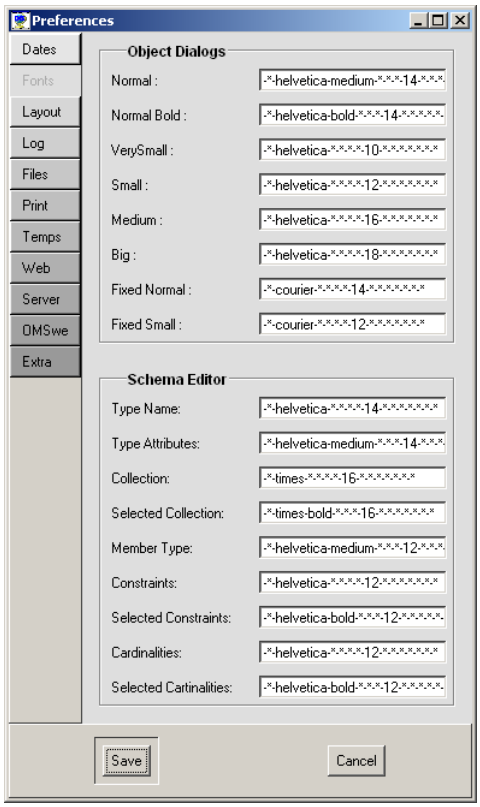


Figure 8.3: Fonts Preferences

By default, the **Dates** preference window, which is the first in the list, will be shown as seen in figure 8.2. **Dates** allows the user to specify their preferred format for displaying and inputting date values. This is only a presentation format and changing it has no effect on the internal representation. The user can therefore freely change the presentation format at any time without loss of information.

The **Fonts** option allows the user to change the fonts used by the OMS Pro system as shown in figure 8.3. There are two groups of font options. The first is to set the font preferences for the objects dialogs, the second is for the graphical schema editors discussed in chapter 5.

The **Layout** option shown in figure 8.4 allows the user to change the appearance of the main window, the general layout for OMS objects in terms of width and height of various fields and also display colours.

Desktop Width and **Desktop Height** are used to specify the dimension of the OMS main panel where all objects are displayed. Increasing these two values will provide more space to manage objects graphically, but losing track of the displayed objects will also become more probable.

Entry width specifies the display width of attribute value fields in terms of number of characters. **Text Height** and **Text Width** specify the display size of text areas in terms of number of lines and number of characters per line, respectively. **Multi Height** and **Multi Width** specify the display size of multi-valued attributes in terms of number of values displayed and the maximum character width for each value. Note that since elements in a collection are actually a multi-valued attribute of the collection considered as an object, the **Multi Height** and **Multi Width** also control the display of collections in terms of number of elements visible and their width.

Colour is used to distinguish values in OMS objects which are links in the sense that they can be clicked on to display other objects or values. For example, objects in a collection are displayed as

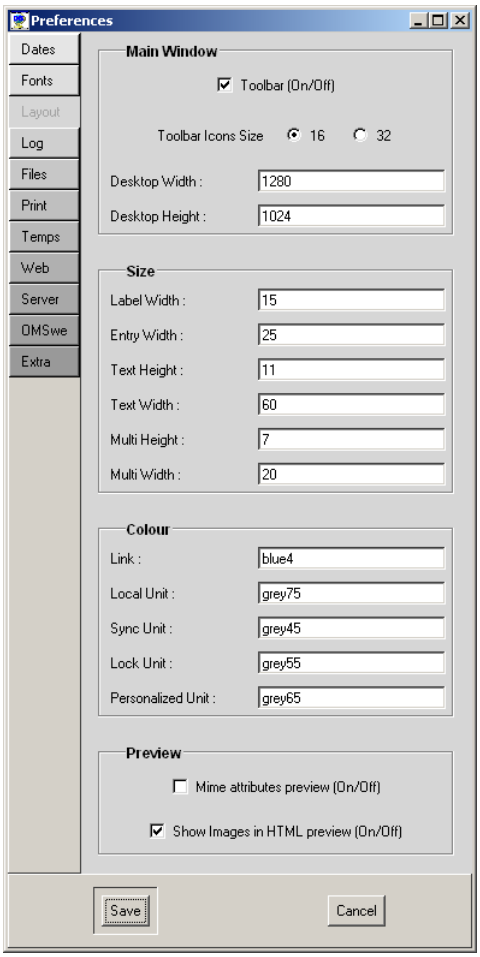


Figure 8.4: Layout Preferences

links. The **Colour** field **Link** specifies the colour of such link values. The remaining **Colour** options are for use with the distributed version of OMS Pro called OMS Connect. They allow the various categories of object parts in terms of whether they are local or remote to be coloured differently as a way of informing the user. Discussion of OMS Connect is outside the scope of this tutorial.

The **Preview** options govern how uri resource files and HTML documents are previewed directly within OMS. Enabling the first option will cause OMS to display every know type of resource file directly without the user having to click the **O** button of the uri attribute. The second option determines how images are handled within HTML source files. If it is enabled images are displayed in the preview.

The **Log** option of **Preferences** given in figure 8.5 is used to specify properties and actions for the log information displayed in the message panel. The log information can be colour coded to make it easier to distinguish error messages from confirmation messages. The **Positive Background** is the colour used for positive feedback in the event of action confirmations, whereas **Negative Background** is used for negative feedback in the even of action errors. **Default Background** is the colour used for the display of general information.

The size of the log window can be set by **Log Height**. The log text can be cleared at any time by clicking on the **Clear Button** and can be saved in a file with **Save As**. The **Scrolling** check box sets a scroll bar on the OMS panel to enable the log information to be read by scrolling rather than having to enlarge the window.

The **Files** option of **Preferences** shown in figure 8.6 allows default prefixes for file names to be specified and also command buttons to reload associated database files.

