TOOBIS

TODL Specifications

Deliverable T32TR.1

UNIV. OF ATHENS

01-PLIROFORIKI S.A.

O_2 Technology

National Technical Univ. of Athens

November 1996

1. SPECIFICATION 5

1.1 INTRODUCTION/ OBJECTIVES 5
1.2 INSTANCE PROPERTIES (ATTRIBUTES/RELATIONSHIPS) WITH TEMPORAL CHARACTERISTICS 5
1.2.1 Attributes 6
1.2.2 Relationships 7
1.2.3 Operations 7
1.3 CLASSES WITH TEMPORAL CHARACTERISTICS 8
1.4 OVERLAPPING VALID TIME 8
1.4.1 Kinds of overlapping valid time timestamps 8
1.4.1.1 Overlapping valid time timestamps for the same object 8
1.4.1.2 Overlapping valid time timestamps for object in the same extent 9
1.4.2 Examples of overlapping valid time timestamps 9
1.4.2.1 Overlapping valid time timestamps of variants of the same object 9
1.4.2.2 Non-overlapping valid time timestamps of variants of the same object 10
1.4.2.3 Inter-object period keys 10
1.4.2.4 Inter-object point keys 11
1.4.2.5 Extent-wide point keys 11
1.5 USER-DEFINED TIME 11
1.5.1 Unanchored quantity of time: Interval 12
1.5.2 Anchored quantity of time: Instant 12
1.5.3 Anchored quantity of time: Period 13
1.6 STATE-RELATIONSHIPS 13
1.7 CALENDAR DEFINITION 14
1.8 LINK TO USER REQUIREMENTS 15

2. DESIGN 16

2.1 INTRODUCTION 16
2.2 PARSER ARCHITECTURE 17
2.3 SEMANTIC CONSIDERATIONS 18
2.3.1 Key definition 18
2.3.2 Calendar and granularity definition 18
2.3.3 Valid time declaration default values 18
2.3.4 Temporal relationships and inverse links 19
2.4 MAPPING OF TODL CONSTRUCT TO O_2 AND TODM 19
2.4.1 Mapping of TODL constructs to O_2 19
2.4.2 Mapping of TODL constructs to TODM 20
2.4.2.1 Mapping temporal attributes 20
2.4.2.2 Mapping temporal relationships 22
2.4.2.3 Mapping temporal objects 23
2.4.2.4 Mapping state relationships 23
2.5 METADATA 24
2.5.1 TemporalMetaObject 25
2.5.2 TemporalMetaProperty 27
2.5.3 TemporalMetaAttribute 27
2.5.4 TemporalMetaRelationship 28
2.5.5 TemporalMetaOperation 28
2.5.6 TemporalMetaType 29
2.5.7 TemporalMetaParameter 29
2.5.8 TemporalMetaCalendar 29
2.5.9 TemporalMetaGranule 29
2.5.10 TemporalMetaExternalFormat 30

3. ODL 2.0 COMPLIANCE 31

4. CONCLUSIONS 31

APPENDIX A: TODL BNF SYNTAX 32

APPENDIX B: LANGUAGE COMPLETENESS WITH RESPECT TO THE USER REQUIREMENTS....36

1. DELTA APPLICATION....36

1.1 TYPE DEFINITIONS....36
1.2 CLASS PRODUCT....37
1.3 CLASS CUSTOMER....38
1.4 CLASS DISTRIBUTOR....38
1.5 CLASS BATCH....39
1.6 CLASS CUSTOMERDELIVERY....39
1.7 CLASS ADDITIONALORDER....39
1.8 CLASS CUSTOMERRETURN....40
1.9 CLASS CUSTOMERSTOCK....40
1.10 CLASS CUSTOMERORDER....40
1.11 CLASS DELIVERYPACKAGE....41
1.12 CLASS DAILYPRODUCTORDER....41
1.13 CLASS TARGET....41
1.14 CLASS DAILYTOTAL....42
1.15 CLASS DAILYCIRCULATION....42
1.16 CLASS WEATHERCONDITION....42
1.17 CLASS DAILYPRODUCTION....43
1.18 CLASS PRODUCTIONSTATISTIC....43
1.19 CLASS ADDRESSLOCATION....43
1.20 CLASS AREA....44
1.21 CLASS EVENT....44

2. GLAXOWELLCOME APPLICATION....44

2.1 TYPE DEFINITIONS....45
2.2 CLASS PATIENT....45
2.3 CLASS OBSERVATIONNOTEBOOK....46
2.4 CLASS INDIVIDUALNOTEBOOK....46
2.5 CLASS THEORDIAGRAM....47
2.6 CLASS DEMOGRAPHIC....47
2.7 CLASS LIFECONDITIONS....47
2.8 CLASS ILLNESSHISTORY....48
2.9 CLASS OBSERVATIONSTATEMENT....48
2.10 CLASS SEXUALMATURITY....48
2.11 CLASS PATHNONALLERGIC....49
2.12 CLASS PATHOLOGYALLERGIC....49
2.13 CLASS FUNCTRESPTEST....49
2.14 CLASS PHYSEXAM....50
2.15 CLASS VERIFICATIONELIGIBLE....50
2.16 CLASS STAT1WEEK....50
2.17 CLASS STAT4WEEK....51
2.18 CLASS VISITSTATEMENT....51
2.19 CLASS MOTIVESADDITIONAL....51
2.20 CLASS TRIALEND....52
2.21 CLASS TRIALTREATMENT....52
2.22 CLASS CONCOMITANTTREAT....52
2.23 CLASS UNDESIRABLEEVENT....52

APPENDIX C: EXAMPLES FOR EXTENDED FUNCTIONALITY....53

1. SUPPLIER-PART-JOB....53

2. COUNTRIES....54

3. EMPLOYEES - SALARIES....54

APPENDIX D: REFERENCES - BIBLIOGRAPHY 56

1. Specification

1.1 Introduction/ Objectives

The Temporal Object Definition Language (TODL), a temporal extension to ODL, is a specification language used to define interfaces for objects that will conform with TODM (Temporal Object Data Model). The same principles that have guided the development of ODL have been followed for the development of TODL:

- TODL should support all semantic constructs of TODM, which in turn should be compatible with the ODMG Object Model.
- TODL should not be a full programming language, but rather a specification language for interface signatures.
- TODL should be programming language independent.
- TODL should be compatible with ODL.
- TODL should be extensible, not only for future functionality, but also for physical optimizations.

As stated above TODL should not be a full programming language, but rather a specification language for interface signatures. Actually, TODL should be a DDL language used for object types and providing for temporal functionality for those object types. Through TODL it is possible to define, along with the object type, the signatures of the operations that can be applied on it. However, the implementation of those operations is out of the scope of TODL. We do not define a standard TOML (Temporal Object Manipulation Language), in the same way that ODMG-93 does not provide a standard OML.

Also TODL is intended to define temporal functionality and object types that can be implemented in a variety of programming languages. In this way the schema is defined regardless of the programming language used, something that permits the user to use the language he prefers for method implementation.

1.2 Instance properties (attributes/relationships) with temporal characteristics

According to ODMG an instance property is either an attribute or a relationship. The difference between an attribute and a relationship is that the latter is an object, whereas the former is a literal[1]. A relationship may be one-to-one, one-to-many or many-to-many, and a inverse relationship may also be defined.

Each instance property may have temporal characteristics, regardless of the class status. In the following paragraphs we define the different temporal characteristics that exist and we show how they apply to the instance properties.

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 word. An instance property with valid time characteristics 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

[1] In draft 2.0 of ODL however, an attribute may be an object. More information on draft 2.0 of ODL and how it may affect the TODL specification and design are presented in a following section.

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 to the 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 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 a time period with the transaction time of the fact the property represents.

We say that an instance property has bitemporal characteristics 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 the following sections we show how these characteristics are defined to hold for a specific property instance.

1.2.1 Attributes

We can define that an attribute has temporal characteristics by using the **valid** and **transaction** clauses.

The **valid** clause is used to define that the attribute in question has valid time characteristics. The time during which the attribute had a specific value may be represented via a timestamp which we can either be a time period or a time point. The former representation is defined using the keyword **state** while the latter using the keyword **event**. When periods are used for variant timestamping, overlapping of timestamps is disallowed by default; the user may, however, override the default by using the **overlaps** keyword which allows the overlapping of the timestamps. The valid timestamp defined over an attribute may be represented in one of different granularities; the desired granularity may be selected using the **granularity** clause. Also the user may define the calendar at which the valid time will be recorded. At this point we use the **gregorian** calendar with six granularities that are specified by the keywords: **second**, **minute**, **hour**, **day**, **month** and **year** are predefined. Other calendars and their associated granularities may be defined using an extension to the data model that would allow the definition of new calendars.

The **transaction** clause is used to define that the specified attribute should have transaction time semantics, that is all the values that at some time point have been stored in the database are not physically deleted from it, but are kept in the object variants along with the associated timestamps.

Some examples for the usage of the above clauses follow:

An attribute with valid time characteristics of period type with year granularity at the default calendar (Gregorian)

```
attribute TypeA attr_name valid granularity year
```

(Note that the period type is the default type for valid time attributes, so the **state** designation, which could be positioned after the **valid** keyword is omitted. The same holds for the calendar specification).

An attribute with overlapping valid time characteristics of period type, with seconds granularity at the default calendar.

```
attribute TypeB attr_name valid state overlaps
      granularity second calendar gregorian
```

(Note that in this case the default values designated by the **state** and **calendar** keywords are supplied)

An attribute with transaction time characteristics.

```
attribute TypeC attr_name transaction
```

An attribute with valid time of instant type, with minutes granularity at the default calendar and transaction time.

```
attribute TypeD attr_name valid event
      granularity minutes
      transaction
```

1.2.2 Relationships

A relationship specification names and defines a traversal path between two objects. As in ODMG, a traversal path definition includes designation of the target type, ordering information and information about the inverse traversal path found in the target type. As with attributes a relationship may have temporal information defined on it with the same semantics as above. The same temporal functionality that was described for attributes is also available for relationships. Thus, a relationship may have valid time, transaction time or bitemporal semantics, which are defined using the **valid** and **transaction** clauses, as they were described above.

Traversal path cardinality information, as in ODMG, is included in the specification of the target of a traversal path. The target type must be specified with its own type definition, unless the relationship is recursive. A collection indicates cardinality greater than one on the target side.

Also an ordering criterion is specified using the **order_by** clause. Each attribute in an ordering criterion must be defined in the attribute list of the target type's definition. If the relationship has also temporal characteristics it is possible to define that the relationship will be ordered over the timestamp used to specify the temporal characteristic. This can be specified using the statements **order_by valid** and **order_by transaction** for ordering on valid or transaction time timestamps, respectively.

If for instance a relationship with valid time and transaction time characteristics should be ordered on the valid time axis this would be specified by the used using the following:

```
relationship AnObject rel_name valid transaction
      {order_by valid}
```

Note that all the specifications of valid time are omitted from the **valid** clause, implying that the valid time type is period, the granularity is seconds and the calendar to be the Gregorian.

If a relationship has temporal characteristics then its inverse should have exactly the same temporal characteristics. A relationship and its inverse are regarded as mirror images of the same relationship.

As with ODMG, the attribute and relationship specifications can be mixed in the property list.

1.2.3 Operations

The definition of the operations is compatible with the ODL specification of operations with the difference that in the parameter declaration the user may specify that one or more of the operation's parameters and/or the result of the operation have temporal characteristics. This is done by using the statements **valid** and/or

transaction, as they were described above, immediately after the parameter's type specification. The same designations may be applied to the operation's result type, in which case they are shown at the end of the operation definition.

Another TODL specific characteristic of operations is that they might be defined to work over the state of an object. This can be defined by prepending the **state** keyword to the operation name, before the operation result type.

1.3 Classes with temporal characteristics

If all the object properties have the same temporal characteristics, TODL provides for a shortcut to the definition of these characteristics. Similar to the persistence declaration in ODL, we may include a temporal definition, at the same level in TODL's BNF. This shortcut can be used in two cases:

- when the objects of this class should be treated as one temporal object, evolving as a whole with time, and
- when all the attributes and relationships of an object share the same temporal characteristic, but some of them have some extra or individual temporal characteristic.

