TODL Manual

TOOBIS Deliverable T35.TR.1

University of Athens

01 Pliroforiki

O_2 Technology

TODL Manual

1. Introduction

TODL is a language used to define objects and types within a Temporal Objected Oriented Data Base. TODL is based on ODL v.1.2 of ODMG and extends this language to handle temporal characteristics. Note that TODL -as ODL- is language independent and it is not a manipulation language. Although the user may define signatures of methods for the interfaces (classes) defined, TODl provides no means for the implementation of these methods. For this purpose he has to use a manipulation language like C++ or Java or Smaltalk[1]

In the following paragraphs you will find out how to use TODL.

2. Defining interfaces

In TODL we use the term *interface* to be synonymous to the term *class*. As in C++ -for instance- we define classes, in TODL we define interfaces, which are actually the public parts of a class, that is the interface through which we may manipulate an instance of a class.

Each interface may have any of the following parts:

- a list of super-classes, other interfaces that is from which the class inherits

1. Note that in the current version of TODL only C++ can be used as a manipulation language, along with the TODM.

- an extent, an entry point in the database containing all the instances of the interface, which are now persistent.
- zero or more instance properties, attributes and/or relationships.
- zero or more instance properties, from the ones defined above, that will be used as keys for the interface
- zero or more operations (methods) that may be applied on the instances of the interface.

Some of those parts may be extended to have temporal characteristics[1]. Note that within an interface type and constant definitions may appear, having scope inside the interface.

The simplest interface that can be defined is one that doesn't inherit from other interfaces, does not have an extent or keys defined, does not have instance properties or operations. This is a null interface and its definition may be:

```
interface nothing
{};
```

This definition is different from the following:

```
interface something;
```

as the first one is a full definition -even if it defines nothing- while the second is a *forward definition* for an interface which will be fully defined in subsequent statements in the TODL file. The difference is the two curly brackets that enclose nothing.

2.1 Inheritance

This interface (nothing) may be extended to inherit from other interfaces, e.g.

```
interface nothing: an_interface, another_interface
{};
```

In the example given above the interface nothing inherits from two other interfaces (multiple inheritance) namely *an_interface* and *another_interface*. Any number of names of previously defined interfaces may be used at the inheritance specification list. The rules for simple and multiple inheritance are given in the Appendix ---.

1. the only parts that cannot have temporal information is the inheritance specification and the extent definition.

2.2 Extent and persistence

Another extension of the interface definition is the use of an *extent* for the interface. This is done through the following definition:

```
interface with_extent
(extent AllInstances)
{};
```

The name of the extent should be unique in the database. Since TODL is used over O_2, the extent names follow the same uniqueness rules as O_2 names.

Another option in interface declaration is that we may define -apart from the extent definition- that all the instances of the interface must be persistent or transient. This is done by using the keyword `persistent` or `transient` in one of the following places:

- if an *extent* or *keys* definition exist immediately after the parentheses containing them
- if an *inheritance list* exist, but not *extent* or *key* definiton, after the *inheritance list*, delimited by colon.
- if none of the above exists after the name of the interface, delimited by colon

If neither `persistent` nor `transient` keywords are present then the instances of the interface may be or not persistent, depending on the application that creates the instance.

2.3 Instance properties

The most important item of interfaces is the instance properties that differentiate the interfaces. By the term *instance property* we mean an attribute or a relationship. Attributes are used to define values, while relationships are used when we want to define links between different interfaces that can be used for integrity checks as well. Instance properties are defined between the two curly brackets that define the interface body.

2.3.1 Attributes

An attribute may be defined as:

```
attribute string productName;
```

In the example presented above we can identify the three mandatory parts of an attribute definition, that is:

- the keyword `attribute`

- the type of the attribute, -that may be one of the simple types defined for TODL, a collection over these types, a previously defined type- either within or outside the interface- or a collection over a previously defined type.
- the name of the attribute. The name of the attribute should be unique within the scope of the interface. This means that no other attribute, relationship, operation, constant, exception, type or extent may have the same name as an attribute, if they belong in the same interface.

In the attribute definition clause two optional parts are available:

- the keyword `readonly` which may prepend the keyword `attribute` and defines that the attribute defined can be set only once -at the constructor of the interface
- the temporal definition which consists of two parts: the valid time definition and the transaction time definition. Note that this temporal definition may appear in different places when defining an interface, but it always has the same form. For more information see Paragraph 2.5 Temporal Definitions.

2.3.2 Relationships

A relationship is defined as:

```
relationship Person father;
```

Let' s assume that this relationship has been defined in the interface Person and declares that a “link” exists between a person -instance of the Person interface - and the person’s father- another instance of the Person interface.

Similarly to attribute declaration, a relationship definition must have the following mandatory parts:

- the keyword `relationship`
- the target path of the relationship which may be either a class (interface) name or a collection defined over an interface. Note that the interface used as target path for the relationship must have been defined before the relationship definition either as a common interface definition or as a forward definition
- the relationship' s name which should be unique in the scope of the interface. Analogously to, attributes this uniqueness means that no other attribute, relationship, operation, type, extent, eception or constant defined within the interface may have the same name.

In the relationship definition optional parts are also available:

- the keyword `state` may prepend the relationship keyword to define that the relationship points to different variants of a temporal object[1], instead of pointing to an object as a whole. A constraint for using the state keyword for relationships is that the target path must have temporal characteristics as a whole. Failure to satisfy this constraint will result to an error in the processing of the TODL file.
- a temporal definition of the relationship which follows the name of the relationship. The temporal definition is identical to the one for attributes and it will be presented in subsequent paragraphs. Note that a state relationship may not have a temporal definition.
- a definition of the inverse relationship, that is the inverse link which points from the target interface back to the current interface. If the relationship is defined to be state, then no inverse relationship may be declared. Of course, the inverse relationship should be also defined in the target interface. An example of the definition of an inverse relationship is given below:

```
relationship Person father
     inverse Person::children;
```

  That is we define the relationship “father” between a person (an instance of the class Person) and another person. This relationship has an inverse relationship pointing from the father to his children. Of course, in general, relationships are defined between different interfaces.