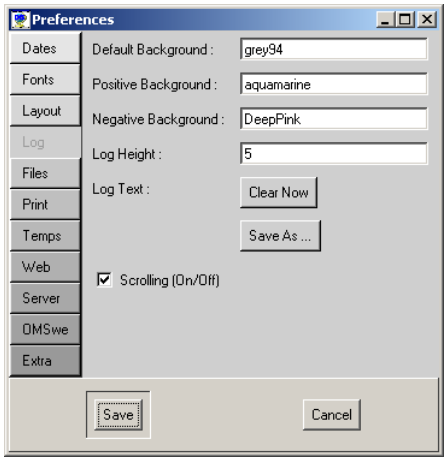


Figure 8.5: Log Preferences

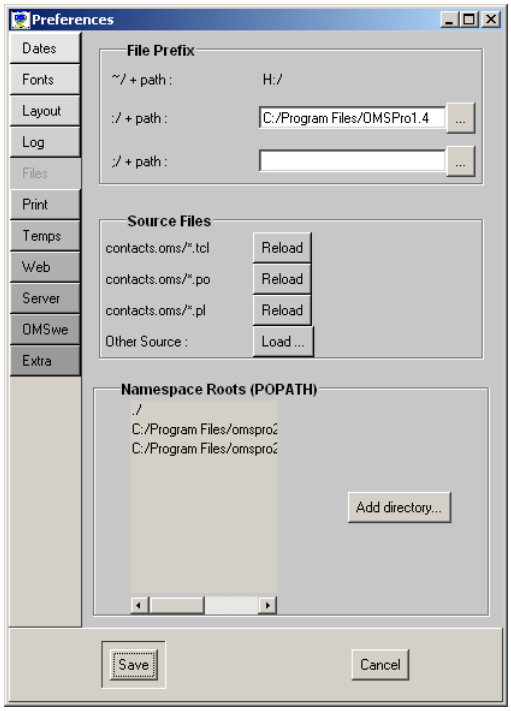


Figure 8.6: File Preferences

The **File Prefix** options allows three forms of file prefixes to be specified. These are useful in a database with many images or mime values to allow all the files to be stored elsewhere in the directory structure and then relative pathnames to be specified. If the value is prefixed with **~/**, for example, **~/myphoto.gif**, then the file name is relative to the directory set as OMSHOME. This directory is seen in the **File** preferences under **~/ + path**. If the user wishes to specify another directory that can be used for relative file names, they can do so in **File** preferences under the **:/ + path** or **;/ + path**. For these two prefixes, there is an editable text field in which the user can specify the required directory. Then all file names in the database prefixed with **:/** or **;/** will be assumed to be relative to those directories.

The **Source Files** options aid database development by allowing the user to reload external Prolog and Tcl files associated with the database. These files are generally located in the same directory as the database files and they are loaded automatically when the database is opened. If, during development, these files are edited, then the new versions can be reloaded by clicking on the appropriate buttons. The **Other Source** option enables source files located in other directories to be loaded.

Under **Namespace Roots** it is possible to specify a set of paths in which OMS will search for Prolog code. This option differs from the one above as the code loaded from these directories is not specific to any given database. It is meant for extensions of the OMS Pro system itself such as OIL.

The **Print** option of **Preferences** given in figure 8.7 specifies print commands and various format preferences.

There are two basic forms of printing in OMS – that of objects and graphics and that of text files. Text files are printed from **Print** options of schema definitions, method implementations and help text files. Printer commands for such text files can be specified under **Text Command**. Objects and graphics such as the various schema graphs are printed as Postscript files and the remainder of the options under Print refer to the printing of such files. Under Microsoft Windows it is highly advisable

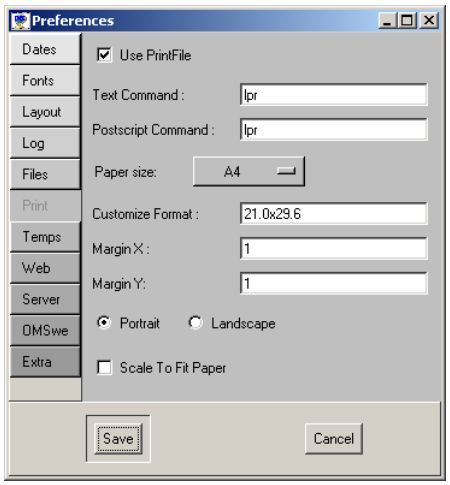


Figure 8.7: Print Preferences

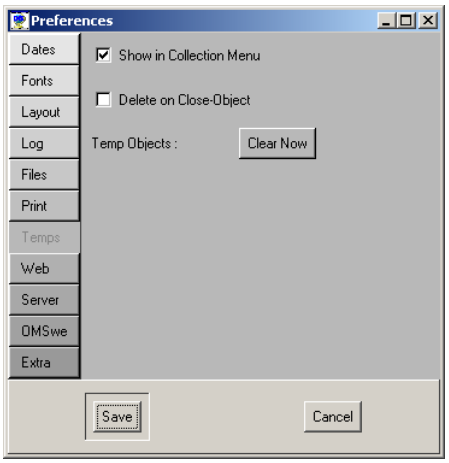


Figure 8.8: Temps Preferences

to check the **`Use PrintFile`** option as it facilitates printing Postscript documents to printers that use a different language such as PCL.

The **`Temps`** option of **`Preferences`** shown in figure 8.8 allows the user to have certain control over the handling of temporary collections that are generated as results of queries.

The top check box **`Show in Collection Menu`** is used to specify whether or not temporary collections should appear in any menus of collections. The **`Delete on Close-Object`** specifies whether a temporary collection should be deleted when the object is closed. The **`Clear Now`** button causes all temporary collections to be deleted.

Note that temporary collections are automatically deleted when a database is closed – either explicitly or by quitting an OMS session. Temporary collections may be made persistent, i.e. non-temporary, through their **`File`** menu.

The **`Web`** option of **`Preferences`** given in figure 8.9 allows the user to set various preferences as to which Web browser should be used to view urls from the OMS database and also on-line help information as provided in the **`Help`** menu of the OMS panel.