In the latter case the redefinition of the temporal characteristics of a specific property should be taken into account. There are the following kinds of problems, that can be viewed at this point:

a. The class is defined to be valid (transaction) and a property is defined to be transaction (valid).
b. The class is defined to be valid with specific characteristics (timestamp type, overlapping, granularity) and a property is defined to be valid, but with different characteristics.

The solution differs in each case. In case (a) the property in question will be defined to be bitemporal, i.e. the local definition complements the general one. In case (b) the property definition will supersede the class definition, for this property.

Note that cases in which a redefinition of calendar takes place are not accepted by TODL.

1.4 Overlapping valid time

Valid time timestamps may be overlapping or not depending on the semantics of the object under definition. Different levels of overlapping valid time periods may be allowed and these different levels will be specified through different constructs in the TODL language. In what follows we will show the different kind of overlapping timestamps that are allowed and then the means to specify them.

1.4.1 Kinds of overlapping valid time timestamps

1.4.1.1 Overlapping valid time timestamps for the same object

In this case we either want to allow for overlapping valid time periods between variants of the same object, or we don't want to do so. This can be done by using the keyword **overlaps** in the valid time definition either at the property level or at the object level, when we allow for overlapping valid time periods. If this keyword is omitted then valid time periods are not allowed to overlap.

Note that in case of redefinition of the valid time at the property level, the property level definition will supersede the object level definition. Also note that in the case that the valid time is of event type, then we have a trivial case of overlapping valid time timestamps.

1.4.1.2 Overlapping valid time timestamps for object in the same extent

In ODMG the user may specify a number of keys for the object under definition. These keys, extended as temporal keys, can be used to declare whether valid time timestamps for objects that belong in the same extent may be overlapping and in what extent. For this reason we define three kinds of temporal keys.

- *inter-object period key*: two distinct objects may have the same value at this attribute (relationship) for overlapping periods, but not for exactly the same period. Also, the valid time timestamps of the variants of the same may be overlapping.
- *inter-object point key*: two distinct objects cannot have the same value in this attribute (relationship) at any time point, i.e. overlapping periods with the same value are not allowed. This is the default behaviour for any property that has temporal characteristics and is defined to be part of the key. Note that the valid time timestamps of the variants of the same object may not overlap.
- *extent-wide point key*: no two objects may have the same value at this attribute (relationship) at any time point, that is if value *a* is given for a time period to object A, then no other object may have the same value at any time point. Note that the valid time timestamps of the variants of the same object may not overlap.

1.4.2 Examples of overlapping valid time timestamps

1.4.2.1 Overlapping valid time timestamps of variants of the same object

Consider a class Country, that has an attribute Inflation_rate, which has valid time characteristics of type state and granularity month. We define the attribute Country_name to be a key (with the meaning it has in ODMG). Attribute Inflation_rate may have overlapping valid time periods for variants of the same object as long as the exact period cannot be used with two different values, this will be defined using the keyword **overlaps** at the end of the valid time definition. This means that for a specified country, say Greece, we may have an object with the following variants (only two of the attributes are shown):

Country_name	Inflation_rate	Valid Time Period
Greece	7.8%	[1995-01, 1996-01)
Greece	8.8%	[1995-01, 1995-07)
Greece	6.8%	[1995-07, 1996-01)
Greece	9.5%	[1996-01, now)

Note that the periods that refer to the inflation rates are overlapping. This is totally acceptable, since the inflation rate depends on the time period on which it is measured as a whole, and has no meaning for subperiods. However an object variant with value for the Inflation_rate attribute equal to, say, 9.9% and valid time period equal to [1995-01, 1995-07) would be unacceptable, since a value for this time period is already defined.

Also note that any other country (which is represented with a different object) can have for the same time period a different inflation rate. That is it is totally acceptable to have an object variant with values Country_name = “France”, Inflation_rate = 9.4% and valid time period [1995-01, 1995-07).

1.4.2.2 Non-overlapping valid time timestamps of variants of the same object

Consider a class Employee having an attribute Salary, which has valid time information, of type state and granularity month. We also define an attribute Employee_name that identifies the employee, so it is defined as a key. We also define that for the attribute Salary no overlapping valid time timestamps are allowed for variants of the same object. No extra keyword is used to define this, as it is the defaut behavior for valid time properties. Let’s assume that an employee named Jones exists and that the following data are already recorded for him.

Employee_name	Salary	Valid Time Period
Jones	12000	[1992-01, 1993-01)
Jones	14000	[1993-01, 1995-07)
Jones	16000	[1995-07, 1996-01)
Jones	17000	[1996-01, now)

Having the above data defined for Jones we cannot add another object variant for him with Salary, say, 15000 and valid time period [1994-01, 1994-06). This holds because an employee cannot have different salaries for the same time point. However the object with Employee_name = Johnson, Salary = 12000 and valid time equal to [1992-01, 1992-09) is acceptable as the salaries of two distinct employees are not, usually, dependent on each other.

Note that if the class has been defined to have bitemporal characteristics, then if the above mentioned operation was a correction, it would be accepted resulting to the following: (assuming that the database was created at 1/1/1992 at midnight, that the erroneous value was inserted at 14/1/1994 13:30:00 and that the correction took place at 1/2/1994 at 12:00:00)

Employee_name	Salary	Valid Time Period	Transaction Time Period	Flag
Jones	12000	[1992-01, 1993-01)	[1992-01-01 00:00:00, UC)	E
Jones	14000	[1993-01, 1994-01)	[1994-01-01 12:00:00, UC)	E
Jones	15000	[1994-01, 1994-06)	[1994-01-01 12:00:00, UC)	E
Jones	14000	[1994-01, 1995-07)	[1994-01-14 13:30:00, 1994-01-01 12:00:00)	C
Jones	16000	[1995-07, 1996-01)	[1995-07-16 14:35:45, UC)	E
Jones	17000	[1996-01, forever)	[1996-01-18 10:08:00, UC)	E

1.4.2.3 Inter-object period keys

If we define that a property is an inter-object period key, we mean that

a) for any single object, we allow variants of this attribute to have overlapping, but not equal timestamps
b) for any two distinct objects in the class extent, variants of this attribute with equal values and timestamps are disallowed.

This type of key is specified using the keyword **period** and is defined for completeness purposes, only.

1.4.2.4 Inter-object point keys

Let's assume a class Department, with an attribute Department_name, with valid time characteristics of type state and granularity day. This class has also an attribute Department_id which is a key for this class. We also specify that the Department_name is an inter-object point key, by using the keyword **point**. We assume that the following data are stored in the database:

Department_id	Department_name	Valid Time Period
1	Toys	[1992-01, 1993-01)
1	Toys and Books	[1993-01, 1995-07)
1	Toys	[1995-07, 1996-01)
2	Books	[1995-07, 1996-01)

We notice that the name of Department 1 has changed during time from "Toys" to "Toys and Books" and then back to "Toys" creating another department "Books". If we try to specify that Department 2 was also valid during [1994-01, 1995-07) with the name "Toys and Books" this operation should be not allowable, as only one department can have a certain name at any time point.

Thus, for inter-object point keys we have the following restrictions:

a) for any single object, variants of this attribute with overlapping timestamps are disallowed

b) for any two distinct objects in the class extent, variants of this attribute with equal values and overlapping timestamps are disallowed.

1.4.2.5 Extent-wide point keys

Extent-wide point keys are in a way an extension to inter-object point keys. Imagine that we want the Department_name to be unique throughout the time. Consider the case of the tax record numbers: tax payer (employee or company) may have different tax record numbers throughout his period of interaction with the revenue service (only one tax record number at any given time), but this number may not be reassigned to any other tax payer at any future time point. These restrictions are implied if the tax record number is declared as an *extent-wide point key*. Such a key would be defined using the keyword **object** in the keys' definition statement.

Extent-wide point keys obey to the following restrictions:

a) for any single object, variants with equal values for this attribute are not allowed.

b) for any two distinct objects in the class extent, variants with equal values are disallowed.

Those are the semantics of the keys, as specified above. Note that the uniqueness property should be preserved at the data model level, as TOQL only has access to what can be retrieved from the extents and not to what is stored in them.

1.5 User-defined time

While defining an object, the user has the ability to define that one or more of the attributes may have a type that is used to represent time. Such types have already been defined in TODM [T31TR1] and the user is allowed to use the type that he feels it represents in the most appropriate way the information he wants to keep. This is the way the user may keep what is defined as "user-defined time" in the temporal

community, i.e. time that has specific semantics which are known only to the application and cannot be dealt by the DBMS. Examples of user-defined time is the date of birth of an employee, the interval in days during which a dairy product may be consumed etc.

In what follows the different types that can be used for user-defined are presented and some examples are given for the usage of each such data type. Note also that as far as user-defined time is concerned the user may decide that the time can be relative, absolute or both. The default will be for the attribute to accept both relative and absolute timestamps. If the user however wishes that the specified attribute may accept only relative (absolute) timestamps, then he can declare that by using the **relative** (**absolute**) keyword. Examples of this will be presented in the following sections:

1.5.1 Unanchored quantity of time: Interval

TODM has introduced data type **Interval** to represent unanchored quantities of time. This data type may be used whenever the user wants to represent values as "7 days" without defining a starting or ending point. Its interval has its own granularity and the user may specify it explicitly, or allow the attribute to have the default granularity, which is defined to be granularity of seconds. Also each interval has a specified calendar, which is either declared explicitly by the user or is defined to be the default, which in our case is the Gregorian calendar.

If the user wants to define an attribute of type Interval and granularity of month then this should be defined as follows:

```
attribute Interval granularity month month_Interval
```

Note that as the Interval data type is by definition unanchored the concept of relativity is inherent to it, so declarations of absolute or relative timestamping are not needed at this point.

In the case the user wants to specify that an attribute is of type Interval, with granularity day in the Julian calendar (provided that the user has defined the Julian calendar either working at the TODM level, or specifying it in TODL) then this will be declared as follows:

```
attribute Interval granularity month calendar Julian month_Julian
```

1.5.2 Anchored quantity of time: Instant

The Instant data type specified in TODM to represent points on the time line may be used by the user. In this case, as with the Interval, the user may define the granularity and the calendar used for the instant, otherwise the default values of granularity of second and Gregorian calendar will be used. Also in the case of the Instant data type the notions of relative and absolute timestamps may be applied.

Let's assume that a user wants to have an attribute of type Instant with day granularity, in order to record the dates of birth of the employees in a company. As dates of birth are always absolute instants then he may declare such an attribute as follows:

```
attribute Instant granularity day absolute birth_date
```

If the user wants to declare an attribute of Instant data type (using the default granularity and calendar) which will have only relative time information then this declaration will be as follows:

```
attribute Instant relative relative_instant
```

Finally if the user does not care whether the information present in an attribute of Instant data type is of relative or absolute time then the definition may be (provided

that he is going to allow for granularity of seconds and the Gregorian calendar) the following:

```
attribute Instant some_instant
```

Note that the user may - instead of the Instant data type which is provided to him by the TODM - use the Date or Time or Timestamp data types that are provided by ODMG. However, this would be somewhat restrictive as he won't have the ability to choose a different granularity and/or calendar, as well as the usage of relative timestamping.

1.5.3 Anchored quantity of time: Period

The user may finally use as data type of an attribute the Period data type, which allows him to declare that an attribute is a period of time, with specified granularity and expressed at a specified calendar. The way to declare an attribute of type Period is the same as to declare an attribute of type Instant.

Thus if the user wants to declare an attribute of type Period and granularity year at the Julian calendar, then this will be achieved through the following declaration:

```
attribute Period granularity year calendar julian some_period
```

If the attribute should only have periods defined as relative time, at the default calendar, with year granularity then this will be achieved with the following declaration:

```
attribute Period granularity year relative some_period
```

Note that the defaults here - as in the previous cases - are

(a) default granularity of seconds

(b) default calendar Gregorian

(c) by default the attribute allows for both relative and absolute time values.

1.6 State-relationships

As it was shown in TODM we need relationships towards states of a temporal object. This means that in some cases it is preferable to access certain "slices" of a temporal object instead a whole one. This need lead to the definition - at the TODM level - of state-relationships. The access to these "slices" - more formally called *states* - is made through a kind of temporal object identifier, which points to states of a temporal object which are identified by having the same valid time or transaction time timestamp. Actually in TODM this kind of temporal object identifier may (depending on the kind of the temporal object) one instant (for historical with no overlapping objects or rollback objects), a period (for historical with overlapping objects), a couple of instants (for bitemporal with no overlapping objects) or a period coupled with an instant (for bitemporal with overlapping objects).