Relationships have a special characteristic called *cardinality*, which defines the expected number of instances that can be found at each end of the link defined by the relationship. The cardinalities available are the following

1-1	one instance of each interface is available at each end of the relationship link
1-N	one instance at the start of the link is connected to many instances at the end of the link
N-1	many instances from the start of the link will be connected to one instance at the end link
N-M	many instances from one end of the link will be linked to many instances at the other end of the link

The inverse of an 1-1 relationship is an 1-1 relationship, the inverse of a 1-N relationship is a N-1 relationship, the inverse of a N-1 relationship is a 1-N relationship and finally the inverse of an N-M relationship is an N-M relationship.

1. See the TODL and TODM specifications for more information on temporal object variants.

Examples: A 1-1 relationship is the relationship “spouse” between married persons: every person (at least in a non-polyamory society) has exactly one spouse. A 1-N relationship is the relationship “state” from a Country interface: each country has one or more states, while a N-1 relationship is the inverse of the “state” relationship, as each state belongs to exactly one country, while many states belong to the same country. Finally, an N-M relationship is the one among companies and the shareholders of them. Each company has a number - usually more than one - of shareholders, and each shareholder may have shares for more than one companies.

2.4 Operations

Apart from instance properties an interface definition may also contain operations that apply to the particular interface. An operation may have a return value which can be one of the valid types defined in TODL, or it doesn't return a value in which case the keyword `void` is used in the place of the return type. An operation has a name, which follows the same uniqueness constraints as the names of attributes and relationships. After the operation name a list of parameters may be given, enclosed in parentheses.

Each parameter has an indication of its type -which should be a valid TODL type- as well as an indication of what we may call its “direction”. The direction of a parameter is one of `in`, `out` or `inout`.The meaning of each value is given in the following table

Value	Comment
in	The parameter is used for input only and it cannot be changed during the execution of the operation
out	The parameter is used for output only so it shouldn't have an input value
inout	The parameter can be used either for input or for output

Each parameter may also have a temporal definition, which will be described in the following paragraphs. A temporal definition may also appear at the end of the parameter list outside the parentheses, meaning that the return value of the operation has temporal characteristics. Of course, the keyword void should not be followed by a temporal specification.

Two more keywords may appear at the signature of an operation before the return type. The first one is the keyword `state` which is used if the operation is to apply to a variant of the temporal object instead of the whole object. Of course, in order to use the state keyword, the interface in which the operation is defined should have temporal characteristics as a whole.

The other keyword is the keyword `oneway` which defines that the operation does not return any information. Only the `void` type is acceptable as return type for `oneway` operations and the direction of all parameters to these operations must be `in`.

2.5 Temporal Definitions

In many points we have already demonstrated that a temporal definition can be used. A temporal definition - if present - can have at least one of two parts: the valid time definition and the transaction time definition. Either of them can appear in any order. This kind of definition may appear in the interface definition, the instance property definition, the operation return type definition and the operation parameter type definition. In the case of temporal keys the definition is different.

2.5.1 Temporal characteristics - an introduction

We define as valid time of a fact stored in the database the time point or time period during which the fact was valid in the real world. An istance property with valid time characteristics[1] is one which stores the full history of the property's evolution in the real world. Each item in the history is called a *variant* and stores, besides the value, the time point or time period that represents the valid time of the fact the property represents.

The transaction time of a fact is the time period during which the fact was known tothe database, i.e. the time period during which the fact was stored in the database. Each such period begins with the time point at which the fact was inserted in the database and its end is either the time point when the fact was logically deleted (i.e. removed or updated) from the database or the special value *Until Changed*, that denotes that the fact is still stored in the database. We say that an instance property has transaction time characteristics[2] when it stores the complete history of the property's evolution within the database. Each item in the history is called a *variant* and has associated with it atiem period with the transaction time of the fact the property represents.

1. Or historical instance property.
2. Or that it is a rollback instance property.

We say that an instance property has bitemporal characteristics[1] when it has both valid and transaction time characteristics. Finally, we say that an instance property has temporal characteristics when it has valid time or transaction time or bitemporal characteristics.

In some case we want an object to have temporal characteristics as whole, meaning that all its instance properties evolve over the valid or transaction time axis in the same way. This creates the temporal objects. A variant of a temporal object is one item in the history of the object which holds a snapshot version of the temporal object along with the valid and/or transaction time timestamp for this version.

2.5.2 Valid time definition

Valid time has been declared to be the time, is the real world, during which the fact modelled in the database was true. That is if we keep payrolls and if Employee Jones had a salary of 10k from May 1997 up to September 1997 then the valid time of the fact that Jones had a 10k salary is [1997-05, 1997-10)[2]

In order to define that an interface (instance property, operating return type or parameter type) has valid time[3] characteristics, we use the `valid` clause of the temporal definition e.g.

```
attribute float salary valid
```

This is the simplest form of the valid clause and implies the following:

- the evolution of the attribute is kept over valid time
- the valid time timestamps will be represented using periods (default setting)
- the valid time timestamps may not overlap each other
- the valid time timestamps are expressed in the default calendar (Gregorian), at *second* granularity (default granularity).

The complete syntax specification for the `valid` clause is

```
valid [state | event] [overlaps]
    [granularity gran] [calendar cal]
```

1. Is a bitemporal instance propery.
2. How a time literal is written, presented and the types will be described later.
3. Or that an interface is an historical interface

Keywords `state` and `event` can be used to specify whether *periods* or *instants* will be used for valid time timestamps. If keyword `state` is present, valid timestamps will be represented using periods. Keyword `event` designates that valid timestamps will be represented using instants. If neither keyword is present, the default is `state`.