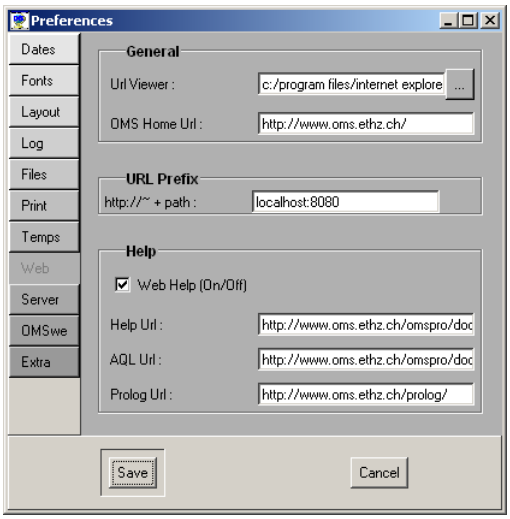


Figure 8.9: Web Preferences

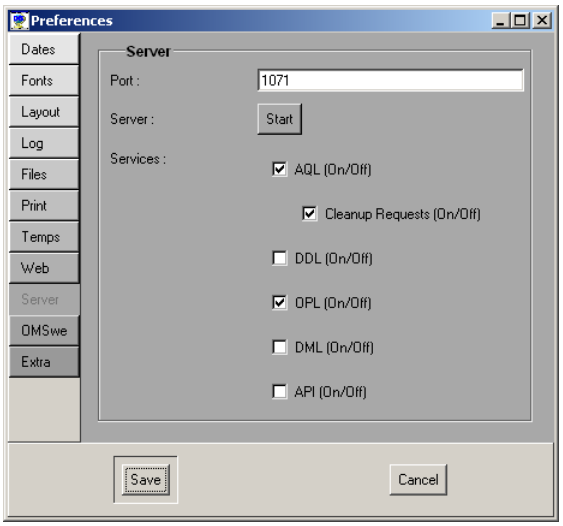


Figure 8.10: Server Preferences

Url Viewer specifies the browser application to be used. The **OMS Home Url** option specifies the address of web pages with general information about OMS.

The various **Help** options specify the addresses of web pages that can be accessed through the **Help** menu of the OMS Pro main panel under the option **Web Help**. For example, the **Help Url** may be set to the address of the OMS Pro On-Line Manual. The ability to change these settings means that help information such as on-line manuals for OMS and SICStus Prolog can be installed locally thereby enabling much faster access. The **Web Help** check box also allows the Web help option to be enabled or disabled.

The **Server** preferences shown in figure 8.10 are used to control access to OMS Pro databases from any form of client that establishes a socket connection.

The **Server** options allow a port number to be specified and a server to be started. Further, options can be set to enable/disable (On/Off) the form of requests that they server will accept – query requests (AQL), data definition requests (DDL), operation requests (OPL), data manipulation requests (DML) and calls to the OMS application programming interface (API). Details of the OMS Pro Server and its protocols are beyond the scope of this tutorial.

The **OMSwe** panel shown in figure 8.11 is used to configure the built-in content management framework. As the main focus of this tutorial is on the database part of OMS, rather than its web publishing capabilities, web elements and the options associated with them will not be discussed any further here.

The **Extra** panel of **Preferences** shown in figure 8.12 provides a number of other general options for the operation of the OMS system.

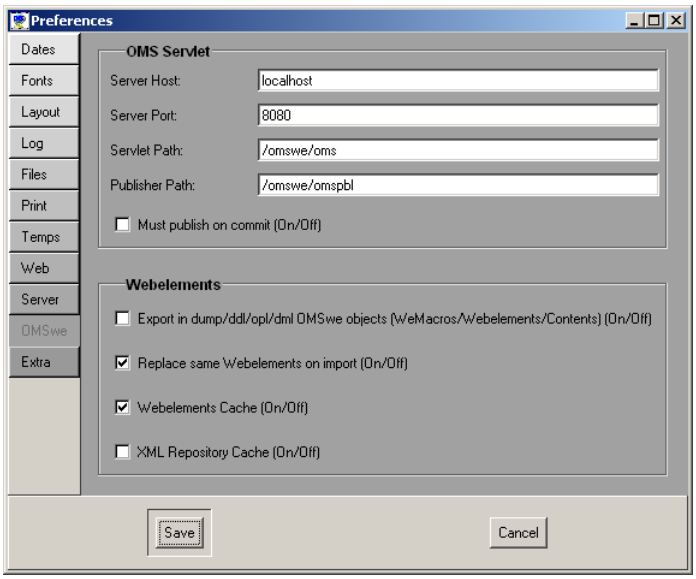


Figure 8.11: OMSwe Preferences

The options of **Match Results** allow the user to specify whether the results of an object matching operation should always appear as a window in the main panel or as a temporary collection. The default is that a single matching object will be displayed directly, but in the case that multiple objects match, they will be returned in a temporary collection.

The **Propagation** check box specifies whether operations to add or delete objects in collections should be propagated automatically in accordance with subcollection constraints. This means, for example, that if an object is added to a subcollection and it is not already a member of a supercollection, then it will be added automatically to that supercollection.

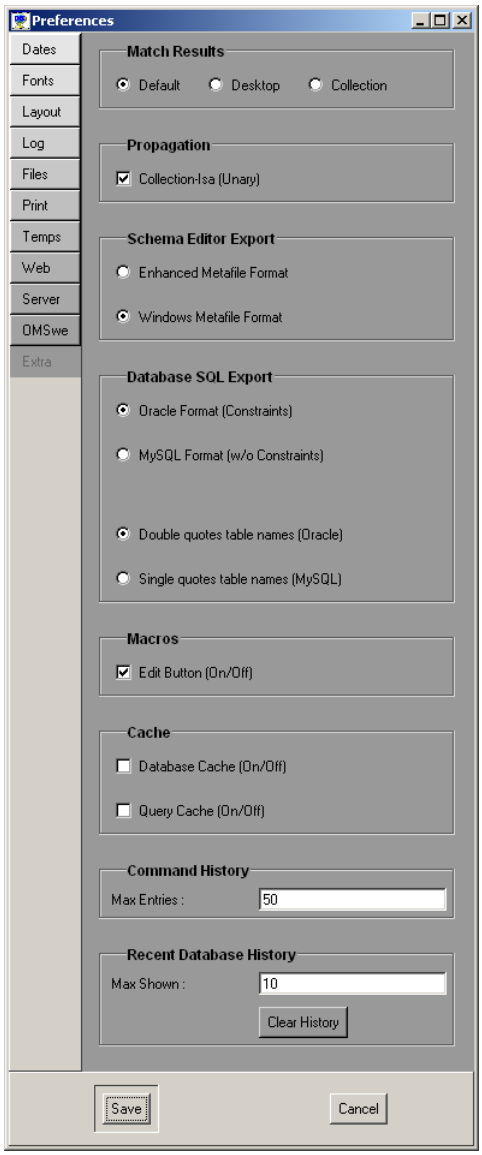


Figure 8.12: Extra Preferences

The **Macros** option specifies whether selection of a macro from the **Macros** menu of the main OMS Panel should execute the macro directly, or ask they user whether they want to run the macro or edit it.