Note that state-relationships are allowed only towards temporal objects, that is objects that have been defined in TODL using the valid or transaction time definition at the object level, with no redefinition of the temporal properties at the instance properties' level. Also note state relationships do not have inverse links.

The user may specify that a relationship is a state relationship by prepending the keyword **state** in the *classical* relationship definition. By using this approach we may define that the valid time timestamps of a state relationship may or may not be overlapping in the same way as with *classical* relationships. They may also participate in the keys definition allowing in this way the definition of point, period and extend-

wide state keys. No extra keyword is required at this level, as the state nature of the relationship is provided by its definition.

1.7 Calendar definition

In TOOM [T23D1.1b] it is possible to have a way to define a new calendar. We are going to supply the user with the syntactic constructs that may be used to define a calendar, but of course the user will be responsible in such a case to define the different operations needed for the granule conversions within the newly defined calendar or for conversions among different calendars. The user might want to define a new calendar if he sees that it fits to the needs of his application.

A calendar can be described by its name, an instant that is considered to be the origin of the calendar, a set of granules that can be mapped regularly or irregularly to each other, the redefinition of three operations (namely: addition of an interval to an instant, addition of an instant to an interval and extraction of an interval from two instants), and a set of external formats for mapping to other calendars. All these may be defined through TODL,except from the redefinitions of the three operations which should be made at a manipulation language level. Also the external formats and the mapping between the granules will be provided as operation names, but the actual implementation should be made outside TODL, using a manipulation language.

The BNF used for calendar definition should be the following (with bold we mark the keywords):

```
<calendar_definition>    ::   calendar <calendar_name>
                         =    {
                                  origin <instant_literal>;
                                  granules <granules_list>;
                                  [external <external_format_list>;]
                              }
<calendar_name>          ::   <string_literal>
                         =
<granules_list>          ::   <granule_name> :
                         =    {<mappings>}
                              [, <granules_list>]
<granule_name>           ::   <string_literal>
                         =
<mappings>               ::   chronon |
                         =    <integer_literal> <granule_name> |
                              <mapping_function_name>,
                              <mapping_function_name>
<mapping_function_-      ::   <string_literal>
name>                    =
<external_format_list>   ::   <format_name> = "<format_definition>"
                         =    {<mapping_definition>}
                              [, <external_format_list>]
<format_name>            ::   <string_literal>
                         =
<format_definition>      ::   <string_literal>
                         =
<mapping_definition>     ::   <return_type> input(<format_name>),
                         =    <format_name> output(<return_type>)
```

An example is the definition of the Gregorian calendar using TODL:

```
calendar Gregorian
{
      origin "01-01-0001, 00:00:00";
      granules
            second : {chronon}
            minute : {minute_second(), second_minute()}
            hour : {60 minute}
            day : {24 hour}
            month : {month_day(), day_month()}
            year : {12 month};
      external
            TimeZone = "instant<Gregorian, hour>, TZ, DST"
                  {instant<Gregorian, hour> input(TimeZone),
                   TimeZone output(instant<Gregorian, hour>)}
}
```

in which case the external format is the local time specified using the timezone of the instant. The operations: minute_second, second_minute, month_day, day_month and input and output (with the appropriate redefinitions for each new external format) should be defined by the user in a manipulation language. Note that in cases where the user is defining his own calendars, he is responsible for the internal representation.

1.8 Link to user requirements

As it was shown in [T22TR21] the user requirements that affect TODL can be summarised as follows:

Characteristics	GLAXO application	DELTA application
temporal dimension	valid time & **transaction time**	valid time
typology of objects	snapshot objects, **rollback objects, bi-temporal objects**	snapshot objects **historical objects**
temporal structure (storage)	temporal point	temporal point, **period**
temporal abstract data-types	period, interval, date, timestamp, **relative time**	period, interval, date, timestamp
granularity & calendar	calendar : default granularity : default, several list of granularities : day, minute	calendar : default, granularity : default, several list of granularities : day, minute

Table 1 : synthesis of the applications analysis

Valid time can be defined using TODL by using the **valid** clause and transaction time using the **transaction** clause. Both these keywords can be used along with the definition of an attribute or relationship, thus yelding an historical, rollback or bitemporal attribute / relationship. When used at object level, these extensions define historical, rollback or bitemporal objects.

Snapshot objects are the default object type and are used for classes that do not have temporal characteristics at class level. Historical objects are defined using the keyword **valid** immediately after the declaration of class properties, i.e. outside the definition of attributes and relationships, provided that no redefinition of the time

dimension is made for any attribute or relationship. Similarly, rollback objects can be defined by using the keyword **transaction** immediately after the declaration of class properties, i.e. outside the definition of attributes and relationships, provided that no redefinition of the time dimension is made for any attribute or relationship. Note that if both keywords are present, and no redefinition is made, then the object will be a bitemporal one.

The type of the timestamps defined - for valid time - through the **state** and **event** keywords. The **state** keyword is used to define that valid time will be stored using the period datatype, provided by TODM. The **event** keyword is used to define that valid time will be stored using the instant data type provided by TODM. The only structure provided for storing transaction time timestamps is the period data type and the user has no means to modify it directly. If user defined time is used, user may choose either an instant or a period data type, which are provided by TODM as type of the attribute in question. Note that the choice of the temporal structures used for storage does not pose any restrictions to the temporal structures used for reasoning, as is shown in the specification of TOQL.

It is possible to define that an attribute is using one of the data types introduced by TODM, that is Interval, Instant and Period as user defined time, and also it is possible to define that an attribute with user-defined time can only accept relative time timestamps. TODL supports by default the Gregorian calendar, but also allows the definition of other calendars too. Also it allows the usage of six granularities for the Gregorian calendar, and through the definition of other calendars it is possible to define other granularities, provided that the proper mapping among the different granularities is defined.

In Appendix B the reader may find the exact TODL statements that can be used to define the classes that are specified in [T22TR21]. Also in Appendix C the user can find some more examples, that do not come from the [T22TR21] but show the functionality of TODL, that can't be shown through the examples in Appendix B.

2. Design

2.1 Introduction

As pointed out earlier, TODL is a specification language which can be used to declare classes in an object-oriented database. The various class characteristics that can be defined are - as stated above -

- the extent of the class, i.e. a name through which all the instances of the class may be accessed.
- the keys, i.e. the attributes and relationships that uniquely identify the instances of the class.
- the attributes and the relationships each instance has, as well as whether these have temporal characteristics.
- the signatures of the operations defined for this class. Note however that the implementation of each operation is not provided using TODL.
- whether the class instances are persistent or transient.
- whether the class has temporal characteristics as a whole or not.

Each one of these definitions should be mapped to the appropriate O_2 and TODM structures for the implementation of the class's definition. All these structures

will be included in a header file that will be submitted to O_2. This header file will contain all the data that are essential for the class's definition, as well as any extra "commands" that are needed for O_2 to accept and process the class.

2.2 Parser architecture

The TODL parser is shown in the following diagram

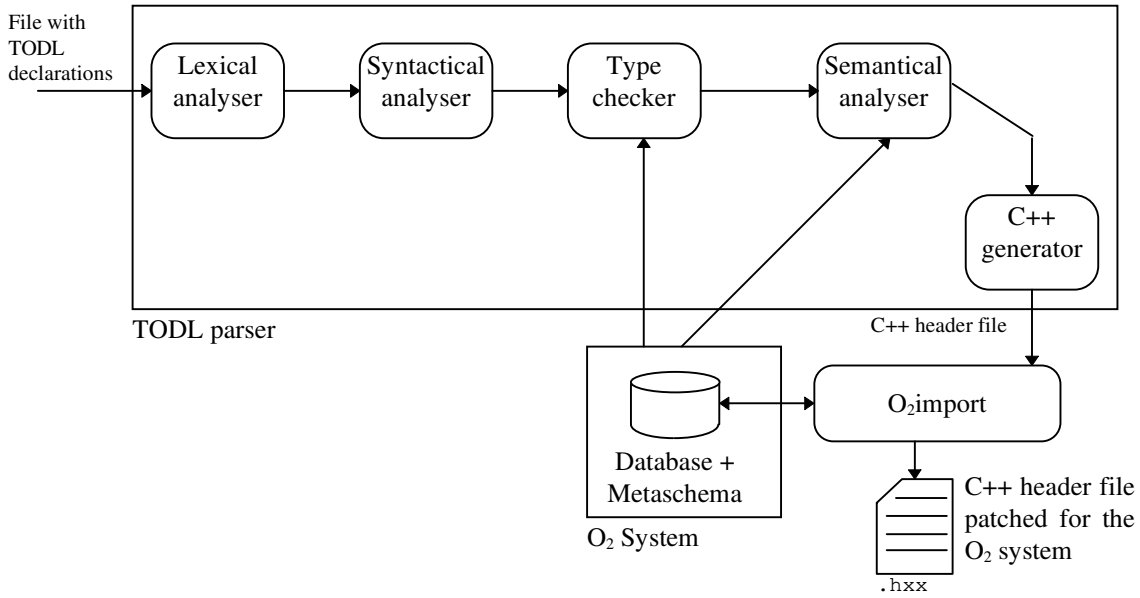

Figure 1: TODL parser architecture

The TODL parser accepts as input a file containing TODL declarations. These declarations are passed to the lexical analyser and then to the syntactical analyser, which will be based on the BNF provided in Appendix A. The results of those two phases will be passed to the type checker phase.

During type checking the TODL parser should check that every class that is referenced in the file under examination either is fully defined within this file, or is already part of the database in the O_2 System. Hence, the Type Checker should communicate with the O_2 System and exchange information with it, regarding the existence and - possibly - the exact definition of a class.

After type checking has finished, the semantical analyser is invoked. During the semantical analysis, structures that were considered valid during the previous phases are examined in order to decide whether they are valid from a semantical point of view. More details about the semantical analysis are provided in Section "Semantic Considerations", that follows.

The result of the previous phases is passed to the C++ generator which actually maps the different TODL constructs either to O_2 constructs or to TODM constructs. The C++ file generated will be used as input for the O_2Import command in order to incorporate the newly defined classes into the O_2 database and modify it appropriately for future use with the O_2 system. More details about the mapping of TODL constructs can be found in Section "Semantic Considerations", that follows.

2.3 Semantic Considerations

Certain semantic considerations must be taken into account at the semantic analysis phase of the TODL parser. These are presented in this section.

2.3.1 Key definition

Care must be taken at the *key's* definition. Consider the case in which an attribute or a relationship that does not have valid time characteristics (i.e. a snapshot or rollback attribute or relationship) is - erroneously - defined as a temporal key. This is a semantic error and it must not be accepted by the TODL parser. Another case to be considered is when the keyword **keys** is used - which actually assumes more than one keys for the object - but only one key is actually defined. In this case a warning should be issued to the user and keyword **keys** should be replaced by **key**.

2.3.2 Calendar and granularity definition

In the cases of calendar and granularity definition, the use of an identifier is accepted, but such a case would require extra work from the part of the user: the user must already have defined a new calendar along with its associated granularities, or a new granularity within an already existing calendar. So if an identifier is specified in either of these declarations then the name used must be already specified within the database, so that the implementation of the class in question will be possible.

This means two things (a) a calendar definition is already provided through TODL to the system and (b) the appropriate operations for conversion between granularities and conversion between different calendars (if any) have been implemented through a manipulation language.

Also note that in the calendar specification phase some semantics should be kept. For instance the origin of the calendar must be a valid instant for the default calendar (which is the gregorian calendar). Also note that in the granules specification, there must always be at least one granule, the finest in the calendar, which will be mapped to a chronon. For each granule coarser than that one there must be a mapping to the immediate finer granule, that is if the order of granules is "seconds, minutes, hours" then the minutes granularity should have a mapping to seconds. This mapping may be either an operation (which is the case with minutes and seconds in the Gregorian calendar, with leap seconds) or a linear mapping.