Period-type timestamps are used when the timestamped information acquires a value at some instant and retains this value until some future instant. Such an example is an employee's salary: the employee begins receiving a certain salary at some instant, and continues receiving the same salary until (s)he receives a raise (or (s)he is fired!). Using period-type timestamps, this piece of temporal information is represented as:

```
value: salary, VT: [starting_instant, ending_instant)
```

Instant-type timestamps are used when the timestamped information describes events that happen instantaneously, i.e.do not have a duration. An example of such information are promotions of employees: if an employee is promoted to some rank, this promotion is instantaneous and may be represented using the following instant-timestamped format:

```
value: head_of_department, VT: promotion_instant
```

Keyword `overlaps` may only be used when the keyword `state` is used (or both `state` and `event` keywords are missing), i.e. when information is timestamped using periods. This keyword designates that two variants may have overlapping (but not identical) timestamps. If the `overlaps` keyword is missing, no pair of variants may have overlapping timestamps. Overlapping timestamps are used when the timestamped information represents aggregate values. For example, a student's average grade may be equal to 8 during the first semester of school year 1996, while the average grade of the same student, during the whole school year, may be equal to 8.5. This piece of information may be represented using overlapping timestamps as follows:

```
value: 8, VT: [September 1996, February 1997)
value: 8.5, VT: [September 1996, June 1997)
```

Note taht this situation could not occur with an employee's salary, since the first variant would indicate that the employee's salary is equal to 8, for each month from September 1996 to February1997, while the second variant would indicate that the employee's salary is equal to 8.5 for each month from September 1996 to June 1997. Thus, for months September 1996 to February 1997, the employee would have two different salaries.

Finally, the `calendar` and `granularity` subclauses may be used to specify the desired calendar and granularity, respectively, for valid timestamps. If the `calendar` subclause is omitted, it defaults to *Gregorian*, whereas if the `granularity` sub-clause is not present, the default granularity for the specified (or default) calendar is used[1]. When using the *Gregorian* calendar, the allowed granularities are *year*, *month*, *day*, *hour*, *minute* and *second*. The allowed granularities for used-defined calendars, depend on the calendar definition (see Paragraph 6. Calendar definition)

2.5.3 Transaction time definition

Transaction time is the time during which some piece of information was known to the database. The transaction time associated with each piece of information begins at the instant when this piece was inserted in the database and ends at the instant that it was (logically) removed. If some piece of information has not yet been removed from the database, the end of its transaction time has a special value *until changed*, which always evaluates to the current instant. Transaction time is assigned and managed by the database. It is expressed at the default calendar and granularity. An example of transaction time is that if we have the class Employee and the fact that employee Jones is receiving 10k as salary was recorded in the database on June 15th 1997 at noon and was deleted on July 17th 1998 at 10:00 a.m., the transaction time of this fact is *[1997-06-15 12:00:00, 1998-07-17 10:00:01)*.

In order to define that an interface (instance property, operation return type or parameter type) has transaction time[2] characteristics we use the `transaction` clause. The transaction clause is very simple and consists only of the `transaction` keyword, e.g.:

```
attribute float salary transaction
```

This means that we keep the evolution of the salary of employees over transaction time.

2.5.4 Bitemporal definition

If both valid and transaction time is kept for an interface (instance property, operation return type or parameter type) then we say that we have a bitemporal interface (instance property, operation return type or parameter type). Note that valid and transaction time are orthogo-

1. Default granularity for the *Gregorian* calendar is the *second* granularity, while for user-defined calendars the default granularity is the one mapped to *chronons* - see Paragraph 6. Calendar definition

2. also called rollback

nal to each other and for this reason the two clauses may appear in any order.

2.6 Keys

Apart from the extent, each interface has one more type property: the keys. The format of the keys-specification clause may be one of the following:

```
key key_specification
keys key_specification {, key_specification }
```

(braces mean that the enclosed syntactical construct is repeated one or more times). Each key specification must designate an instance property name of the corresponding interface, or a set of instance properties of the corresponding interface, grouped using parenthesis. For instance properties without valid time semantics, the key declaration specifies that no two instances of the corresponding interface may hold the same value for the specified instance properties (or combination of values if the key comprises of multiple instance properties).

For instance properties with valid time characteristics, the following extensions are available:

- if no keyword is prepended, or the keyword `point` is used, then no two instances of the interface may have the same value for the same time point for the specified instance property
- if the keyword `period` is used then no two instances of the interfaces may have the same value for exactly the same timestamp (period or instant) for the specified instance property
- if the keyword `object` is used then no two instances of the interface may have the same value for the specified instance property for any point in the time axis. That is if one instance of the interface has had the value X at some point in time, no other instance may have the same value -at the specified instance property- at any time in the past, present or future.

The key specification appears immediately after the extent definition -if such exists- within the parentheses, and it is introduced by one of the keywords `key` or `keys`, depending on the number of instance properties present.

Using the keyword `key` when more than one instance properties are defined as keys, or the keyword `keys` when exactly one instance property is defined as key, does not raise a syntax error, but results to a warning message. Instance properties defined as keys form a comma separated list and some of them may also appear in parentheses.

Example. In the folwoing example we present an interface with multiple keys. The interface definition in TODL is the following:

```
interface subsidiary: company
(keys owner, point name)
{
      relationship company owner;
      attribute String name valid granularity day;
}
```

In this case we define an interface to model subsidiaries of companies. Each subsidiary belongs to exactly one company (this is modelled through the **owner** relationship) and all the subsidiaries of the same company may not have the same name at a specific time point.

This case should be more restrictive, by declaring that if a subsidiary of a company holds a specific name for a time period, then no other subsidiary of the same company may have this name again, declaring thus an object temporal key. This definition is given below:

```
interface subsidiary: company
(keys owner, object name)
{
      relationship company owner;
      attribute String name valid granularity day;
}
```

3. Type definitions

In TODL constructs are provided for naming data types; that is, C-language-like declarations are allowed, that associate an identifier with a type. TODL uses the `typedef` keyword to associate a name with a data type; a name is also associated with a data type via the `struct`, `union` and `enum` declarations.

3.1 Basic types

TODL basic types include integers, floats, characters, boolean, octet, user-defined time and `any` types. These are described in the following paragraphs:

3.1.1 Integer types

TODL supports **`long`** and **`short`**, **`signed`** and **`unsigned`** integer data types. **`long`** represents the range [-2^{31}, 2^{31}), while **`unsigned long`** the range [0, 2^{32}). **`short`** represents the range [-2^{15}, 2^{15}), while **`unsigned short`** the range [0, 2^{16}).