The **Cache** options specify whether caches should be used during an OMS session. In the case of query caching, query results are cached for use in further query processing. Query caching speeds up processing, but does not take into account database updates since the time of the last query evaluation. This means that query caching is best used during query sessions when there are no updates between queries.

Command History and **Recent Database History** specify the maximum number of entries retained in the respective histories. Thus, **Recent Database History** determines the maximum number of entries that will appear in the **Open Recent** option of the **Database** menu of the main OMS Panel whereas **Command History** governs the number of commands that are stored in the language panel.

Appendix A

OMS Graphical Schema Notation

A.1 Typing

Types

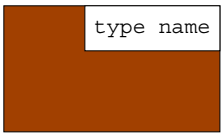


Figure A.1: Type Representation

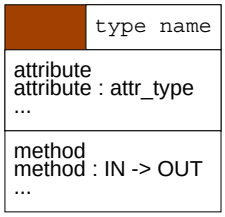


Figure A.2: Type with Properties

Subtypes

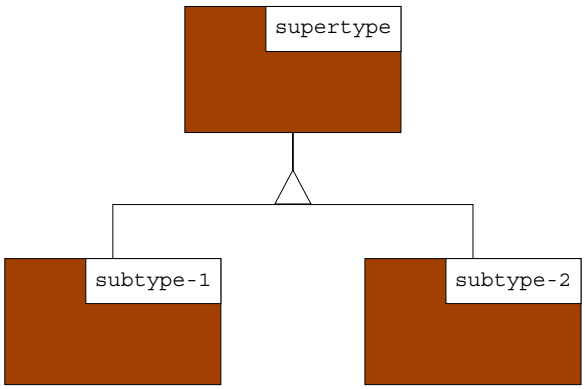


Figure A.3: Subtype Relationship (Inheritance)