In the first case each granule should have two operations attached to it: one for mapping from the granule in question to the immediate finer granule (minutes_seconds()) and one for mapping from the immediate finer granule to the granule in question (seconds_minutes()). In the second case in the definition of the granule one should find a statement which will declare the number of granules of the immediate finer granularity that form one granule of the specified granularity (for minutes-hours mapping 60 minutes). In this case the "inverse" mapping is actually the inverse number of the integer presented here.

2.3.3 Valid time declaration default values

Valid time declaration clause might be as small as "**valid**", but can also be expanded as much as to contain information about the specified calendar and granularity and overlapping valid time timestamps. If any of the above is not present, then we use default values which are the following:

- **gregorian** calendar,

- granularity of **seconds,** and
- no overlapping allowed for variants of the same object
- no restrictions imposed for variants in different objects

Also in this clause we can define if the property is recorded as a state (so the valid time associated with it will be a time period) or an event (so the valid time associated with it will be an instant). If none of the reserved words is used then we accept as default the state nature of the property.

2.3.4 Temporal relationships and inverse links

As shown in TODM there in the typology of temporal relationships towards temporal objects the inverse relationships cannot always be defined. In the following table it is shown when the inverse relationship may be defined in a temporal relationship. Note that if a temporal relationship is allowed to have an inverse link, then this inverse link should be an exact mirror of the "forward" link.

From/To		Snapshot Object	Valid Time Object	Transaction Time Object	Bitemporal Object
Snapshot Object	Snapshot relationship	inverse exists	no inverse	no inverse	no inverse
	Valid time relationship	inverse exists	inverse exists*	no inverse	no inverse
	Transaction time relationship	inverse exists	no inverse	inverse exists*	no inverse
	Bitemporal relationship	inverse exists	no inverse	no inverse	inverse exists*
Valid time object		inverse exists	inverse exists*	no inverse	no inverse
Transaction time object		inverse exists	no inverse	inverse exists*	no inverse
Bitemporal Object		inverse exists	no inverse	no inverse	inverse exists*

* Time in these cases is specified at the object level.

2.4 Mapping of TODL construct to O_2 and TODM

2.4.1 Mapping of TODL constructs to O_2

Two TODL clauses require mapping to O_2 constructs. These are the persistence clause and the extent clause. The first one specifies whether every instance of the class to be defined is persistent by default and the second gives a name to a collection that contains every single instance of the class to be defined. Both these clauses in O_2 would mean that an O_2 name should be specified whose type would be a collection of objects of this class and that in the class's constructor an additional statement should be present, that would insert each newly created instance to this O_2 name. Also in the class's destructor a statement that would remove the destroyed instance from its associated O_2 name should be present.

The only difference in how these two clauses will be treated is that although the O_2 name to be used when the extent clause is present is known, when only a **persistent** clause is present we know nothing about the O_2 name that is going to be used. In this case a randomly named O_2 name should be defined.

Note also that if an extent clause is defined, all the instances of this class will be treated as persistent by O_2, although according to TODL they may or may not be persistent. In other words the existence of an O_2 name for a class, does not imply that all the class's instances should be persistent. This information should be kept in an external data dictionary.

2.4.2 Mapping of TODL constructs to TODM

The definition of temporal characteristics through TODL is actually implemented through mapping to TODM constructs, in the same manner that in an implementation of ODL some of its constructs should be mapped to the ODMG data model. The temporal characteristics will require the definition of a temporal object or the definition of a snapshot object that will have temporal attributes and/or relationships.

In the Temporal Object Oriented Methodology (TOOM) [T23D1.1a, T23D1.1b], the variable classes (classes that have temporal characteristics) are always associated to a permanent class (a class with no temporal characteristics). After their definition such classes may be merged at the TODL definition level in one definition. So, it is expected that in most cases only some of the attributes or relationships of a object at the TODL definition level will have temporal characteristics, and thus in most cases a snapshot object will be defined through TODM. This object will have one or more snapshot attributes of relationships, as well as some historical, rollback or bitemporal attributes or relationships.

The same holds if every attribute/relationship of the class has one time dimension (e.g. valid time dimension) and some of them also have the other time dimension (e.g. transaction time dimension), or when all the attributes/relationships have the same time dimension, but some of them have different granularity. This is the case when the shortcut at the outside level of the class definition is used, but a specialization is also defined within the attributes/relationships definition.

If all the attributes and all the relationships of the defined class have the same time dimension at the same calendar and granularity, then a temporal object is defined through TODM constructs. Depending on the time dimension the object will be a historical, a rollback or a bitemporal object.

If the class specification does not have any temporal characteristics, then a snapshot object will be defined through TODM, with all its attributes and relationships being also snapshot.

The mapping between TODL constructs and actual TODM constructs is also shown in the TODM deliverable. We are going to present some examples of such mappings in the following sections.

2.4.2.1 Mapping temporal attributes

In TODM we have defined new classes to model the temporal attributes. There is the *Temporal_Attribute* class which is used to model any temporal attribute and is a sub-class of the Attribute class defined to model any attribute in the object data model of ODMG. *Temporal_Attribute* class is specialized to three sub-classes used to map the different flavors of temporal dimensions, namely to *Historical_Attribute* (for valid time modeling), *Rollback_Attribute* (for transaction time modeling) and *Bitemporal_Attribute* (for modeling bitemporal attributes). *Historical_Attribute* and *Bitemporal_Attribute* are further more specialised to model the different variations of valid time.

When a temporal attribute is present in the definition of an interface then this attribute must be mapped to a sub-class of one of those classes and presented properly in the class definition. This is done by creating on sub-class of the appropriate class for each type of each temporal attribute. Note that by *type* of a temporal we mean the data type used in the interface definition. That is there will be a different sub-class for a rollback attribute holding floating point numbers and a different type for a rollback attribute holding strings. Note however that both sub-classes will be a sub-class of the *Rollback_Attribute* class defined previously. These sub-classes are going to be used for the final definition of the attribute within the class in TODM. The following example illustrates what is said in this paragraph.

Let *Product* one of the types used in Delta application (see Appendix B). Part of its definition is as follows:

```
interface Product
{
      attribute String productName;
      attribute Float amount valid state granularity day;
      attribute Interval duration valid state granularity day;
...
}
```

The two temporal attributes (actually historical attributes) defined at this point will be mapped to the following TODM types and used to define the Product type in TODM:

```
class Historical_state_Float_Attribute : Historical_state_Attribute
{
      Historical_state_Float_Attribute(Calendar:c, Granule:g)
                  : Historical_state_Attribute(c,g);
      set_value(Float f, Timestamp vt)
            { Historical_state_Attribute::set_value(f,vt);};
}

class Historical_state_Interval_Attribute: Historical_state_Attribute
{
      Historical_state_Interval_Attribute(Calendar:c, Granule:g)
                  : Historical_state_Attribute(c,g);
      set_value(Interval i, Timestamp vt)
            { Historical_state_Attribute::set_value(i,vt);};
}

class Product
{
      String productName;
      Historical_state_Float_Attribute amount;
      Historical_state_Interval_Attribute duration;

Product()
{ // initialisation of calendar and granularities
amount = new Historical_state_Float_Attribute('gregorian', 'day');
duration= new Historical_state_Interval_Attribute('gregorian','day');
}
...
}
```

Note that as stated in TODM the sub-classes defined at this point do not create new storage data types neither new manipulation operations and are only used to override the set_value operation and the constructors to assure that only objects of the specified type will be used at this point.

2.4.2.2 Mapping temporal relationships

Temporal relationships are modelled in a similar manner, as temporal attributes. Note however that the hierarchy created is larger, as their must be specialisations for every kind of relationship (one to one, one to many, many to many).

The mapping is quite similar in this case too, as each temporal relationship is used to created a sub-class of specified classes, and then in these classes certain operations are overridden in order to have the wanted behaviour. The exact way this happens is shown in the following example, which uses again part of the Product type defined in the Delta application.

```
Interface Product
{
...
      relationship DailyTotal circulation valid event granularity day
            inverse DailyTotal::isProduct;
      relationship ProductionStatistic inDailyProduction valid event
            granularity day inverse ProductionStatistic::isProduct;
...
}
```

The two relationships are mapped to two classes which in turn are used to define the TODM type used for Product. Note of course that this is supplementary to the above definition of the Product type

```
class Historical_event_DailyTotal_one_to_one
            : Historical_event_one_to_one
{
      Historical_event_DailyTotal_one_to_one(Calendar:c, Granule:g)
            : Historical_event_one_to_one(c,g);
      create(DailyTotal to, Timestamp vt)
            { Historical_event_one_to_one::create(to, vt);};
}

class Historical_event_ProductionStatistic_one_to_one
            : Historical_event_one_to_one
{
      Historical_event_ProductionStatistic_one_to_one(Calendar:c,
            Granule:g) : Historical_event_one_to_one(c,g);
      create(ProductionStatistic to, Timestamp vt)
            { Historical_event_one_to_one::create(to, vt);};
}

class Product
{
...
   Historical_event_DailyTotal_one_to_one circulation;
   Historical_event_ProductionStatistic_one_to_one inDailyProduction;

Product()
{
//Initialisation of calendar and granularities
...
circulation = new
 Historical_event_DailyTotal_one_to_one('gregorian','day');
inDailyProduction = new
 Historical_event_ProductionStatistic_one_to_one('gregorian','day');
}
...
}
```

As stated with the temporal attributes the newly created sub-classes are not used neither for storage nor for new manipulation operations but simply to override the create operation so as it will only accept objects of the correct type.

2.4.2.3 Mapping temporal objects

With the temporal objects, there are some more points that must be taken into account during the mapping. Note that there exists a similar to the attributes hierarchy for temporal objects. However, a one-to-one mapping from objects to new subclasses is not appropriate at this point as temporal objects have also states that have to be manipulated along with the object as a whole. This actually means that for each temporal object two classes are created, one to hold the different states (variants) of the object and one to model the object itself. The following example from the GlaxoWellcome application illustrates the method that is going to be used.

```
interface Patient transaction
{
      attribute String patientName;
      attribute String firstName;
      attribute String address;
      relationship Demographic demogrCriteria;

// operation defined on a state
      state operation display();

// operation defined on the temporal object
      operation make_statistics();
...
}
```

This TODL definition will be mapped to the following two TODM definitions:

```
class Patient_State : Object_State
{
      String patientName;
      String firstName;
      String address;
      relationship Demographic demogrCriteria;

      display();
...
}

class Patient : Rollback_Object
{
      Patient();

      make_statistics();

      set_state(Patient_State s)
            { Rollback_Object::set_state(s);};
      correct_state(Rollback_State s)
            { Rollback_Object::correct_state(s);};
...
}
```

Note that the type Patient_State is a sub-type of the Object_State and defines the structure of a state of the temporal object Patient. Patient, in its turn, is a sub-type of the Rollback_Object and allows to check that only Patient_State states will be handled within Patient objects by overriding the set operations of the Rollback_Object.

2.4.2.4 Mapping state relationships

State relationships are defined in previous sections of this document. These are also mapped to TODM through a hierarchy analogous to the one given by TODM for temporal relationships. For each state relationship the target object is mapped to two TODM objects, as happens with temporal objects and then a mapping of the state

relationship to a sub_class of the appropriate state relationship is made and finally the object which contains the state relationship is mapped using the newly created classes.

This can be illustrated by means of an example which is used to illustrate the fact that a patient is treated by a treatment, but each treatment may vary in time, as medical world changes and new data become available that might help the patient. This case could be modelled using state relationships with the following two TODL definitions:

```
interface Treatment valid
{
      attribute String posology;
...
      relationship Set<Components> components;
...
}

interface Patient
{
      attribute String patientName;
...
      state relationship Treatment treatment;
}
```

These TODL definitions may be mapped to the following:

```
class Treatment_state : Object_State
{
      attribute String posology;
...
      relationship Set<components> components;
...
}

class Treatment : Rollback_Object
{
      Treatment();

      set_state(Treatment_State s)
            { Rollback_Object::set_state(s); };
      correct_state(Treatment_State s)
            { Rollback_Object::correct_state(s); );
...
}

class State_Treatment_one_to_one : State_one_to_one
{
      create(Treatment o, Timestamp t)
            { State_one_to_one::create(o,t); };
}

class Patient
{
      String patientName;
      State_Treatment_one_to_one treatment;
...
}
```

2.5 Metadata

Metadata will be used to describe the TODL definitions in the TODBMS, and will also be used during runtime in order as guide to the format and definitions of the different data that are being accessed. Metadata are essential for TOQL to work, as they will provide the types and temporal dimensions of all the interfaces created through TODL.