Implementation note: As O_2 provides only the `integer` data type, this type is actually used in all cases. The O_2 `integer` data type is the same as the C `int` data type.

3.1.2 Float types

TODL floating-point types are **`float`** and **`double`**. The **`float`** type represents the IEEE single-precision floating point numbers while the **`double`** represents IEEE double-precision floating-point numbers.

Implementation note: As O_2 provides only the `float` data type, this type is actually used in both cases, and which is of double-precision.

3.1.3 Character type

TODL uses the **`char`** data type that consists of 8-bit quantities.

3.1.4 Boolean type

The **`boolean`** data type is used to denote a data item that can take only one of the values `TRUE` or `FALSE`.

Implementation note: Although O_2 supplies for a `boolean` type, C++ used to create the definitions of the classes before being imported in O_2, so the `boolean` type is mapped to the O_2 `integer` data type.

3.1.5 User-Defined Time Types

User defined time (UDT) types are allowed in TODL and have the following flavours: `interval`, `instant` and `period`. In all cases the user may define the calendar and granularity at which the time quantity can be expressed. In the following paragraphs the way to represent intervals, instants and periods that are expressed in the default (Gregorian) calendar is demonstrated. For more information the user is referred to the TODM manual.

Interval type: Intervals are defined to be time amounts like 5 years, or 2 days. An interval does not have a starting or ending point and is known only by its duration. An interval can be expressed in the following way: the keyword **`interval`** is followed by a string containing the integer showing the interval's duration. Then a definition of the granularity and the calendar at which the interval is expressed may be found. Note that contrary to other UDT types, in intervals the string representing the interval does not change depending on the type of the calendar and is always an integer. Examples of interval types are showed in the following table:

TABLE 1. Examples of intervals

Interval literal	Explanation
interval “3” granularity month	An interval representing 3 months in the *Gregorian* (default calendar)
interval “1” granularity semester calendar academic	An interval representing one semester in the Academic calendar (user defined calendar)
interval “5”	An interval of 5 seconds (default) in the default calendar (*Gregorian*)
interval “76” granularity day calendar Gregorian	An interval of 76 days in the *Gregorian* calendar

Instant types: An instant is a specific point on the time axis. An instant is represented by the keyword **`instant`** followed by a string specifying the location of the instant on the time axis, and an optional specification of the granularity and the calendar at which the instant is expressed. Granularity specification is introduced by the **`granularity`** keyword, followed by the identifier representing the granularity, and the calendar specification is introduced by the **`calendar`** keyword followed by the identifier representing the calendar. Depending on the calendar used, the string specifying the instant value may vary. For user-defined calendars, the string format depends on the calendar definition.When using the default (Gregorian) calendar the format of the string is the following:

YYYY-MM-DD HH:MM:SS

Note also that any of the fields - except for the year field - may not appear, but if a field is missing, following fields may not appear. That is if the month field is missing, the day, hour, minute and second fields may not appear. Depending on whether or not a granularity specification is given, extra fields may not be taken into account, or missing fields may be filled in using defaults.

Special instant values, are the “beginning”, “forever”, “now” and “now()”. They can be given either directly or prepended by the **`instant`** keyword. They do not need a granularity specification - they always take the default - but they may be used with user defined calendars in which case the **`calendar`** keyword, followed by the calendar name can be used. The **`beginning`** instant represents the earliest time point (the start of the time axis on the specified calendar), while the **`forever`** instant represents the latest time point (the end of the time axis) at the specified calendar. The **`now`** instant is the present time point, while the **`now()`** instant represents the late binding now, i.e. an instant which is always equal to the current value of the system clock.

Period types: A period is a specified time duration on the time axis, with well specified starting and ending points. A period literal is introduced by the **`period`** keyword, followed by a string specifying the value of the period and an optional granularity and/or calendar specification. Granularity and calendar specifications follow the same rules as instant granularity and calendar specifications.

The format of the string specifying the valua of a period is *[starting_instant, ending_instant)*, i.e. the string always starts with an opening bracket (`[`), followed by the instant representing the starting instant of the period. This instant is actually the string part of any instant literal in the specified granularity. After a comma the instant specifying the ending instant of the period is given and the period string finishes with a closing parenthesis (`)`). The use of an opening bracket and a closing parenthesis means that the starting instant of a period is part of the period, while the ending one is not.

3.1.6 Any and octet types

The **octet** type is a remnant of OMG IDL and is used to model an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system. The **any** type is used to permit specification of values that can express any TODL type.

Implementation note. The **`octet`** data type is transformed to the **`char`** data type and the **`any`** data type is transformed to the **`d_Any`** type provided by the O_2 C++ binding.

3.2 Template types: Strings

In TODL, a string is a sequence of characters. A string is defined by using the keyword **string** which may be followed by a number representing the maximum string's length. The number is enclosed in <>. If a positive integer maximum size is given, the string is termed a bounded string; if no maximum size is specified the string, is termed an unbounded string.

3.3 Complex declarators: Arrays

Using TODL, one may define multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension. After the identifier representing the array, a list of positive integers - or expressions yielding positive integers - may appear, each one enclosed in brackets representing each dimension of the array.

Example: The following definition defines an attribute consisting of an array of 15 booleans

```
attribute boolean boolean_array[15];
```

The following definition defines an attribute consisting of a three-dimensional array of shorts with dimensions 15, 17, and 9 respectively:

```
attribute short short_3d_array[15][17][9]
```

3.4 Typedef definitions

Using the `typedef` keyword one can associate one or more names to a type name. The syntax of the typedef clause is:

```
typedef <type> <declarators>
```

where `<type>` is one of the simple or complex standard types of TODL, a structure, enumeration or union definition, or a previously defined type or collection on such a type. `<declarators>` is a list of identifiers, which will be associated with the type defined at this point. The user may also define that one of these identifiers is an array, by specifying the number of elements enclosed in brackets.

Example. The declaration

```
typedef set<short> vectint
```

associates the identifier `vectint` to the type `set<short>`.