A.2 Classification

Collections

Figure A.4: Collection Representation

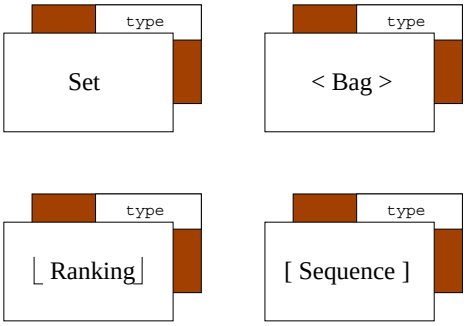


Figure A.5: Bulk Collections

Kinds and Roles

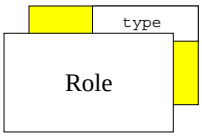


Figure A.6: Role Collection

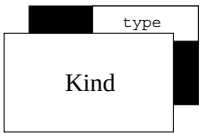


Figure A.7: Kind Collection

Subcollections

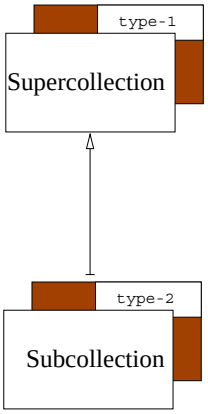


Figure A.8: Subcollection Relationship

Associations

Figure A.9: Association Representation

Constraints over Subcollection Relationships

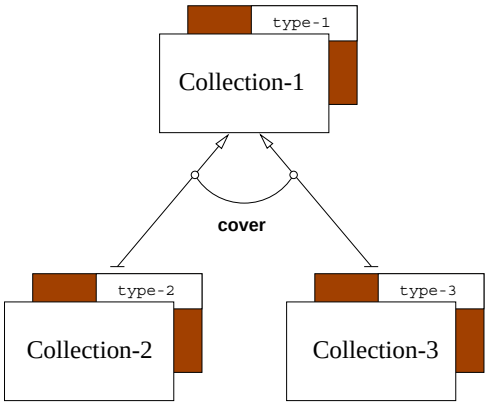


Figure A.10: Cover Constraint

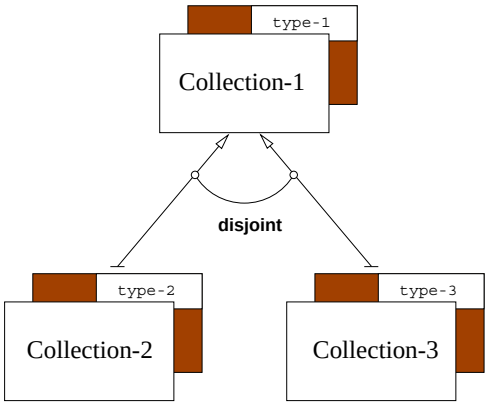


Figure A.11: Disjoint Constraint

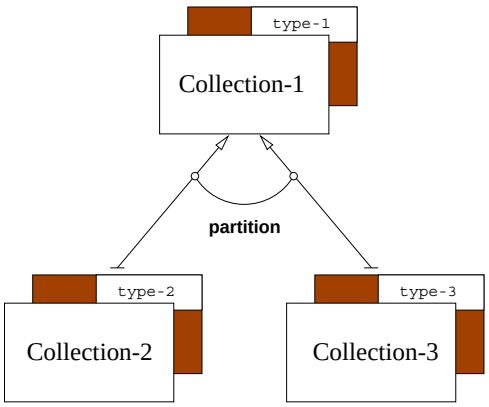


Figure A.12: Partition Constraint

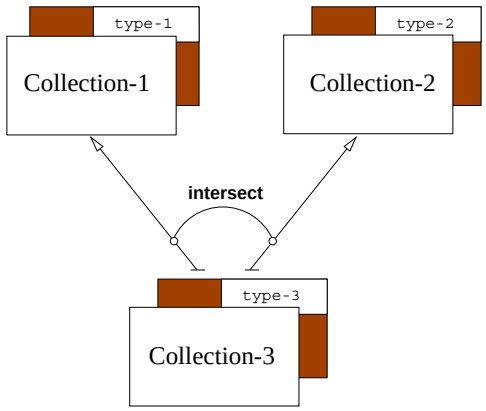


Figure A.13: Intersect Constraint

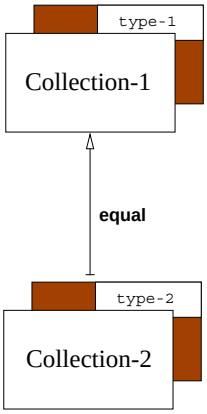


Figure A.14: Equal Constraint

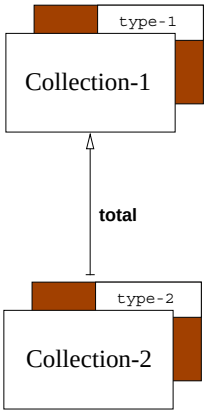


Figure A.15: Total Constraint

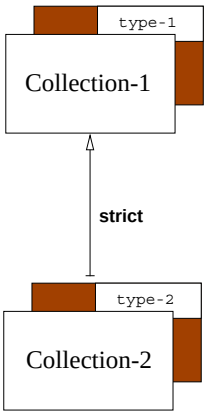


Figure A.16: Strict Constraint

Appendix B

Data Definition Language (DDL)

```
DDL             =  FileVer  ';'  {  DDLStmt  ';' }  .
DDLStmt         =  TypeDefn  |  CollDefn  |  ConstrDefn  |  MacroDecl  .

FileVer         =  'ddl'  '('  Version  ','  Options  ','  Reserved  ')'  .
Version         =  SQString  .       // Of the form: VersionTag-Major-Minor-Beta
Options         =  SQString  .       // Must be a prolog list of (Opt,Val) pairs
Reserved        =  SQString  .

COMMENT         =  '//'  <comment_until_eol>  .

IdentName       =  Ident  |  String  .
Ident           =  Alpha  { AlphaNum }  .      // string without whitespaces
String          =  SQString  |  DQString  .
SQString        =  "'"  StringVal  "'"  .
DQString        =  '"'  StringVal  '"'  .
StringVal       =  { AlphaNum  |  Whitespace  }  .

AlphaNum        =  ALPHA  |  NUMBER  .
ALPHA           =  'a'-'Z'  |  '_'  .
NUMBER          =  '0'-'9'  .
Whitespace      =  SPACE | TAB .
```

B.1 Types

```
TypeDefn        =  BTypeDefn  |  StructDefn  |  PTypeDefn  .

BTypeDefn       =  'btype'  TypeName  'subtype of'  TypeName
                   '('  { BProperty  ';' }  ')'  .
BProperty       =  ( 'default'  |  'null'  |  'constraint' )  ':'  PropVal  .
PropVal         =  'null'  |  String  .

StructDefn      =  'struct'  TypeName  '('  { FieldDecl  ';' }  ')'  .
FieldDecl       =  FieldName  ':'  Type  .

PTypeDefn       =  'type'  TypeName  [ 'subtype of'  Supertype ]
```

```
                '('  { Property  ';' }  ')'  .
Supertype       =  TypeName  |  '('  TypeName  { 'and'  TypeName }  ')'  .

Property        =  AttrDecl  |  MethodDecl  | TriggerDecl  .
AttrDecl        =  AttrName  ':'  [ Bulk  'of' ]  Type  .
Bulk            =  'set'  |  'bag'  |  'ranking'  |  'sequence'  .

Type            =  BaseType  |  TypeName  .
BaseType        =  'boolean'  |  'bulk'  |  'card'  |  'date'
                |  'integer'  |  'proc'  |  'real'  |  'string'
                |  'text'  |  'type'  |  'uri'  |  'value'  .

MethodDecl      =  MethodName  ':'  '('  [ InputDecl ]  ')'  '->'
                   '('  [ OutputDecl ]  ')'  .
TriggerDecl     =  TriggerName  ':'  'trigger'  'on'  Event  .
Event           =  'create'  |  'update'  |  'delete'  |  'commit'  |  'rollback'  .

InputDecl       =  ParamsDecl  .
OutputDecl      =  ParamsDecl  .
ParamsDecl      =  AttrDecl  { ','  AttrDecl }  .

TypeName        =  IdentName  .
AttrName        =  IdentName  .
FieldName       =  IdentName  .
MethodName      =  IdentName  .
TriggerName     =  IdentName  .
```

B.2 Collections and Constraints

```
CollDefn        =  'collection'  CollName  { ','  CollName }
                   ':'  Bulk  'of'  Type  .

ConstrDefn      =  SubCollDefn  |  AssocDefn  |  AConstrDefn  |  VConstrDefn
                |  IConstrDefn  |  KindDefn  .

SubCollDefn     =  'constraint'  SubCollName  { ','  SubCollName }
                   'subcollection of'  SuperCollName  .
AssocDefn       =  'constraint'  BinCollName  'association from'  SourceCollName
                   CardConstr  'to'  TargetCollName  CardConstr  .
CardConstr      =  '('  Integer  ':'  InfNumber  ')'  .
InfNumber       =  Integer  |  '*'  .

AConstrDefn     =  'constraint'  '('  SubCollName  { 'and'  SubCollName}  ')'
                   AConstraint  SuperCollName  .
AConstraint     =  'disjoint'  |  'cover'  |  'partition'  .

VConstrDefn     =  'constraint'  SubCollName  VConstraint
                   '('  SuperCollName  { 'and'  SuperCollName }  ')'  .
VConstraint     =  'intersect'  .

IConstrDefn     =  'constraint'  SubCollName  IConstraint  SuperCollName  .
IConstraint     =  'equal'  |  'total'  |  'strict'  .
```

```
KindDefn        =  'constraint'  CollName  'is'  'kind'  .

BinCollName     =  IdentName  .
SubCollName     =  IdentName  .
SuperCollName   =  IdentName  .
SourceCollName  =  IdentName  .
TargetCollName  =  IdentName  .
```

B.3 Macros

```
MacroDecl       =  'macro'  MacroName  ':'  '('  [ InputDecl ]  ')'  .
MacroName       =  IdentName  .
```