Metadata information will be created and maintained by TODL, and queried by both TOQL and other applications. This is feasible, as all applications created using the extensions provided by TODM, TODL and TOQL will be able to access the O_2 names that will be used for the storage of the metadata. The names that will be used will be part of what we like to call Temporal Meta Schema.

Note that although some suggestions have been proposed in [ODMG2] for metadata to be defined we choose at the present moment to use this approach only at a minimum level, as the new draft of version 2.0 is about to emerge. However, pointers of how the ODMG version 2.0 approach may be used in TODL are presented in Appendix D.

At the moment we propose ten classes which will be used in order to keep the metadata in the O_2 database. These classes will be part of an O_2 hierarchy and may be exported in any O_2 schema in order to be used by the TODBMS. Seven of these classes refer to the interface definition of the classes with information about the properties and operations. Then the last three classes are used to provide information about the calendars that may be used in any application, that is extra information will be present in these classes only if the user chooses to declare another calendar. All the classes are presented in the next sections, giving their TODL definition[2]. Note that all those classes have a name that begins with *TemporalMeta*. Note also that when these classes will be defined in O_2 they should be sub-classes of the Object class.

2.5.1 TemporalMetaObject

TemporalMetaObject is a class used for to keep most of the information about a temporal or snapshot object. The TODL definition for TemporalMetaObejct is the following:

[2] The definition is actually ODL definition, as no temporal information is kept for these classes, but as we have already decided, all ODL statements are valid in TODL.

```
typedef interfaceTypeEnum Enum{permanent, transient, nospec};
typedef overlapsEnum Enum{na, yes, no};
typedef validTypeEnum Enum{none,state, event};
typedef keyTypeEnum Enum{none, point, period, object};
typedef cardinalityEnum Enum{c1_1, c1_n, cn_1, cm_n};
typedef manyTypeEnum Enum{na, isList, isSet, isBag};
typedef temporalSortedEnum Enum{none, svalid, stransaction, sboth};
typedef in_outEnum Enum{in, out, inout};
typedef granularityMappingEnum Enum{chronon, linear, operation};
typedef Struct classInfo {String className, String classComment};

interface TemporalMetaObject
{
      attribute String interfaceName;
      attribute String interfaceExtent;
      attribute interfaceTypeEnum interfaceType;
      attribute Boolean hasValidTime;
      attribute overlapsEnum overlaps;
      attribute validTypeEnum validType;
      relationship TemporalMetaCalendar calendar;
      relationship TemporalMetaGranularity granularity;
      attribute Boolean hasTransactionTime;
      attribute Set<classInfo> createdClasses;
      relationship Set<TemporalMetaProperty> keys;
      relationship Set<TemporalMetaProperty> properties
            inverse TemporalMetaProperty::belongInterface;
      relationship Set<TemporalMetaOperation> operations
            inverse TemporalMetaOperation::belongInterface;
}
```

Some comments about the properties of the above interface follow. The **interfaceName** attribute is used to keep the name that the user has given at the definition of the specified The **interfaceExtent** attribute is used to keep either the extent name the user proposed for the interface, or the name that is being used if the interface is specified to be permanent with no definition of extent, in which case it will be an arbitrary unique name that will correspond in this case with an O_2 name. The **interfaceType** attribute will actually give us a hint of whether the user has defined the interface instances to be permanent or transient.

The **hasValidTime** attribute which will be true or false, depending on whether the object has valid time characteristics as a whole or not. The **overlaps** attribute which will be yes if the object has valid time characteristics as a whole, with overlapping valid timestamps allowed, no if overlapping valid timestamps are not allowed and na (not available) if no valid time characteristics are defined. The **validType** attribute will have the value none if the object does not have valid time characteristics, state or event depending on the type of the valid time timestamps. The **calendar** relationship is a many-to-one relationship pointing to the calendar in which the valid time will be expressed or to null if no valid time is defined. The **granularity** relationship will be a many-to-one relationship, pointing to the granularity in which the valid time will be expressed or to null if no valid time is defined. Note that both relationships do not have an inverse link, as this is not of importance.

The **hasTransactionTime** attribute will be true or false depending on whether the object has transaction time characteristics as a whole.

The **createdClasses** attribute will hold the name and a comment for each class actually created in TODM to hold the information about this interface.

The **keys** relationship is a one-to-many relationship that holds those properties (attributes and relationships) of the interface that have been defined as part of the key of the interface. Note that an inverse path doesn't exist as it is available from the

belongInterface relationship defined for each TemporalMetaProperty (see below). The **properties** relationship is a superset of the keys relationship and holds all the properties (attributes and relationships) defined in the interface. This relationship is also one-to-many. The **operations** relationship is also an one-to-many relationship used to keep a pointer to all the operations defined in the specified interface.

Note also at this point the definition of some enumeration types, that are going to be used for the rest of the metadata definition.

2.5.2 TemporalMetaProperty

The *TemporalMetaProperty* class is used to keep information common for any kind of properties defined in an interface. Such information is the name of the property and the temporal characteristics of the attributes and relationships defined. The TODL definition for TemporalMetaProperty follows:

```
interface TemporalMetaProperty
{
      attribute String propertyName;
      attribute Boolean hasValidTime;
      attribute overlapsEnum overlaps;
      attribute validTypeEnum validType;
      attribute keyTypeEnum keyType;
      relationship TemporalMetaCalendar calendar;
      relationship TemporalMetaGranule granularity;
      attribute Boolean hasTransactionTime;
      relationship TemporalMetaObject belongInterface
            inverse TemporalMetaObject::properties;
}
```

The **hasValidTime**, **overlaps, validType**, **calendarName**, **granularity** and **hasTransactionTime** attributes and relationships are analogous to the ones defined for the TemporalMetaObject, but their scope is the property, instead of the whole object. The **keyType** attribute declares whether the property is part of the key of the interface pointed at by the **belongInterface** relationship. Note that as a property exists only through an object it cannot belong to more than one interfaces at the same time. Two properties that have the same name and are part of the definition of two different interfaces, are not the same property.

2.5.3 TemporalMetaAttribute

In the *TemporalMetaAttribute* class - which inherits from the TemporalMetaProperty class - we keep information specific to the attributes defined within an interface. Much of the necessary information is kept through the parent class of *TemporalMetaProperty*. At this level we only keep information such as whether the attribute is read-only or not (through attribute **isReadOnly**), whether it is used for sorting a relationship (using the many-to-many relationship **isSorter**) and of course the type of the attribute (through the attribute **attributeType**). The TODL definition is the following:

```
interface TemporalMetaAttribute: TemporalMetaProperty
{
      attribute Boolean isReadOnly;
      relationship Set<TemporalMetaRelationship> isSorter
            inverse TemporalMetaRelationship::isSorted;
      attribute String attributeType;
}
```

2.5.4 TemporalMetaRelationship

Analogous to the *TemporalMetaAttribute*, *TemporalMetaRelationship* is used to keep information about relationships defined within a specified interface. Such information includes the cardinality of the relationship (through the **cardinality** attribute), a pointer to the inverse path of the relationship, if one exists, through the one-to-one **inversePath** relationship, the object being at the other end of the path, using the one-to-one relationship **traversalPath**, the attributes of the pointed object which are used as sorters for this relationship, through the many-to-many relationship **isSorted**. In the case the relationship which we are describing is pointin to more than one objects, we use an attribute, **manyType**, which gives us more information as to the way the collection is implemented (through set, list or bag[3]). Note that relationships may be sorted over valid or transaction time or both, if they have temporal characteristics, this is indicated by the attribute **temporalSorted** which shows whether such an order exists, as well as which of the two time dimensions is first. Finally an indication whether the relationship is a state relationship or not is provided, through the **isState** attribute. The TODL definition of the *TemporalMetaRelationship* follows:

```
interface TemporalMetaRelationship: TemporalMetaProperty
{
      attribute Boolean isState;
      attribute cardinalityEnum cardinality;
      attribute manyTypeEnum manyType;
      relationship TemporalMetaObject traversalPath;
      relationship TemporalMetaRelationship inversePath;
      relationship Set<TemporalMetaAttribute> isSorted
            inverse TemporalMetaAttribute::isSorter;
      attribute temporalSortedEnum temporalSorted;
}
```

2.5.5 TemporalMetaOperation

The *TemporalMetaOperation* is used to keep information about the signatures of the operations defned for each interface. Such information includes the name of the operation, its parameters, what the operation returns and if it has any temporal information, any exceptions the operation might be raising, whether the operation is defined on the state of an object or not, the interface in which the operation is defined and the context in which the operation is defined. The TODL declaration of the *TemporalMetaOperation* follows:

```
interface TemporalMetaOperation
{
      attribute String operationName;
      relationship TemporalMetaObject belongInterface
            inverse TemporalMetaObject::operations;
      attribute String resultType;
      relationship TemporalMetaType returnsTemporal;
      relationship Set<TemporalMetaParameters> parameters;
      attribute Set<String> exceptionsRaised;
      attribute Boolean isState;
      attribute String inContext;
}
```

[3] Note that in ODMG 2.0 draft version the types array and dictionary may also be used as collection types.

2.5.6 TemporalMetaType

The *TemporalMetaType* is used to keep information about the temporal types that might be used within an operation. This includes both the type of the result and the types of the parameters of operations. The information needed is mainly information about which the basic type is and whether it has valid or transaction time characteristics.The TODL definition of *TemporalMetaType* is the following:

```
interface TemporalMetaType
{
      attribute String basicTypeName;
      attribute Boolean hasValidTime;
      attribute overlapsEnum overlaps;
      attribute validTypeEnum validType;
      relationship TemporalMetaCalendar calendar;
      relationship TemporalMetaGranularity granularity;
      attribute Boolean hasTransactionTime;
}
```

2.5.7 TemporalMetaParameter

The *TemporalMetaParameter* inherits from *TemporalMetaType* and has extra information that may be used for the parameters of an operation, that is whether the parameter is in, out or inout.

```
interface TemporalMetaParameter: TemporalMetaType
{
      attribute in_outEnum in_outSpec;
}
```

2.5.8 TemporalMetaCalendar

In *TemporalMetaCalendar* we keep information about the calendar used in the database. Note that by default only the gregorian calendar is defined. Whenever the user defines through TODL another calendar, then the information about it will be stored through this interface. The TODL definition about the *TemporalMetaCalendar* is the following:

```
interface TemporalMetaCalendar
{
      attribute String calendarName;
      attribute Instant origin;
      relationship Set<TemporalMetaGranule> granules
            inverse TemporalMetaGranule::calendar;
      relationship Set<TemporalMetaExternalFormats> externalFormats
            inverse TemporalMetaExternalFormat::calendar;
}
```

The attribute **calendarName** is used to keep the name of the calendar defined, the attribute **origin** is used to store the Instant that is considered to be the origin of the calendar.Then two relationships one-to-many relationships are presented. Namely: **granules** which points to the granules defined in the calendar, and **externalFormats** which points to the external formats that might have been defined for the calendar.

2.5.9 TemporalMetaGranule

Using *TemporalMetaGranule* we keep information about the granularities defined within a calendar. Note that no granule can exist outside a calendar. So the TODL definition of *TemporalMetaGranule* is the following:

```
interface TemporalMetaGranule
{
	attribute String granuleName;
	relationship TemporalMetaCalendar calendar
		inverse TemporalMetaCalendar::granules;
	relationship TemporalMetaGranule coarser
		inverse TemporalMetaGranule::finer;
	relationship TemporalMetaGranule finer
		inverse TemporalMetaGranule::coarser;
	attribute granuleMappingEnum kindOfMapping
	attribute String operationToFiner;
	attribute String operationFromFiner;
	attribute Short mapToFiner;
}
```

The **granuleName** attribute holds the name of the granule and the **calendar** relationship is pointing the calendar in which the granularity is defined. Note that two granularities with the same name which point to two different calendars are actually two different granularities. The relationships **coarser** and **finer** point to the immediate coarser and the immediate finer granularities, and of course the one is the inverse relationship of the other. The attribute **kindOfMapping** is used to present the way the mapping to the finer granularity is made: the value *chronon* actually means that the granularity is the finest granularity in the calendar, the value *linear* declares that a linear function exists, and the value *operation* declares that an operation is used to make the mapping. Depending on the value of the **kindOfMapping** attribute the next three attributes may or may not have a value:

1. If **kindOfMapping** = *chronon* then the following attributes have meaningless values (they should have as values the empty string the first two and a negative number the third, but there should be no access to them in such a case)
2. if **kindOfMapping =** *linear* then only the **mapToFiner** attribute has a meaningful value and it should be the integer n which is the number of granules of the immediate finer granularity that form one granule of the specified granularity. Attribute **operationToFiner** and **operationFromFiner** should have as their values an empty string.
3. if **kindOfMapping** = *operation* then the attributes **operationToFiner** and **operationFromFiner** have meaningful values which should be the names of the operations used to map a granule of the specified granularity to the immediate finer granularity and a granule of the immediate finer granularity to a granule of the specified granularity, respectively. Attribute **mapToFiner** should have a negative value.