3.5 Union definition

TODL unions are a cross between C unions and switch statements. TODL unions must be discriminated; that is the union header must specify a type tag field that determines which union member to use for the current instance of the call. The identifier following the `union` keyword defines a new legal type. Union types may also be defined using a `typedef` declaration. The union statement syntax is the following:

```
union <identifier> switch (<switch_type_spec>)
{<switch_body>}
```

where the `<switch_type_spec>` may be one of integer (`short`, `long`, `unsigned short` or `unsigned long`) , `char`, `boolean` or `enum` or a scoped name that has previously been defined as one of these types. The <switch_body> is a list of case statements which consists of a case label and an element specifier. The case label may be of the form

```
case <const_expr>
```

or simply the **`default`** keyword, followed by a colon. The <const_expr> must be consistent with the <switch_type_spec>. The **`default`** case may appear at most once. The element specifier is a couple of the form:

```
<type_spec> <identifier>
```

where <type_spec> is any of the valid simple and basic types defined in TODL and collections on those types and the identifier is the name associated with this type of the union. There may be more than one case labels corresponding to each element specifier. A colon is found after each element specifier.

Case labels must match or be automatically castable to the defined type of the discrimination. The complete set of matching rules as shown in the next table:

TABLE 2. Case Label Matching

Type	Matched By
`long`	any integer value in the value range of `long`
`short`	any integer value in the value range of `short`
`unsigned long`	any integer value in the value range of `unsigned long`
`unsigned short`	any integer value in the value range of `unsigned short`
`char`	`char`
`boolean`	`TRUE` or `FALSE`
`enum`	any enumerator for the discriminator enum type

Name scoping rules require that the element declarators in a particular union be unique. If the <switch_type_spec> is an enumeration, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the <switch_body>. The value of the union is the value of the discriminator together with one of the following:

- if the discriminator value was explicitly listed in the **case** statement, the value of the element associated with the **case** statement;
- if a default case label was specified, the value of the element associated with the default case label;
- no additional value.

Implementation note: although unions can be defined and will be present in the C++ header file created, they can not be imported in O_2 using the O_2 import tool. So the user will also receive a warning regarding the union type definition (s)he just made.

3.6 Enumeration definition

Enumerated types consist of ordered lists of identifiers having the following syntax:

```
enum <identifier> {<enumerator> {, <enumerator>}}
```

where the `<enumerator>` is an identifier. The enumerators are represented by a comma-separated list enclosed in curly brackets. The <identifier> immediately after the **enum** keyword defines a new valid type. Enumeration definitions may be done also using the **typedef** keyword.

Examples. The following definition defines an enumeration for the representation of the male and female sex:

```
enum sex_type { male, female };
```

Implementation note: enumerations are passed as integers to O_2. They are present in the C++ file, and are imported to the database, but they appear with their integer equivalent in queries. This means that the application should be aware of the mapping from the enumeration value to the integer value.

3.7 Structure definition

Structures can be defined either using a typedef statement or by directly assigning a name to this structure. In either case this can be done using the following syntax:

```
struct <identifier> {<member_list>}
```

The `<member_list>` contains a list of members - fields - of the structure. Each member is a couple of the form:

```
<type_specificator> <identifiers>
```

where <type_specificator> is any of the valid simple and basic typed defined within TODL. No recursive type specification is allowed in TODL. The <identifiers> part is a list of identifiers associated with the members (fields) of the structure.

Name scoping rules require that the fields of each structure are unique, within the structure. The value of a structure is the value of all of its members.

Examples: The following definition defines a structure used to keep the address information of employees:

```
struct address_type
{
      string street;
      string number;
      string zip_code;
      string city;
      string state;
      string country;
}
```

4. Constant expression definition

In a TODL file the user may define constants. In order to do so the `const` statement is used. After the keyword `const` the type of the expression is defined. The types allowed are the following

TABLE 3. Allowed constant types

short	char	period
long	string	boolean
double	interval	unsigned short
float	instant	unsigned long

After the type definition the name for the expression is given followed by the equal sign and then the expression itself. Depending on the expression type operators may or may not be allowed. In any case, operators allowed for any type, other than the signed and unsigned short and long and the float. Also in an expression having operators the user cannot mix integers (signed and unsigned short and long) with floats.

Operators that can be used (and the types on which they may apply are the following:

unary +	(unsigned) short, (unsigned) long, float, interval, double
unary minus	short, long, float, double, interval
binary plus, binary minus, multiply, divide	(unsigned) short, (unsigned) long, float, double
modulo, shift left, shift right, bitwise or, bitwise and, 2's complement	(unsigned) short, (unsigned) long

In the expression, apart from literals the user may also use scoped names that have been previously defined in the TODL file. If a scoped name is used in a constant expression, then it must have been previously defined as integer, character, boolean, floating point, string, instant, period or interval type.

In all integer expressions the results are computed as long and when the final result is provided then the casting to the appropriate type -the one that the expression is defined to have- is made. The same hold for float expression were all calculations are made with results of type double. If an expression results a type different than the one specified in its definition, then an error is raised.

When in a TODL file the name of a constant expression is found then it is replaced with the result of the expression -if it can be computed- or the expression itself. Constants can be used only in TODL declarations and not in the TOQL queries or other applications querying the database created.

5. Exception declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicated that an exceptional condition has occurred during the execution of a request. The syntax used for exception declarations is the following:

```
exception <identifier> {<member_list>}
```

Each exception is characterised by its identifier, an exception type identifier and the type of the associated return value (as specified by the members present in the possibly empty <member_list> in its declaration). If an exception is return as the outcome of an operation, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

A set of standard exceptions is defined corresponding to standard run-time errors which may occur during the execution of a request. These standard exceptions are presented in the following paragraph, as they are defined in OMG IDL. Note that user defined exceptions should be handled by the application and information for them should be given by the programmer of the appropriate operations.

Examples. The following example creates an exception which defines that a student tries to resign from a course he had never taken:

```
exception not_in_course
{
      Student this_student;
      Course this_course;
}
```

If this exception is raised then its members `this_student` and `this_course` will take as values the student and the course that caused the error.