B.4 OMS Schema Evolution

```
DDL             =  ( TypeChange  |  CollChange  |  MacroChange )  ';'  .

TypeChange      =  DeleteType  |  RenameType  |  AddProp  |  DeleteProp
                |  ChangeProp  |  SwapAttr  .

DeleteType      =  'delete'  'type'  TypeName  .
RenameType      =  'rename'  'type'  OldTypeName  'to'  NewTypeName  .
AddProp         =  'type'  TypeName  'add'  '('  { Property  ';' }  ')'
                |  'type'  TypeName  'add'  Property  .
DeleteProp      =  'type'  TypeName  'delete'  AttrName
                |  'type'  TypeName  'delete'  MethodName
                |  'type'  TypeName  'delete'  TriggerName  .
ChangeProp      =  'type'  TypeName  'change'  AttrName  'to'  AttrDecl
                |  'type'  TypeName  'change'  MethodName  'to'  MethodDecl
                |  'type'  TypeName  'change'  TriggerName  'to'  TriggerDecl  .
SwapAttr        =  'type'  TypeName  'swap'  AttrName1  'and'  AttrName2  .

CollChange      =  DeleteColl  |  RenameColl  |  ChangeColl  .
DeleteColl      =  'delete'  'collection'  CollName  .
RenameColl      =  'rename'  'collection'  OldCollName  'to'  NewCollName  .
ChangeColl      =  'change'  'collection'  OldCollName  'to'  [ NewCollName  ':' ]
                   Bulk  'of'  Type  .

MacroChange     =  DeleteMacro  |  RenameMacro  .
DeleteMacro     =  'delete'  'macro'  MacroName  .
RenameMacro     =  'rename'  'macro'  OldMacroName  'to'  NewMacroName  .
```

Appendix C

Operation Specification Language (OPL)

```
OPL             =  {  ( PrologSpec  |  LangSpec  )  '.'  } .
```

C.1 Prolog Implementations

```
PrologSpec      =  PrologMethod  |  TriggerSpec  |  MacroSpec  |  MacroCall  .

MethodSpec      =  TypeName  '::'  MethodName
                   '('  Inputs  ','  Outputs  ')'  ':-'  Body '.'
                |  TypeName  '::'  MethodName
                   '('  Inputs  ','  Outputs  ')'  '.'  .

TriggerSpec     =  TypeName  '::'  TriggerName  ':-'  Body  '.'
                |  TypeName  '::'  TriggerName  '.'  .

MacroSpec       =  '::'  MacroName  '('  Inputs  ')'  ':-'  Body  '.'
                |  '::'  MacroName  '('  Inputs  ')'  '.'  .
MacroCall       =  MacroName  "("  Params  ")"  "."  .

Inputs          =  '['  Params  ']'  .
Outputs         =  '['  Params  ']'  .
Params          =  [ Param  { ','  Param } ]  .
Param           =  PrologAtomic  |  PrologList  |  PrologVariable  .
Body            =  PrologTerm  .
```

C.2 Other Languages (OIL)

```
LangSpec        =  LangID  '::'  (  LangMethod  |  LangTrigger  |  LangMacro  )  .

LangMethod      =  'meth'  '('  TypeName  "'"  Code  "'"  ')' .
```

```
LangTrigger     =  'trig'  '('  TypeName  "'"  Code  "'"  ')' .
                                  // must return boolean

LangMacro       =  'macro'  '('  "'"  Code  "'"  ')' .

LangId          =  Ident  .       // unique language identifier (oil)
Code            =  LANG_CODE  .   // source code accepted by Language;
                                  // "'" must be escaped
```

Appendix D

OMS Query Language (AQL)

```
Query           =  Basevalue  |  TypeCast  |  Evaluation  |  Projection
                |  AttrAccess |  MethodSel |  StructSel  |  ElemGet
                |  IfThenElse  |  Reduce  |  Expression |  PQuery
                |  CollQuery  |  BinQuery |  TypeQuery  |  BoolQuery |  NumQuery  .

CollQuery       =  Query  |  CollQuery0  .     // return value is collection or list
BinQuery        =  Query  |  BinQuery0  .      // return value is binary collection
TypeQuery       =  Query  |  TypeQuery0  .     // return value is type object
BoolQuery       =  Query  |  BoolQuery0  .     // return value is boolean value
NumQuery        =  Query  |  NumQuery0 .       // return value is integer/real value

CollQuery0      =  Collection  |  CollOpColl   // return value is collection or list
                |  List  |  Extent  |  BCollOpColl.
BinQuery0       =  BCollOpBColl .              // return value is binary collection
TypeQuery0      =  Type  .                     // return value is type object
BoolQuery0      =  Boolean  |  BoolExp  .      // return value is boolean value
NumQuery0       =  CollCount  |  Integer  .    // return value is integer/real value
PQuery          =  '('  Query  ')'  .

Assignment      =  Ident ':=' QUERY  .
Comment         =  '//' <comment_until_eol>  .

QueryList       =  { Query { '|' QueryList } }  .
```

D.1 Basevalues

```
Basevalue        =  Integer  |  Real  |  String  |  Date  |  Boolean  |  Object  .

Ident            =  Alpha  { AlphaNum }  .      // string without whitespaces
String           =  SQString  |  DQString  .
SQString         =  "'"  StringVal  "'"  .
DQString         =  '"'  StringVal  '"'  .
StringVal        =  { AlphaNum  |  Whitespace  }  .

Object           =  Objectvalue  |  'o'  '"'  Objectvalue  '"'  .
```

```
Objectvalue      =  'o'  Integer  .

Integer          =  Integerval  |  'i'  '"'  Integerval  '"'  .
Integerval       =  Number  { Number }  .

Real             =  Real  |  'r'  '"'  Realvalue  '"'  .
Realvalue        =  Number  { Number }  [ '.' Number { Number } ]  .

Date             =  "'"  Datevalue  "'"  |  'd'  '"'  Datavalue  '"'  .
Datevalue        =  <Year>  '/'  <Month>  '/'  <Day>
                    '/'  <Hour>  '/' <Min>  '/'  <Sec>
                 |  'now'  .

Boolean          =  BOOL_VAL  |  'l'  '"'  BOOL_VAL  '"'  .
BOOL_VAL         =  'true'  |  'false'  .

List             =  BULK  '['  QueryList  ']'  [ 'of'  Type ]  .
BULK             =  'set'  |  'bag'  |  'ranking'  |  'sequence' .

Collection       =  CollName  |  [ 'collection'  |  'coll'  |  'c' ]  String .
CollName         =  Ident  .                    // if not a reserved word

Type             =  TypeName  |  [ 'type'  |  't' ]  String  .
TypeName         =  Ident  .                    // if not a reserved word

AlphaNum         =  ALPHA  |  NUMBER  .
ALPHA            =  'a'-'Z'  |  '_'  .
NUMBER           =  '0'-'9'  .
Whitespace       =  SPACE | TAB .
```