2.5.10 TemporalMetaExternalFormat

Using *TemporalMetaExternalFormat* we may keep information about the external formats a calendar has. The TODL definition is the following:

```
interface TemporalMetaExternalFormat
{
	attribute String externalFormatName;
	relationship TemporalMetaCalendar calendar
		inverse TemporalMetaCalendar::externalFormats;
	attribute String stringPresentation;
	attribute String inputResult;
	attribute String inputParameter;
	attribute String outputResult;
	attribute String outputParameter;
}
```

The information stored is the name of the external format, using the attribute **externalFormatName**, the calendar in which the external format belongs, using the **calendar** relationship, the presentation of the external format using the **stringPresentation** attribute, and the results and parameters of the input and output operations which are provided by attributes **inputResult, outputResult, inputParameter** and **outputParameter**, respectively.

3. ODL 2.0 Compliance

As presented in the draft version of ODMG v. 2.0 there are not major changes that might affect the design of TODL. In what follows we are going to present how the changes in ODL 2.0 might affect TODL specifications.

One change that is easily identifiable is that type names are no longer capitalised. Of course such a change can be very easily adopted by TODL, with the provision that definitions with capitalised data types will also be accepted - issuing a warning message may be - in order to be backwards compatible with older specifications.

Another change is that the **inverse** statement is obligatory in ODL 2.0.This might cause some problems, as we have already defined that for certain temporal and state relationships, the inverse relationship cannot be defined. The solution to this problem may be that we allow for the inverse statement to be obligatory in TODL for non-temporal relationships, but for temporal and state relationships the constraints already posed by this version of TODL will still be valid in the next version of TODL.

In ODL 1.1, attribute were allowed to contain only literal values. However in ODL 2.0 attributes are allowed to have object types for their types. This can also be easily accommodated in the next version of TODL, provided of course that TODM will be able to accept such attributes. Note however, that for the implementation in the framework of TOOBIS project, there will be no problem, as O_2 treats relationships and attributes in the same way, so attributes can have an object type.

Some more changes are made in ODM, that eventually might affect TODL. Such changes are the incorporation of new data types, like the Dictionary collection type that may be incorporated subsequently in ODL, and then in TODL. Also in the object data model of ODMG v.2.0, a proposal for metadata is supplied. In order to define the TODL metadata schema, we have used ideas from this draft and of course if it is fully accepted, then changes should be made in the corresponding meta classes in order for the two schemas to be fully compatible.

4. Conclusions

In this report we gave the specification and design of a temporal extension to ODL v.1.1. The extension is consistent with ODL, in that it preserves its syntax and semantics. The design of the TODL processor allows for portability over any ODMG compliant DBMS.

Appendix A: TODL BNF SYNTAX

In this appendix you can find the complete BNF syntax of TODL. It is based on ODL's BNF

```
<specification>          ::=  <definition> [<specification>]      1
<definition>             ::=  <type_dcl> ; | <const_dcl> ; |      2
                              <except_dcl> ; | <interface> ; |
                              <module> ; |
                              <calendar_definition> ;
<module>                 ::=  module <identifier>                 3
                              {<specification>}
<interface>              ::=  <interface_dcl> | <forward_dcl>     4
<interface_dcl>          ::=  <interface_header>                   5
                              [: <persistence_dcl>]
                              [<temporal_dcl>]
                              {[<interface_body>]}
<persistence_dcl>        ::=  persistent | transient              6
<temporal_dcl>           ::=  <valid_dcl> | <transaction_dcl> |   7
                              <valid_dcl> <transaction_dcl> |
                              <transaction_dcl> <valid_dcl>
<transaction_dcl>        ::=  transaction                         8
<valid_dcl>              ::=  valid [<state_event_dcl>]           9
                              [overlaps]
                              [granularity <granularity>]
                              [calendar <calendar>]
<state_event_dcl>        ::=  state | event                       10
<granularity>            ::=  year | month | day | hour |         11
                              minute | second | <identifier>
<calendar>               ::=  gregorian | <identifier>            12
<forward_dcl>            ::=  interface <identifier>              13
<interface_header>       ::=  interface <identifier>              14
                              [: <inheritance_spec>]
                              [<type_property_list>]
<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]                  19
                              <property_name> |
                              ( <property_list> )
<temporal_key_dcl>       ::=  point | period | object             20
<property_list>          ::=  [temporal_key_dcl]                  21
                              <property_name>
                              [,<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>                       25
                              [,<inheritance_spec>]
<scoped_name>            ::=  [[<scoped_name>] ::] <identifier>   26
<const_dcl>              ::=  const <const_type> <identifier>     27
                              = <const_expr>
<const_type>             ::=  <integer_type> | <char_type> |      28
                              <boolean_type> |
                              <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> |                        39
                               <string_literal> |
                               <character_literal> |
                               <floating_pt_literal> |
                               <boolean_literal> |
                               <interval_literal> |
                               <instant_literal> |
                               <period_literal>
<boolean_literal>         ::=  TRUE | FALSE                              40
<instant_literal>         ::=  "<absolute_instant>" |                     41
                               "<relative_instant>"
<period_literal>          ::=  "[<instant_literal>,                       42
                               <instant_literal>)"
<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>                             47
                               [<instant_calendar>]
<instant_calendar>        ::   identifier | gregorian                     48
<interval_calendar>       ::=  identifier | gregorian                     49
<positive_int_const>      ::=  <const_expr>                               50
<type_dcl>                ::=  typedef <type_declarator> |                51
                               <struct_type> | <union_type> |
                               <enum_type>
<type_declarator>         ::=  <type_spec> <declarators>                  52
<type_spec>               ::=  <simple_type_spec> |                       53
                               <constr_type_spec>
<simple_type_spec>        ::=  <base_type_spec> |                         54
                               <template_type_spec> |
                               <scoped_name>
<base_type_spec>          ::=  <floating_pt_type> |                       55
                               <integer_type> | <char_type> |
                               <boolean_type> | <octet_type> |
                               <interval_type> | <instant_type>
                               | <period_type> | <any_type>
<template_type_spec>      ::=  <array_type> | <string_type>               56
<constr_type_spec>        ::=  <struct_type> | <union_type> |             57
                               <enum_type>
<declarators>             ::=  <declarator> [, <declarators>]             58
<declarator>              ::=  <simple_declarator> |                      59
```

```
                                <complex_declarator>
<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> |                   64
                                <signed_short_int>
<signed_long_int>         ::=   Long                                  65
<signed_short_int>        ::=   Short                                 66
<unsigned_int>            ::=   <unsigned_long_int> |                 67
                                <unsigned_short_int>
<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                              73
                                [granularity <granularity>]
                                [calendar <calendar>]
<instant_type>            ::=   Instant                               74
                                [granularity <granularity>]
                                [calendar <calendar>]
                                [absolute | relative]
<period_type>             ::=   Period                                75
                                [granularity <granularity>]
                                [calendar <calendar>]
                                [absolute | relative]
<any_type>                ::=   Any                                   76
<struct_type>             ::=   Struct <identifier>                   77
                                {<member_list>}
<member_list>             ::=   <member> [<member_list>]              78
<member>                  ::=   <type_spec> <declarators> ;           79
<union_type>              ::=   union <identifier> switch             80
                                (<switch_type_spec>)
                                {<switch_body>}
<switch_type_spec>        ::=   <integer_type> | <char_type> |        81
                                <boolean_type> | <enum_type> |
                                <scoped_name>
<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>                     87
                                {<enumerator_list>}
<enumerator_list>         ::=   <enumerator> [,<enumerator_list>]     88
<enumerator>              ::=   <identifier>                          89
<array_type>              ::=   <array_spec> < <simple_type_spec>     90
                                [,<positive_int_const>] >
<array_spec>              ::=   Array | Sequence                      91
<string_type>             ::=   String [< <positive_int_const> >]     92
<array_declarator>        ::=   <identifier> <array_size_list>        93
<array_size_list>         ::=   <fixed_array_size>                    94
                                [<array_size_list>]
<fixed_array_size>        ::=   [ <positive_int_const> ]              95
<attr_dcl>                ::=   [readonly] attribute                  96
                                <domain_type> <attribute_name>
                                [fixed_array_size] [temporal_dcl]
<domain_type>             ::=   <simple_type_spec> |                  97
                                <struct_type> | <enum_type> |
                                <attr_collection_specifier>
                                <literal> |
                                <attr_collection_specifier>
                                <identifier>
```

```
<attr_collection_-        ::=  Set | List | Bag | Array                98
specifier>
<rel_dcl>                 ::=  [state] relationship                    99
                               <target_of_path> <identifier>
                               [<temporal_dcl>]
                               [inverse
                               <inverse_traversal_path>]
                               {[<order_by>]}
<order_by>                ::=  order_by <attribute_list> |            100
                               order_by [valid] [transaction]
<target_of_path>          ::=  <identifier> |                         101
                               <rel_collection_type>
                               <<identifier>>
<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>                 105
                               {[<member_list>]}
<op_dcl>                  ::=  [oneway] [state] <op_type_spec>        106
                               <identifier> <param_dcls>
                               [<raises_expr>] [<context_expr>]
                               [temporal_dcl]
<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>                      110
                               <simple_type_spec> <declarator>
                               [temporal_dcl]
<param_attribute>         ::=  in | out | inout                       111
<raises_expr>             ::=  raises ( <scoped_name_list> )          112
<scoped_name_list>        ::=  <scoped_name>                          113
                               [,<scoped_name_list>]
<context_expr>            ::=  context ( <string_literal_list> )      114
<string_literal_list>     ::=  <string_literal>                       115
                               [,<string_literal_list>]
<calendar_definition>     ::=  calendar <calendar_name>               116
                               {origin <instant_literal>;
                               granules <granules_list>;
                               [external
                               <external_format_list>;]}
<calendar_name>           ::=  <identifier>                           117
<granules_list>           ::=  <granule_name> :                       118
                               {<mappings>}
                               [, <granules_list>]
<granule_name>            ::=  <identifier>                           119
<mappings>                ::=  chronon |                              120
                               <integer_literal> <granule_name>
                               |
                               <mapping_function_name>,
                               <mapping_function_name>
<mapping_function_name>   ::=  <identifier> ()                        121
<external_format_list>    ::=  <format_name> =                        122
                               <string_literal>
                               {<mapping_definition>}
                               [, <external_format_list>]
<format_name>             ::=  <identifier>                           123
<mapping_definition>      ::=  <return_type>                          124
                               input(<format_name>),
                               <format_name>
                               output(<return_type>)
```

Appendix B: Language completeness with respect to the user requirements

In this Appendix we will demonstrate how the objects that have been identified by the user requirements of Delta and GlaxoWellcome can be created using TODL. In the tables the following abbreviations are used:

V Valid Time
T Transaction Time
B Bitemporal
E Event (for valid time and bitemporal only)
S State (for valid time and bitemporal only)
Y Year (granularity)
D Day (granularity)
H Hour (granularity)
M Minute (granularity)
S Second (granularity)

1. Delta Application

The Delta application has a number of classes associated with it. These classes have been identified through the User Requirements part and are presented in what follows in a tabulated manner. The exact TODL statements that may be used to create the objects follow. First we introduce the different type definitions that are used in the rest of the Delta Application.