5.1 Standard exceptions[1]

The standard exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in **raises** expression, when defining operations. In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. Each standard exception includes a minor code to designate the subcategory of the exception.

Each standard exception also includes a **completion_status** code which takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES: The object implementation has completed processing prior to the exception being raised.

1. From the CORBA v.2.0 OMG IDL Specifications

COMPLETED_NO: The object implementation was never initiated prior to the exception being raised.

COMPLETED_MAYBE: The status of implementation completion is indeterminate.

5.1.1 Standard Exceptions Definitions

The standard exceptions are defined in the following table:

TABLE 4. Standard Exceptions

Exception	Exception definition
UNKNOWN	The exception is unknown
BAD_PARAM	An invalid parameter was passed to the operation
NO_MEMORY	Dynamic memory allocation failed during operation
IMP_LIMIT	Violated implementation limit
COMM_FAILURE	A communication failure happened
INV_OBJREF	An invalid object reference took place
NO_PERMISSION	No permission exists for the attempted operation
INTERNAL	An internal ORB error happened
MARSHAL	An error during marshalling of parameters or of the result occurred
INITIALIZE	A failure during ORB initialization happened
NO_IMPLEMENT	The implementation of the operation is not available
BAD_TYPECODE	A bad type code was given
BAD_OPERATION	An invalid operation was invoked
NO_RESOURCES	Insufficient resources for the request
NO_RESPONSE	The response to the request is not yet available
PERSIST_STORE	A failure of the persistent storage occurred
BAD_INV_ORDER	Routine invocations are out of order
TRANSIENT	A transient failure occurred - the request should be re-issued
FREE_MEM	Cannot free allocated memory

TABLE 4. Standard Exceptions

Exception	Exception definition
INV_IDENT	The identifier syntax is invalid
INV_FLAG	An invalid flag was specified
INTF_REPOS	An error accessing the interface repository occurred
BAD_CONTEXT	An error during the processing of the context object occurred
OBJ_ADAPTER	A failure was detected by the object adapter
DATA_CONVERSION	A data conversion error occurred
OBJECT_NOT_EXIST	An invocation on a deleted object took place

6. Calendar definition

Users may define their own calendars if they wish to. This is done by using the **`calendar`** statement. The syntax for it is given below:

```
calendar <identifier>
{
      origin <instant_literal>;
      granules <granule_list>
}
```

where `<granule_list>` is a list of comma-separated `<granules>` defined as follows:

```
<identifier> chronon
```

or

```
<identifier> <identifier>: <integer_literal>
```

This means that if the **`chronon`** keyword is used, then the granule defined is mapped directly to the chronons of the calendar, i.e. is the finest granularity available. This is always the default granularity for the defined calendar.

If the second form of granule definition is used then this has the following meaning: the first identifier is the name of the granule to be defined, while the second identifier is the name of the immediately finer granule, followed by a constant mapping from the granule being defined to its finer.

Note that if the user wants to define his own calendars, then he must also define a number of functions, outside TODL. The exact functions to be defined are described in the TODM manual.

7. Scope Rules

To be able to use the same names in different modules, scope rules are defined, in accordance to CORBA [ODM], for TODL. An entire TODL file forms a naming scope. In addition, the following kinds of definitions form nested scopes:

- module
- interface
- structure
- union
- operation
- exception
- calendar

Identifiers for the following kinds of definitions are scoped:

- types
- constants
- enumeration values
- exceptions
- interfaces
- attributes
- operations
- granules within calendars
- calendars[1]

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scoped (except from the calendars). An identifier declaring a module is considered to be defined by its first occurrence in a scope. Subsequent occurrences of a module declaration within the same scope reopen the module allowing additional definitions to be added to it. Identifiers are case insensitive, although when referring to an identifier the same case as the defining occurrence should be used[2].

1. A calendar should be unique in the database - not just in one of the nested scopes

Type names defined in a scope are available for immediate use within that scope. A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching further out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined - i.e. if one has used a name defined in an enclosing scope in the current scope, one cannot then redefine a version of the name in the current scope. Such a redefinition yields a compilation error.

A qualified name (one of the form <scoped-name>::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with "::", the resolution process starts with the file scope and locates subsequent identifiers in the qualified name by the rule described above.

Each TODL definition in a file has a global name within that file. The global name for a definition is constructed as follows: Prior to starting to scan a file containing a TODL specification, the name of the current root is initially empty and the name of the current scope is initially empty. Whenever a **`module`** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current root; upon detection of the termination of the module the trailing "::" and identifier are deleted from the name of the current scope.Whenever an **`interface`**, **`struct`**, **`union`**, or **`exception`** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **`interface`**, **`struct`**, **`union`**, or **`exception`**, the trailing "::" and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of a TODL definition is the concatenation of the current root, the current scope, a "::" and the <identifier>, which is the local name for that definition.

2. This is a OMG IDL remnant too. For more information check the CORBA specifications

8. How to use TODL

TODL accepts as input a file, containing all the TODL specifications. Note that those specifications should give autonomous definitions of classes. This means that when referring to classes whose definitions do not form part of the file being processed, then these classes should already be imported into the database. The same holds for types, other than the simple ones.

If the definition of a class that already exists in the database is supplied, then this class is redefined, according to the O_2 rules for class redefinition.

9. BNF of TODL

The following is the BNF of the TODL language. Reserved words are marked with bold letters.