D.2 Properties and Type Operations

```
AttrAccess       =  Query  '.'  Ident .

MethodSel        =  Query  '.'  Ident  '('  BTypeList  ')'  .
BTypeList        =  { Query  { ','  Query }  }.

StructSel        =  Query  '@'  [ Ident  |  Integer  ]  .

Projection       =  ('project'  |  'proj' )  AttrNames  'of'  Query  .
AttrNames        =  Ident  |  '('  [ Ident  { ','  Ident } ]  ')'  .

Extent           =  ( 'extent'  |  'ext' )  TypeQuery   .

TypeCast         =  Query  ':'  Type
                 |  Type  '('  Query  ')'  .
```

D.3 Expressions

```
Evaluation       =  EVAL  String  |  EVAL  Var  .   // Var must return a string/text
```

```
EVAL            =  'evaluate'  |  'eval'  .

IfThenElse      =  'if'  '(' BoolQuery ')'  'then'  PQuery  'else'  PQuery  .

Expression      =  NumExp  |  BoolExp  |  Product  .

NumExp          =  UNARY_OP  NumQuery  |  NumQuery  BIN_OP  NumQuery  .
UNARY_OP        =  '+'  |  '-'  |  'abs'  |  'integer'  |  'float'  |  'floor'
                |  'sin'  |  'cos'  |  'atan'  |  'sqrt'  |  'log'  |  'exp'  .
BIN_OP          =  '+'  |  '-'  |  '*'  |  '/'  |  'mod'  |  '^'  .

BoolExp         =  'not'  BoolQuery  |  Query  BOOL_OP  Query  |  ElemCheck  .
BOOL_OP         =  'and'  |  'or'  |  'xor'
                |  '='  |  '<'  |  '>'  |  '=<'  |  '>='  |  '<>'  |  'like'  .

Product         =  Query  PROD  Query  .  // both for single or coll operants
PROD            =  ( 'product'  |  'x' )  .
```

D.4 Operations on Collections

```
ElemGet         =  'the'  ( Integer  |  Var )  'in'  CollQuery
                |  SpecIndex  CollQuery  .

SpecIndex       =  'first'  |  'last'  |  'min'  |  'max'  |  'any'  .

ElemCheck       =  Query  'in'  CollQuery  .

CollExpr        =  CollOpColl  |  CollCount  .
CollOpColl      =  CollQuery  CBIN_OP  CollQuery
                |  CollFlat  |  SelQuery  |  MapQuery  .

CBIN_OP         =  'union'  |  'intersection'  |  'intersect'  |  'minus'  .

CollCount       =  'count' CollQuery  .

CollFlat        =  ( 'flatten'  |  'flat' )  CollQuery  .

SelQuery        =  'all'  Var  'in'  CollQuery  'having'  '('  BoolQuery ')'  .

MapQuery        =  'map'  Var  'in'  CollQuery  'by'  PQuery  .

Reduce          =  ( 'reduce'  |  'red' )  Ident  'in'  CollQuery
                   ( 'aggregate'  |  'aggr' )  Ident  'by'  PQuery(Var1,Var2)
                   'default'  Query  .
```

D.5 Operations on Binary Collections

```
BCollOpColl     =  BCOLL_OP  BinQuery                    // return Collection
                |  BinQuery  'div'  CollQuery  .
BCOLL_OP        =  'domain'  |  'dom'  |  'range' | 'ran'  .
```

```
BCollOpBColl    =  BCOLL_BOP1  BinQuery                // return BinCollection
                |  BinQuery  BCOLL_BOP  CollQuery
                |  BinQuery  BCOLL_BOP2  BinQuery  .

BCOLL_BOP       = 'domain_restriction'  |  'domrest'  |  'dr'  // Bin Op Coll
                |  'range_restriction'  |  'ranrest'  |  'rr'
                |  'domain_subtraction'  |  'domsub'  |  'ds'
                |  'range_subtraction'  |  'ransub'  |  'dr'  .

BCOLL_BOP2      =  'composition'  |  'compose'  |  'compose' . // Bin Op Coll

BCOLL_BOP1      =  'nest'  |  'inv'  |  'unnest'  .           // Op Bin
```

Appendix E

Data Manipulation Language (DML)

```
DML             =  (  FileVer  |  DMLStmt  )  ';' .
DMLStmt         =  CreateStmt  |  DressStmt  |  StripStmt  |
                |  DeleteStmt InsertStmt  |  RemoveStmt  .

FileVer         =  'dml'  '('  Version  ','  Options  ','  Reserved  ')'  .
Version         =  SQString  .       // Of the form: VersionTag-Major-Minor-Beta
Options         =  SQString  .       // Must be a prolog list of (Opt,Val) pairs
Reserved        =  SQString  .

COMMENT         =  "//"  <comment_until_eol>  .

Ident           =  ALPHA  { AlphaNum }  .      // string without whitespaces
String          =  SQString  |  DQString  .
SQString        =  "'"  StringVal  "'"  .
DQString        =  '"'  StringVal  '"'  .
StringVal       =  { AlphaNum  |  Whitespace  }  .
IdentName       =  Ident  |  String  .

AlphaNum        =  ALPHA  |  NUMBER  .
ALPHA           =  'a'-'Z'  |  '_'  .
NUMBER          =  '0'-'9'  .
Whitespace      =  SPACE | TAB .
```