1.1 Type definitions

```
typedef shopTypeEnum Enum{local, sm, sm_group};
typedef customerTypeEnum Enum{old, new, lost_competition};
typedef Struct productType {Product p, Short Quantity};
typedef weatherTypeEnum Enum{very_hot, hot, rainy, snowy,
                             cold, very_cold};
typedef categoryEnum Enum{daily, expired, sm};
typedef areaTypeEnum Enum{port, airport, railway, metro,
                             central_area, remote_area};
typedef effectEnum Enum{increase, decrease};
```

1.2 Class Product

```
interface   Product
(extent Products
      key productName)
{
// Attributes
      attribute Interval duration valid state granularity day;
      attribute String productName;
      attribute Float price valid state granularity day;

// Relationships
      relationship DailyTotal circulation valid event granularity day
            inverse DailyTotal::isProduct;
      relationship ProductionStatistic inDailyProduction
            valid event granularity day
            inverse ProductionStatistic::isProduct;

// Operations
      DailyProductOrder inDailyOrder();
      AdditionalOrder inAdditionalOrder();
      Set<CustomerStock> inCustomerStock();
      Set<CustomerOrder> inCustomerOrder();
      Set<CustomerReturn> inCustomerReturn();
      Set<CustomerDelivery> inCustomerDelivery();
      DeliveryPackage inDeliveryPackage();
      Target inTarget();
}
```

1.3 Class Customer

```
interface Customer
(extent Customers
      key customerName)
{
// Attributes
      attribute Boolean deltaClubMember valid state granularity day;
      attribute Instant granularity day fYPurchaseDate;
      attribute Instant granularity day installedTentDate;
      attribute Instant granularity day lYPurchaseDate;
      attribute Instant granularity day lYTransactionDate;
      attribute String customerName;
      attribute shopTypeEnum shopType;
      attribute customerTypeEnum customerType valid granularity day;

// Relationships
      relationship CustomerStock maintains valid event
            granularity minute inverse CustomerStock::maintainedBy;
      relationship CustomerReturn returns valid event
            granularity minute inverse CustomerReturn::returnedBy;
      relationship CustomerDelivery receives valid event
            granularity minute inverse CustomerDelivery::receivedBy;
      relationship CustomerOrder orders valid event
            granularity minute inverse CustomerOrder::orderedBy;
      relationship AdditionalOrder ordersAdditional valid event
            granularity minute inverse AdditionalOrder::orderedBy;
      relationship DeliveryPackage receivesAdditional valid event
            granularity minute
            inverse DeliveryPackage::deliveredAdditional;
      relationship Distributor assigned valid state granularity day
            inverse Distributor::serves;
      relationship AddressLocation situated valid state
            granularity day inverse AddressLocation::situates;
      relationship Distributor visited valid event
            granularity minute inverse Distributor::makesVisit;

// Operations
}
```

1.4 Class Distributor

```
interface Distributor
(extent     Distributors)
{
// Attributes
      attribute String loadingPriviledge valid granularity day;
      attribute String distributorName;

// Relationships
      relationship Set<Customer> serves valid state
            granularity day inverse Customer::assigned;
      relationship Set<Customer> makesVisit valid event
            granularity minute inverse Customer::visited;
      relationship Set<DeliveryPackage> delivers valid event
            granularity minute inverse DeliveryPackage::deliveredBy;
      relationship DailyProductOrder orders valid event
            granularity minute inverse DailyProductOrder::orderedBy;
      relationship Target hasTarget valid event granularity day
            inverse Target::isDistributor;

// Operations
}
```

1.5 Class Batch

```
interface Batch
(extent Batches
      key batchID)
{
// Attributes
      attribute Long batchID;
      attribute Set<productType> products;
      attribute Float discount;
      attribute Instant timestamp;

// Relationships

// Operations
      Set<Product> isProduct();
}
```

1.6 Class CustomerDelivery

```
interface CustomerDelivery
(extent CustomerDeliveries)
{
// Attributes

// Relationships
      relationship Batch deliveryData valid event granularity minute;
      relationship Customer receivedBy valid event granularity minute
            inverse Customer::receives;

// Operations
}
```

1.7 Class AdditionalOrder

```
interface AdditionalOrder
(extent AdditionalOrders)
{
// Attributes

// Relationships
      relationship Batch additionalOrderData valid event
            granularity minute;
      relationship DeliveryPackage matchedBy valid event
            granularity minute inverse DeliveryPackage::matches;
      relationship Customer orderedBy valid event granularity minute
            inverse Customer::ordersAdditional;

// Operations
}
```

1.8 Class CustomerReturn

```
interface CustomerReturn
(extent CustomerReturns)
{
// Attributes
      attribute categoryEnum category;

// Relationships
      relationship Batch returnData valid event granularity minute;
      relationship Customer returnedBy valid event granularity minute
            inverse Customer::returns;

// Operations
}
```

1.9 Class CustomerStock

```
interface CustomerStock
(extent CustomerStocks)
{
// Attributes

// Relationships
      relationship Batch stockData valid event granularity minute;
      relationship Customer maintainedBy valid event
            granularity minute inverse Customer::maintains;

// Operations
}
```

1.10 Class CustomerOrder

```
interface CustomerOrder
(extent CustomerOrders)
{
// Attributes

// Relationships
      relationship Batch orderData valid event granularity minute;
      relationship Customer orderedBy valid event granularity minute
            inverse Customer::orders;

// Operations
}
```

1.11 Class DeliveryPackage

```
interface DeliveryPackage
(extent DeliveryPackages)
{
// Attributes

// Relationships
      relationship Batch packageData valid event granularity minute;
      relationship AdditionalOrder matches valid event
            granularity minute inverse AdditionalOrder::matchedBy;
      relationship Customer deliveredAdditinal valid event
            granularity minute inverse Customer::receivesAdditional;
      relationship Distributor deliveredBy valid event
            granularity minute inverse Distributor::delivers;

// Operations
}
```

1.12 Class DailyProductOrder

```
interface DailyProductOrder
(extent DailyProductOrders)
{
// Attributes

// Relationships
      relationship Batch dailyProduct valid event granularity minute;
      relationship Distributor orderedBy valid event
            granularity minute inverse Distributor::orders;

// Operations
}
```

1.13 Class Target

```
interface Target
(extent Targets)
{
// Attributes

// Relationships
      relationship Batch targetData valid event granularity day;
      relationship Distributor isDistributor valid event
            granularity day inverse Distributor::hasTarget;

// Operations
}
```

1.14 Class DailyTotal

```
interface DailyTotal
{extent DailyTotals)
{
// Attributes
      attribute Float loadTotalAmt;
      attribute Unsigned Long quantityLoaded;
      attribute Unsigned Long quantityReturned;
      attribute Float returnTotalAmt;
      attribute Instant returnedAt;

// Relationships
      relationship Product isProduct valid event granularity day
            inverse Product::circulation;

// Operations
}
```

1.15 Class DailyCirculation

```
interface DailyCirculation
(extent DailyCirculations)
{
// Attributes
      attribute Float loadTotalAmt;
      attribute Float returnTotalAmt;
      attribute Instant granularity day circulatedAt;

// Relationships
      relationship Distributor circulatedBy valid event
            granularity day inverse Distributor::circulates;
      relationship Set<DailyTotal> totalOfProducts;

// Operations
}
```

1.16 Class WeatherCondition

```
interface WeatherCondition
(extent WeatherConditions)
{
// Attributes
      attribute Float humidity;
      attribute Float temperature;
      attribute weatherTypeEnum weatherType;
      attribute Instant granularity day dataDate

// Relationships

// Operations
}
```

1.17 Class DailyProduction

```
interface DailyProduction
(extent DailyProductions)
{
// Attributes
      attribute Instant granularity day dateOfProduction;

// Relationships
      relationship DailyProduction nextYear
            inverse DailyProduction::prevYear;
      relationship DailyProduction prevYear
            inverse DailyProduction::nextYear;
      relationship Set<ProductionStatistic> StatOfProducts;

// Operations
}
```

1.18 Class ProductionStatistic

```
interface ProductionStatistic
(extent ProductionStatistics)
{
// Attributes
      attribute Unsigned Long actualQty;
      attribute Unsigned Long plannedQty;
      attribute Unsigned Long soldQty;
      attribute Instant granularity day dayOfProduction;

// Relationships
      relationship Product isProduct valid event granularity day
            inverse Product::inDailyProduction;

// Operations
}
```

1.19 Class AddressLocation

```
interface AddressLocation
(extent AddressLocations)
{
// Attributes
      attribute Float hPosition;
      attribute Float vPosition;
      attribute Short page;

// Relationships
      relationship Set<Customer> situates valid state granularity day
            inverse Customer::situated;
      relationship Area inArea inverse Area::consistsOf;

// Operations
}
```

1.20 Class Area

```
interface Area
{
// Attributes
      attribute String description;
      attribute areaTypeEnum areaType;

// Relationships
      relationship Set<AddressLocation> consistsOf
            inverse AddressLocation::inArea;
      relationship Set<EventLocation> event
            inverse EventLocation::takesPlace;

// Operations
}
```

1.21 Class Event

```
interface Event
(extent Events)
{
// Attributes
      attribute String category;
      attribute effectEnum effect;
      attribute String eventName;
      attribute Period happens valid state granularity year;

// Relationships
      relationship Area takesPlace inverse Area::event;

// Operations
}
```

2. GlaxoWellcome Application

In the following sections the reader can find the definition of the classes used for the GlaxoWellcome application, as well as the TODL statements that can be used to create the classes. Also the type definitions used in the GlaxoWellcome Application are presented here.

2.1 Type definitions

```
typedef listOfInstants List<Instant>;
typedef Struct questionAndInfo {Boolean has, String info};
typedef Struct modifiedData {Boolean modified, Instant atTime};
typedef Struct questionAndNumber {Boolean had, Short howMany};
```

2.2 Class Patient

```
interface Patient
( extent Patients
  keys (patientName, firstName))
transaction
{
      attribute String patientName;
      attribute String firstName;
      attribute String address;

      relationship Demographic demogrCriteria;
      relationship SexualMaturity sexMaturity;
      relationship LifeConditions lifeConditions;
      relationship IllnessHistory illnessHistory;
      relationship PathNonAllergic nonAllergic;
      relationship PathologyAllergic allergic;
      relationship ObservationNotebook observes
            inverse ObservationNotebook::ofPatient;
      relationship IndividualNotebook fillsOut valid event
            granularity hour inverse IndividualNotebook::ofPatient
}
```

2.3 Class ObservationNotebook

```
interface ObservationNotebook
( extent ObservationNotebooks
  keys (protocolCode, investigatorName, investigationCenter,
       treatmentNumber, ofPatient))
transaction
{
      attribute String protocolCode;
      attribute String investigationCenter;
      attribute String investigatorName;
      attribute String treatmentNumber;

      relationship TheorDiagram theoretical;
      relationship ObservationStatement obsPeriodStat valid state
            granularity day;
      relationship PhysExam physExam valid event granularity day;
      relationship FunctRespTest functRespTest valid event
            granularity day;
      relationship VerificationEligible verifEligib;
      relationship Stat1Week stat1Week valid event granularity day;
      relationship VisitStatement visitStat valid event
            granularity day;
      relationship Stat4Week stat4Week;
      relationship MotivesAdditional motivesAdditional valid event
            granularity day;
      relationhsip TrialEnd trialEnd;
      relationship TrialTreatment trialTreatm;
      relationship ConcomitantTreat concomitant valid event
            granularity hour;
      relationship UndesirableEvent undesirableEvent valid event
            granularity minute;
      relationship Patient ofPatient inverse Patient::observes;
}
```

2.4 Class IndividualNotebook

```
interface IndividualNotebook
( extent IndividualNotebooks
  key ofPatient)
transaction
{
      attribute String responsibleInvestigator;
      attribute String type;
      attribute Boolean waking;
      attribute Long ventoline;
      attribute Float expiratoryDebit;
      attribute Long cough;
      attribute Long breathDifficulties;
      attribute Long whistleBreathing;

      relationship Patient ofPatient valid event granularity hour
            inverse Patient::fillsOut;
}
```

2.5 Class TheorDiagram

```
interface TheorDiagram
transaction
{
      attribute Instant timestamp;
      attribute Period obsPeriod
      attribute Period treatmentPeriod;
      attribute Period cisWritingPeriod;
      attribute listOfInstants visits;
      attribute listOfInstants selectionCriteria;
      attribute listOfInstants clinicalExams;
      attribute listOfInstants respiratoryTest;
}
```