<specification>	::= <definition> [<specification>]	1
<definition>	::= <type_dcl> **;** | <const_dcl> **;** | <except_dcl> **;** | <interface> **;** | <module> **;** | <calendar_definition> **;**	2
<module>	::= **module** <identifier> **{**<specification>**}**	3
<interface>	::= <interface_dcl> | <forward_dcl>	4
<interface_dcl>	::= <interface_header> [**:** <persistence_dcl>] [<temporal_dcl>] **{**[<interface_body>]**}**	5
<persistence_dcl>	::= **persistent** | **transient**	6
<temporal_dcl>	::= <valid_dcl> | <transaction_dcl> | <valid_dcl> <transaction_dcl> | <transaction_dcl> <valid_dcl>	7
<transaction_dcl>	::= **transaction**	8
<valid_dcl>	::= **valid** [<state_event_dcl>] [**overlaps**] [**granularity** <granularity>] [**calendar** <calendar>]	9
<state_event_dcl>	::= **state** | **event**	10
<granularity>	::= **year** | **month** | **day** | **hour** | **minute** | **second** | <identifier>	11
<calendar>	::= **gregorian** | <identifier>	12
<forward_dcl>	::= **interface** <identifier>	13
<interface_header>	::= **interface** <identifier> [**:** <inheritance_spec>] [<type_property_list>]	14
<type_property_list>	::= ([<extent_spec>] [<key_spec>])	15
<extent_spec>	::= **extent** <string>	16
<key_spec>	::= **key**[**s**] <key_list>	17
<key_list>	::= <key> [**,** <key_list>]	18
<key>	::= [temporal_key_dcl] <property_name> | (<property_list>)	19

```
<temporal_key_dcl>    ::= point | period | object                          20
<property_list>       ::= [temporal_key_dcl] <property_name>               21
                          [,<property_list>]
<property_name>       ::= <identifier>                                     22
<interface_body>      ::= <export> [<interface_body>]                      23
<export>              ::= <type_dcl> ; | const_dcl ; |                     24
                          <except_dcl> ; | <attr_dcl> ; |
                          <rel_dcl> ; | <op_dcl> ;
<inheritance_spec>    ::= <scoped_name> [,<inheritance_spec>]              25
<scoped_name>         ::= [[<scoped_name>] ::] <identifier>                26
<const_dcl>           ::= const <const_type> <identifier>                  27
                          = <const_expr>
<const_type>          ::= <integer_type> | <char_type> | <boolean_type> |  28
                          <floating_pt_type> | <string_type> |
                          <interval_type> | <instant_type> | <period_type>
                          | <scoped_name>
<const_expr>          ::= <or_expr>                                        29
<or_expr>             ::= <xor_expr> |                                     30
                          <or_expr> | <xor_expr>
<xor_expr>            ::= <and_expr> |                                     31
                          <xor_expr> ^ <and_expr>
<and_expr>            ::= <shift_expr> |                                   32
                          <and_expr> & <shift_expr>
<shift_expr>          ::= <add_expr> |                                     33
                          <shift_expr> >> <add_expr> | <shift_expr> <<
                          <add_expr>
<add_expr>            ::= <mult_expr> |                                    34
                          <add_expr> + <mult_expr> | <add_expr> -
                          <mult_expr>
<mult_expr>           ::= <unary_expr> |                                   35
                          <mult_expr> * <unary_expr> |
                          <mult_expr> / <unary_expr> |
                          <mult_expr> % <unary_expr>
<unary_expr>          ::= [<unary_operator>] <primary_expr>                36
<unary_operator>      ::= - | + | ~                                        37
<primary_expr>        ::= <scoped_name> | <literal> |                      38
                          ( <const_expr> )
<literal>             ::= <integer_literal> | <string_literal> |           39
                          <character_literal> | <floating_pt_literal> |
                          <boolean_literal> | <interval_literal> |
                          <instant_literal> | <period_literal>
<boolean_literal>     ::= TRUE | FALSE                                     40
<instant_literal>     ::= "<absolute_instant>" | "<relative_instant>"      41
<period_literal>      ::= "[<instant_literal>, <instant_literal>)"         42
<interval_literal>    ::= Interval "<interval_spec>"                       43
                          [<interval_calendar>]
<relative_instant>    ::= <temporal_mark> <plus_or_minus>                  44
                          <integer_literal>
<relative_mark>       ::= <identifier> | now                               45
<plus_or_minus>       ::= + | -                                            46
<absolute_instant>    ::  <instant_spec> [<instant_calendar>]              47
<instant_calendar>    ::  identifier | gregorian                           48
<interval_calendar>   ::= identifier | gregorian                           49
```

<positive_int_const> ::= <const_expr> 50

<type_dcl> ::= **typedef** <type_declarator> | <struct_type> | <union_type> | <enum_type> 51

<type_declarator> ::= <type_spec> <declarators> 52

<type_spec> ::= <simple_type_spec> | <constr_type_spec> 53

<simple_type_spec> ::= <base_type_spec> | <template_type_spec> | <scoped_name> 54

<base_type_spec> ::= <floating_pt_type> | <integer_type> | <char_type> | <boolean_type> | <octet_type> | <interval_type> | <instant_type> | <period_type> | <any_type> 55

<template_type_spec> ::= <array_type> | <string_type> 56

<constr_type_spec> ::= <struct_type> | <union_type> | <enum_type> 57

<declarators> ::= <declarator> [**,** <declarators>] 58

<declarator> ::= <simple_declarator> | <complex_declarator> 59

<simple_declarator> ::= <identifier> 60

<complex_declarator> ::= <array_declarator> 61

<floating_pt_type> ::= **Float** | **Double** 62

<integer_type> ::= <signed_int> | <unsigned_int> 63

<signed_int> ::= <signed_long_int> | <signed_short_int> 64

<signed_long_int> ::= **Long** 65

<signed_short_int> ::= **Short** 66

<unsigned_int> ::= <unsigned_long_int> | <unsigned_short_int> 67

<unsigned_long_int> ::= **Unsigned Long** 68

<unsigned_short_int> ::= **Unsigned Short** 69

<char_type> ::= **Char** 70

<boolean_type> ::= **Boolean** 71

<octet_type> ::= **Octet** 72

<interval_type> ::= **Interval**
[**granularity** <granularity>]
[**calendar** <calendar>] 73

<instant_type> ::= **Instant**
[**granularity** <granularity>]
[**calendar** <calendar>]
[**absolute** | **relative**] 74

<period_type> ::= **Period**
[**granularity** <granularity>]
[**calendar** <calendar>]
[**absolute** | **relative**] 75

<any_type> ::= **Any** 76

<struct_type> ::= **Struct** <identifier> **{**<member_list>**}** 77

<member_list> ::= <member> [<member_list>] 78

<member> ::= <type_spec> <declarators> **;** 79

<union_type> ::= **union** <identifier> **switch (**<switch_type_spec>**)** **{**<switch_body>**}** 80