E.1 Object Manipulation

```
CreateStmt      =  'create'  [ 'object' ]  ObjectAlias   .

DressStmt       =  'dress'  [ 'object' ]  ObjectAlias  'as'  TypeName
                   [ InstanceParam ]  [ 'values'  '('  { AttrAssign }  ')' ]  .

AttrAssign      =  AttrName  '='  AttrVal  .

AttrVal         =  UniValue  |  MultiValue  .

StripStmt       =  'strip'  [ 'object' ]  ObjectAlias
```

```
                  [ 'to'  |  'off' ]  TypeName  .

DeleteStmt      =  'delete'  [ 'object' ]  ObjectAlias  .

UniValue        =  UniUnary  |  UniBinary  .
UniUnary        =  BaseValue  |  [ 'object' ]  ObjectAlias  .
UniBinary       =  '('  UniValue  ','  UniValue  ')'  .
MultiValue      =  '['  [ UniValue  { ','  UniValue } ]  ']'  .

ObjectAlias     =  IdentName  .
BaseValue       =  Integer  |  Boolean  |  ...  |  Ident  |  String  .

TypeName        =  IdentName  .
AttrName        =  IdentName  .
```

E.2 Collection Manipulation

```
InsertStmt      =  'insert'  'into'  [ 'collection'  |  'association' ]
                   CollName  ':'  MultiValue  .

RemoveStmt      =  'remove'  'from'  [ 'collection'  |  'association' ]
                   CollName  ':'  MultiValue  .

CollName        =  IdentName  .
```

Bibliography

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modelling Language User Guide*. Addison-Wesley, 1998.

[Cat96] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996. release 1.2.

[EN98] A. Erni and M.C. Norrie. Approaches to Accessing Databases through Web Browsers. *INFORMATIK, Journal of the Swiss Informaticians Society*, October 1998.

[ENK98] A. Erni, M. C. Norrie, and A. Kobler. Generic Agent Framework for Internet Information Systems. In *Proc. IFIP WG 8.1 Conference on Information Systems in the WWW Environment*, Beijing, China, July 1998.

[JGJ+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*, 4(2):167–192, 1995.

[KN99a] A. Kobler and M. C. Norrie. OMS Java: Lessons Learned from Building a Multi-Tier Object Management Framework. In *Proc. Workshop on Java and Databases: Persistence Options, OOPSLA'99*, Denver, Colorado, USA, October 1999.

[KN99b] A. Kobler and M.C. Norrie. OMS Java: Introductory Tutorial. Technical report, Institute for Information Systems, ETH Zurich, CH-8092 Zurich, Switzerland, June 1999.

[KN00] A. Kobler and M. C. Norrie. OMS Java: A Persistent Object Management Framework. *L'0bjet*, 6(3/2000), November 2000.

[KNSG01] A. Kobler, M. C. Norrie, B. Signer, and M. Grossniklaus. OMS Java: Providing Information, Storage and Access Abstractions in an Object-Oriented Framework. In *Proc. 7th International Conference on Object-Oriented Information Systems*, Clagary, Canada, August 2001.

[KNW98] A. Kobler, M. C. Norrie, and A. Würgler. OMS Approach to Database Development through Rapid Prototyping. In *Proc. 8th Workshop on Information Technologies and Systems (WITS'98)*, Helsinki, Finland, December 1998.

[MT94] M. Missikoff and M. Toiati. MOSAICO – A System for Conceptual Modelling and Rapid Prototyping of Object-Oriented Database Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 508. ACM, 1994.

[Nor95] M. C. Norrie. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, volume VI, chapter 25. IOS, 1995. (originally appeared in Proc. European-Japanese Seminar on Information and Knowledge Modelling, Stockholm, Sweden, June 1994).

[Nor00] M. C. Norrie. Client-Server Database Architectures for the Web. In Y. Masunga and S. Spaccapietra, editors, *Advances in Databases and Multimedia for the New Century - A Swiss/Japanese Perspective*. World Scientific Publishing, 2000.

[NP02] M. C. Norrie and A. Palinginis. A Modelling Approach to the Realisation of Modular Information Spaces. In *Proc. 14th Conference on Advanced Information Systems Engineering*, Toronto, Canada, May 2002.

[NPW98] M. C. Norrie, A. Palinginis, and A. Würgler. OMS Connect: Supporting Multidatabase and Mobile Working through Database Connectivity. In *Proc. Conference on Cooperative Information Systems*, New York, USA, August 1998.

[NS02] M. C. Norrie and B. Signer. Web-Based Integration of Printed and Digital Information. In *Proc. 2nd Workshop on Data Integration over the Web*, Toronto, Canada, May 2002.

[NW97] M. C. Norrie and A. Würgler. OM Framework for Object-Oriented Data Management. *INFORMATIK, Journal of the Swiss Informaticians Society*, (3), June 1997.

[NW98] M. C. Norrie and A. Würgler. OMS Rapid Prototyping System for the Development of Object-Oriented Database Application Systems. In *Proc. Intl. Conf. on Information Systems Analysis and Synthesis*, Orlando, USA, August 1998.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison Wesley, 1992.

[SEN00] B. Signer, A. Erni, and M. C. Norrie. A Personal Assistant for Web Database Caching. In *Proc. 12th Conference on Advanced Information Systems Engineering (CAiSE'00)*, Stockholm, Sweden, June 2000.

[SGN01] B. Signer, M. Grossniklaus, and M. C. Norrie. Java Framework for Database-Centric Web Site Engineering. In *Proc. 4th Workshop on Web Engineering*, Hong Kong, May 2001.

[SKN98] A. Steiner, A. Kobler, and M. C. Norrie. OMS/Java: Model Extensibility of OODBMS for Advanced Application Domains. In *Proc. 10th Conf. on Advanced Information Systems Engineering (CAiSE'98)*, Pisa, Italy, June 1998.

[SN97] J. Supcik and M. C. Norrie. An Object-Oriented Database Programming Environment for Oberon. In *Proc. of the Joint Modular Languages Conference (JMLC'97)*, Linz, Austria, March 1997.

[SNGH02] B. Signer, M. C. Norrie, P. Geissbuehler, and D. Heiniger. Aural Interfaces to Databases based on VoiceXML. In *Proc. 6th IFIP Working Conference on Visual Database Systems*, Brisbane, Australia, May 2002.

[Sup99] J. Supcik. *Odeon: An Object-Oriented Data Model and its Integration in the Oberon System*. Phd thesis, Department of Computer Science, ETH, CH-8092 Zurich, Switzerland, 1999.

[Wue00] A. Wuergler. *OMS Development Framework: Rapid Prototyping for Object-Oriented Databases*. PhD Thesis, Department of Computer Science, ETH, CH-8092 Zurich, Switzerland, 2000.