2.6 Class Demographic

```
interface Demographic
transaction
{
      attribute Instant timestamp;
      attribute Instant granularity day birthday;
      attribute Char sex;
      attribute Char race;
      attribute Short height;
      attribute Float weight;
}
```

2.7 Class LifeConditions

```
interface LifeConditions
transaction
{
      attribute Instant timestamp;
      attribute Long accomodation;
      attribute Long lifeConditions;
      attribute Long school;
}
```

2.8 Class IllnessHistory

```
interface IllnessHistory
transaction
{
      attribute Instant timestamp;
      attribute Interval length;
      attribute Char allergic;
      attribute Boolean seasonalAllergy;
      attribute Boolean desensitization;
      attribute Short hospitalizationNo;
      attribute Interval hospitalizationDur;
      attribute Short corticoodesNo;
      attribute String corticoodeSpec;
      attribute String corticoodePoso;
      attribute Instant corticoodeBegin;
      attribute Boolean theophyline;
      attribute Boolean b2inhaledLong;
      attribute Boolean b2inhaledShort;
      attribute Boolean b2oralLong;
      attribute Boolean b2oralShort;
      attribute Boolean atropinique;
      attribute Boolean cromone;
      attribute Boolean antihistaminique;
      attribute Boolean others;
}
```

2.9 Class ObservationStatement

```
interface ObservationStatement
transaction
{
      attribute Boolean apptRandom;
      attribute Boolean treatmModif;
}
```

2.10 Class SexualMaturity

```
interface SexualMaturity
transaction
{
      attribute Instant timestamp;
      attribute Short pubic;
      attribute Short breast;
      attribute Short penis;
      attribute Short testicles;
}
```

2.11 Class PathNonAllergic

```
interface PathNonAllergic
transaction
{
      attribute Instant timestamp;
      attribute questionAndInfo respiratory;
      attribute questionAndInfo orl;
      attribute questionAndInfo ocular;
      attribute questionAndInfo cardiovascular;
      attribute questionAndInfo gastrointestinal;
      attribute questionAndInfo heparPancreas;
      attribute questionAndInfo urologic;
      attribute questionAndInfo genital;
      attribute questionAndInfo neurologic;
      attribute questionAndInfo bloodLymphatic;
      attribute questionAndInfo endocrine;
      attribute questionAndInfo locomotor;
      attribute questionAndInfo cutaneous;
      attribute questionAndInfo psychiatric;
      attribute questionAndInfo noLocalization;
}
```

2.12 Class PathologyAllergic

```
interface PathologyAllergic
transaction
{
      attribute Instant timestamp;
      attribute Boolean pathNoAsthma;
      attribute Boolean rhiniteSeason;
      attribute Boolean rhiniteAnnual;
      attribute Boolean cutaneousAllergy;
      attribute Boolean alimentaryAllergy;
      attribute Boolean otherAllergies;
      attribute Boolean followOtherTreat;
}
```

2.13 Class FunctRespTest

```
interface FunctRespTest
transaction
{
      attribute Float vemsObs;
      attribute Float vemsTheory;
      attribute Float cvf;
      attribute Float dem25_75;
      attribute Float dem50;
      attribute Float dem75;
      attribute Float dep;
      attribute Float revVEMSObs;
      attribute Float revVEMSTheory;
}
```

2.14 Class PhysExam

```
interface PhysExam
transaction
{
      attribute questionAndInfo planeresSkin;
      attribute questionAndInfo ganglionLympa;
      attribute questionAndInfo eyes;
      attribute questionAndInfo orl;
      attribute questionAndInfo breast;
      attribute questionAndInfo respiratory;
      attribute questionAndInfo cardiovascular;
      attribute questionAndInfo abdomen;
      attribute questionAndInfo urogenital;
      attribute questionAndInfo pelvis;
      attribute questionAndInfo rectal;
      attribute questionAndInfo locomotor;
      attribute questionAndInfo neurologic;
      attribute questionAndInfo physic;
}
```

2.15 Class VerificationEligible

```
interface VerificationEligible
transaction
{
      attribute Instant timestamp;
      attribute Boolean age8_16;
      attribute Boolean seriousAsthma;
      attribute Boolean corticoodeTreat;
      attribute Short noOfTreatments;
      attribute String symptoms;
      attribute Float vems;
      attribute Boolean parentConsent;
      attribute Boolean corticotherapy;
      attribute Boolean treatmModif;
      attribute Boolean contraindication;
      attribute Boolean seriousConcPath;
      attribute Boolean desensitization;
      attribute Boolean inapte;
      attribute Boolean pregnant;
      attribute Boolean contraception;
}
```

2.16 Class Stat1Week

```
interface Stat1Week
transaction
{
      attribute Boolean prematureEnd;
      attribute String treatmentPrescr;
      attribute modifiedData treatmModDate;
      attribute Boolean followingProtocol;
      attribute String stopReason;
      attribute String otherTreatments;
      attribute Boolean undesirableEvents
}
```

2.17 Class Stat4Week

```
interface Stat4Week
transaction
{
	attribute Instant timestamp;
	attribute questionAndNumber gpVisit;
	attribute questionAndNumber specialistVisit;
	attribute Boolean hospitalization;
	attribute Interval hospitalizationDur;
	attribute Short hospitalizationType;
	attribute String hospitalizationMotiv;
	attribute questionAndInfo paramedicalActions;
	attribute Boolean responsibleAdultStop;
	attribute Short noStoppageDays;
	attribute Boolean bloodStatement;
	attribute String bloodStatementType;
	attribute Short noBloodStatements;
	attribute Boolean radiologicExam;
	attribute String radiologicExamType;
	attribute Short noRadiologicExam;
	attribute Boolean efrExam;
	attribute Short noEFRExams;
	attribute Boolean otherExams;
	attribute String otherExamsType;
	attribute Short noOtherExams;
}
```

2.18 Class VisitStatement

```
interface VisitStatement
transaction
{
	attribute Boolean asthmaControl;
	attribute Boolean aggravationCriteria;
}
```

2.19 Class MotivesAdditional

```
interface MotivesAdditional
transaction
{
	attribute Boolean noAmelioration;
	attribute Boolean aggravation;
	attribute Boolean undesirableEvent;
	attribute Boolean other;
	attribute Boolean goOnTreatment;
	attribute String treatmentMode;
	attribute Boolean treatmentModif;
	attribute Instant treatmentModifDate;
	attribute Boolean followingProtocol;
	attribute String otherTreatments;
	attribute Boolean undesirableEffects;
}
```

2.20 Class TrialEnd

```
interface TrialEnd
transaction
{
      attribute Instant timestamp;
      attribute Boolean prematureExit;
      attribute Instant prematureEnd;
      attribute Boolean undesirableEvents;
      attribute Boolean noFollowingProt;
      attribute Boolean inefficacity;
      attribute Boolean lostOfContact;
      attribute Boolean others;
}
```

2.21 Class TrialTreatment

```
interface TrialTreatment
transaction
{
      attribute Instant timestamp;
      attribute Interval realPeriod;
      attribute Boolean followingProtocol;
      attribute String noFollowingReas;
}
```

2.22 Class ConcomitantTreat

```
interface ConcomitantTreat
transaction
{
      attribute String internationalAppel;
      attribute Short dose;
      attribute String unity;
      attribute Short noOfUses;
      attribute String administrationWay;
      attribute String indication;
      attribute Boolean treatUndesEvent;
}
```

2.23 Class UndesirableEvent

```
interface UndesirableEvent
transaction
{
      attribute String diagnostic;
      attribute String symptoms;
      attribute String frequence;
      attribute String severity;
      attribute Char evolution;
      attribute String relationTrial;
      attribute String therapeuticAttit;
      attribute Boolean isSerious;
}
```

Appendix C: Examples for extended functionality

1. Supplier-Part-Job

The following example is an object-oriented version of the Supplier-Part-Job database, along with some temporal extension. Note that only valid time is defined for the attributes of these objects. Also note that valid time is of type state and that the granularity is day for all the attributes.

```
interface Supplier: Person
( extent Suppliers
  key ident)
{
      attribute String  ident;
      relationship Addr address valid state granularity day;
      relationship Set<Part> inventory valid state granularity day;

      Supplier RecvOrder (inout Supplier FromSupplier,
                        in Set<Part> RecvParts)
                        raises no_such_supplier;
}

typedef Struct componentType {Part P, Short Qty};
interface Part
(extent Parts
 key Num)
{
      attribute String Num;
      relationship Addr address valid state granularity day;
      attribute String color valid state granularity day;
      relationship Set<componentType>  components
            valid granularity day;
      relationship drawing plan;
      relationship List<Part> BillOfMaterial valid granularity day;

      Part Order(in Part part_order) raises no_such_part;
}


interface Job
(extent Jobs
 key Num)
{
      attribute String Num;
      relationship Addr address valid state granularity day;
      relationship Set<Part> PartsNeeded valid granularity day;
      relationship List<Supplier> Preferred_Suppliers
            valid state granularity day;

      Job NewPart(in Job this_Job, in Part new_part)
            raises no_such_job;
}


interface Addr
{
      attribute String Street;
      attribute String City;
      attribute String State;
}
```

2. Countries

We have a database containing data about different countries around the world. Among those data we have inflation rates, population figures, surface area, capital, and currency used in country. The definition of this database should be the following:

```
interface Country
(extent Countries,
 key name)
{
      attribute String name;
      attribute Float inflation valid granularity month overlaps;
      attribute Float population valid event granularity day;
      attribute Float surface;
      attribute String currency;
      relationship City capital inverse City::is_capital;
      relationship Set<City> cities valid state granularity year
            inverse City::belongs_to_country;
}


interface City
(extent Cities)
{
      attribute String name;
      relationship Country is_capital inverse Country::capital;
      relationship Country belongs_to_country valid granularity year
            inverse Country::cities;
}
```

The idea is that a Country is always defined by its name, however more than one countries can have the same name at different time points. Each country can have an inflation rate that is unique over a time period, but overlapping valid time periods are allowed. In the case of the inflation rate the granularity of the periods we are using is month.

As far as the country's population is concerned we record it as an event, that is it has associated with it the timestamp of the date during which the census was made at this country. No overlapping definition is made here.

Each Country has a set of cities that belong to it and one of those cities is the country's capital. The former is recorded by two mirror relationships which both have valid time characteristics at year granularity. Each city can be part of only one country at any given moment. Another point is that the name of the City may not be unique, as many cities have the same names.

3. Employees - Salaries

We have a company that has employees working at different departments. For each employee we want to know his name, his salary, if he is manager in a department and the department for which he works. For each department we want to keep track of its name, its employees and its manager.

For the employee's salary we want to keep the history of it, as well as a log of every transaction made on it. For these reasons we define this attribute to have bitemporal characteristics.

We also want to keep the history of the departments in which the employee was manager, so the corresponding relationship has valid time information. Finally we

want to log any changes made about the relationship *works_for* for the Employee, so this relationship has transaction time information.

The inverse relationships of the above mentioned ones, have the same temporal behavior with them.

```
interface Employee
(     extent Employee
      keys (name, point salary))
{
      attribute String name;
      attribute Short salary valid granularity month transaction;
      relationship Department isManager valid granularity month;
      relationship Department worksFor transaction;
};

interface Department
(     extent      Department
      key   name)
{
      attribute String name;
      relationship Set<Employee> hasEmployee transaction;
      relationship Employee withManager valid granularity month;
};
```

Appendix D: References - Bibliography

[T22TR21] "Requirements Analysis", GlaxoWellcome S.A., Delta S.A., Matra Cap Systèmes, 01 Pliroforiki S.A., Univ. of Sorbonne.

[T23D1.1a] C. Souveyet, R. Deneckère, C. Rolland, "Part II: The Object Oriented Methodology (OOM)" TOOBIS - Deliverable T23D1.1

[T23D1.1b] Université de Sorbonne, "Part III: Reference Manual of the Temporal Extension", TOOBIS Deliverable T23D1.1, draft version.

[T31TR1] Matra Cap Systèmes, "TODM Specification and Design"

[ODMG1] The Object Database Standard: ODMG-93, Release 1.1, Edited by R.G.G Cattell, Editions Morgan Kaufmann

[ODMG2] ODMG ODM and ODL draft of version 2.0, available at www.odmg.org