<switch_type_spec> ::= <integer_type> | <char_type> | <boolean_type> | <enum_type> | <scoped_name> 81

<switch_body> ::= <case> [<switch_body>] 82

<case> ::= <case_label_list> <element_spec>**;** 83

<case_label_list> ::= <case_label> [<case_label_list>] 84

<case_label> ::= **case** <const_expr> **:** | **default :** 85

<element_spec> ::= <type_spec> <declarator> 86

<enum_type> ::= **Enum** <identifier> **{**<enumerator_list>**}** 87

<enumerator_list> ::= <enumerator> [**,**<enumerator_list>] 88

<enumerator> ::= <identifier> 89

<array_type> ::= <array_spec> **<** <simple_type_spec> [**,**<positive_int_const>] **>** 90

<array_spec> ::= **Array** 91

<string_type> ::= **String** [**<** <positive_int_const> **>**] 92

<array_declarator> ::= <identifier> <array_size_list> 93

<array_size_list> ::= <fixed_array_size> [<array_size_list>] 94

<fixed_array_size> ::= **[** <positive_int_const> **]** 95

<attr_dcl> ::= [**readonly**] **attribute** <domain_type> <attribute_name> [fixed_array_size] [temporal_dcl] 96

<domain_type> ::= <simple_type_spec> | <struct_type> | <enum_type> | <attr_collection_specifier> <literal> | <attr_collection_specifier> <identifier> 97

<attr_collection_-specifier> ::= **Set** | **List** | **Bag** | **Array** 98

<rel_dcl> ::= [**state**] **relationship** <target_of_path> <identifier> [<temporal_dcl>] [**inverse** <inverse_traversal_path>] **{**[<order_by>]**}** 99

<order_by> ::= **order_by** <attribute_list> | **order_by** [**valid**] [**transaction**] 100

<target_of_path> ::= <identifier> | <rel_collection_type> **<**<identifier>**>** 101

<inverse_traversal_path> ::= <identifier> :: <identifier> 102

<attribute_list> ::= <scoped_name>[**,** <attribute_list>] 103

<rel_collection_type> ::= **Set** | **List** | **Bag** | **Array** 104

<except_dcl> ::= **exception** <identifier> **{**[<member_list>]**}** 105

<op_dcl> ::= [**oneway**] [**state**] <op_type_spec> <identifier> <param_dcls> [<raises_expr>] [<context_expr>] [temporal_dcl] 106

<op_type_spec> ::= <simple_type_spec> | **void** 107

<param_dcls> ::= **(** [<param_dcl_list>] **)** 108

<param_dcl_list> ::= <param_dcl> [**,** <param_dcl_list>] 109

<param_dcl> ::= <param_attribute> <simple_type_spec> <declarator> [temporal_dcl] 110

<param_attribute> ::= **in** | **out** | **inout** 111

<raises_expr> ::= **raises** (<scoped_name_list>) 112

<scoped_name_list> ::= <scoped_name> [**,**<scoped_name_list>] 113

<context_expr> ::= **context** (<string_literal_list>) 114

<string_literal_list> ::= <string_literal> [**,**<string_literal_list>] 115

<calendar_definition> ::= **calendar** <calendar_name> **{origin** <instant_literal>**;** **granules** <granules_list>**;}** 116

<calendar_name> ::= <identifier> 117

<granules_list> ::= <granule_name> **:** **{**<mappings>**}** [**,** <granules_list>] 118

<granule_name> ::= <identifier> 119

10. Reserved words

In the following tables the reserved words of TODL are presented:

absolute	any	array	attribute
bag	boolean	calendar	case
char	chronon	const	context
day	default	double	enum
event	exception	extent	false
float	granularity	granules	gregorian
hour	in	out	inout
input	instant	interface	interval
inverse	key	keys	list
long	module	month	now
object	octet	oneway	order_by
origin	out	output	overlaps
period	persistent	point	raises
readonly	relationship	relative	second
sequence	set	short	state
string	struct	switch	transaction
transient	true	typedef	union
unsigned	valid	void	year

11. Multiple inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, inst-nce properties, exceptions and operations). In addition, unless rede-fined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::") may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any type, constant and exception names which have been inherited; the scope rules for such names are described in Paragraph 7. Scope Rules.

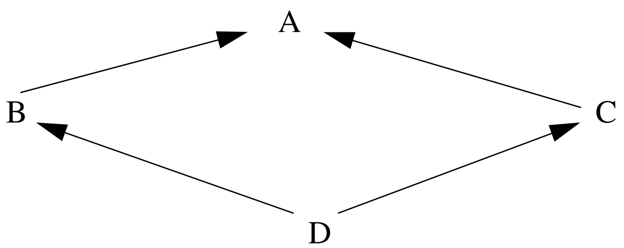
An interface is called a direct base if it mentioned in the inheritance specification and an inderect base if it is not a direct base, but is a base interface of one of the interfaces mentioned in the inheritance specification.

An interface may be derived from any number of interfaces. Such use of more than one direct base interface is often called *multiple inheritcance*. The order of derivation is not significant.

Any interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A {...}
interface B: A {...}
interface C: A {...}
interface D: B, C {...}
```

The relationships between these interfaces are shown in the following figure:



Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type or exception in more than one base interface. (It is currently illegal to inherit from two interfaces with the same operation or instance property name, or to redefine this operation or instance property in the derived interface). Ambiguities can be resolved by qualifying a name with its interface name (that is, using a scoped name).

Reference to constants, types and exceptions are bound to an interface when it is defined, i.e., replaced with the equivalent global scoped names. This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface.

Interface inheritance causes all identifiers in the closure of the inheritcance tree to be imported into the current naming scope. A type name, constant name, enumeration value name or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification is a compilation error.

12. References

- Common Object Request Broker: Architecture & specification, Rev. 2.0, July 1995, Update July 1996
- TODL Specifications - TOOBIS Deliverable
- TODM Specifications - TOOBIS Deliverable