Université Libre de Bruxelles
Faculté des Sciences Appliquées
Ecole Polytechnique

Année académique 2000-2001

Integration of spatio-temporal data in Databases: Design of a schema editor for MurMur

Directeur de Mémoire:
Prof. Esteban Zimányi

Travail de fin d'études presenté par Louis Jacomet en vue de l'obtention du grade d'Ingénieur Civil Informaticien

Acknowledgements

I wish to thank all the people who helped me realise this work. Although this is a personal work, it could not have been achieved without their help.

I first wish to thank Professor Esteban Zymányi for providing me with such an interesting topic. I also thank him for the support given throughout the whole realisation of this work and to have offered me the opportunity to go in Lausanne, Switzerland, for two months.

I also wish to thank the team at the Laboratoire de Bases de Données at the Ecole Polytechnique Fédérale de Lausanne. Professor Stefano Spaccapietra, head of the laboratory, for accommodating me in his team. PhD Sophie Monties for her help through my time in Lausanne. Manish Aggarwal and Claudiu Ciba for taking me in as a fellow programmer.

I also wish to thank all my relatives. Kathleen Schatt for her ever present love and support. My family for supporting me through this. My father and mother, ever present and loving. My sisters and brothers for making me laugh. My friends, especially Nicolas and Matthieu, for changing my mind when it was needed.

A special thank goes to my fellow students, for the mood, the partying and all other things shared. While they can not be all listed here, some of them will make it: Olivier (Arlon), Jim (Java Interfaced Man), Mike, Obu, Canard, Fred, and many more.

Contents

1 Introduction **1**

1.1 Scope of the work 2

1.2 Outline of the work 2

2 Modelling spatio-temporal and multi-represented information **4**

2.1 Space and time 4

2.1.1 Discrete and continuous views 5

2.1.2 Valid- and transaction-time 7

2.1.3 Time and space integration 8

2.2 Multi-representation 10

2.3 State-of-the-art 17

2.3.1 Spatial models 17

2.3.2 Temporal models 18

2.3.3 Spatio-temporal models 19

3 The MurMur project **23**

3.1 Project objectives 24

3.2 Data model specifications 26

3.2.1 Abstract data types for space and time 26

3.2.2 Space- and time-varying information 28

3.2.3 Space, time and generalisation links 30

3.2.4 Constraints 32

3.3 Multiple representation support 35

3.3.1 Multi-representation stamps 35

3.3.2 Correspondence relationship types 38

3.4 Conclusion 44

4 Implementation of the data structure **45**
4.1 Proof of concept . . . 46
4.1.1 Independent data structure . . . 46
4.1.2 Example of schema editing . . . 47
4.1.3 Validation . . . 50
4.2 General analysis . . . 55
4.3 Functional analysis . . . 56
4.3.1 The schema and other elements . . . 56
4.3.2 The domains . . . 59
4.4 Implementation . . . 60
4.4.1 Package `mads.tstructure.core` . . . 61
4.4.2 Package `mads.tstructure.domains` . . . 66
4.4.3 Package `mads.tstructure.utils` . . . 68
4.4.4 Package `mads.tstructure.utils.exceptions` . . . 68

5 Conclusions **71**
5.1 Personal challenges . . . 72
5.2 Issues to be considered for future works . . . 72

A MADS syntax **73**

B Contents of the CD-ROM **81**

Bibliography **84**

List of Figures

2.1 The eight binary spatial relationships, applied to an example. 9

3.1 Hierarchy of MADS spatial abstract data types. 27
3.2 Hierarchy of MADS temporal abstract data types. 28
3.3 Timestamped objects and generalisation. 31
3.4 Inheritance, refinement, redefinition and overloading. 31
3.5 Example of aggregation relationship type. 38
3.6 An example of SetToSet correspondence. 39
3.7 An example of a SetToSet relationship. 39
3.8 Hierarchy of correspondence relationship types. 41

4.1 Schema editing: object creation 48
4.2 Schema editing: object properties 48
4.3 Schema editing: role properties 49
4.4 Schema editing: attribute properties 50
4.5 Sequence diagram of `validate` in a schema (1) 51
4.6 Sequence diagram of `validate` in a schema (2) 52
4.7 Cross relationship type validation 54
4.8 General class diagram of `mads.tstructure.core` package. 62
4.9 Relationship types class diagram. 64
4.10 Topological and synchronisation relationship types class diagrams. 64
4.11 Properties class diagram. 65
4.12 Links and groups class diagram. 66
4.13 General class diagram of `mads.tstructure.domains` package. 67
4.14 Predefined domains class diagram. 68
4.15 Class diagram of `mads.tstructure.utils.exceptions` package. 70

Chapter 1

Introduction

Contents

1.1 Scope of the work 2

1.2 Outline of the work 2

Computer applications have dramatically evolved in the last few years. The creation and the globalisation of networks have had a significant impact on the use and development of computer tools, resulting in a broadening of the application users' profile. The multiplicity of profiles asks for adapted representations of information. This leads to the notion of *multi-representation* where a single model can be customised and adapted to the needs of different user profiles.

The complexity of applications has increased along with their deployment. Whereas information could be presented in a scientific way to people expert in the application's field of interest, generalisation asks for more easily understandable descriptions. The increased power of computers also allows for sharper representations of reality. In such a context, *spatial* information can be efficiently managed to the point of providing photo-like end-user views. On the same level, the notion of *time* can be introduced in applications, to deal with historical data, for example.

With all these expectations in mind, it is obvious that data modelling facilities can and should be expanded. In order to develop an application, or even a data model, including space, time, and multi-representation, a high level of abstraction is needed. This can be achieved by using *conceptual* modelling. This approach allows for effectively remaining on a higher level, describing characteristics and features, while staying independent from the

implementation. This is a quality, rarely brought to the fore, which is important in today's context. Moreover, the use of conceptual schemas, and their visual diagrams, represents an easy-to-understand medium of communication between designers and end-users.

1.1 Scope of the work

My work takes place inside the MurMur – Multiple Representation - Multiple Resolution – project which is funded by the European Commission. The main project objective, detailed in Section 3.1, is to design a conceptual spatio-temporal multi-represented data model.

Conceptual because the goal is to provide modelling facilities that effectively expand the actual available solutions with a middle-ware application. In order for it to be used on top of existing databases, it must be above implementation concerns.

Spatio-temporal because although some model implements such features, few are integrating the two. And among these, limitations are frequently set, reducing their possibilities.

Multi-represented because, today, users are less prone than before to adapt to a computer application. They wish for the application to adapt to them.

With the review of the above mentioned concepts under MurMur's point of view, I intend to show that MurMur's scope is more than just a theoretical study of actual advanced modelling.

In order to propose a working solution, an application of schema editing is one of the tools to be built. It will allow to fully exploit the features of the underlying data model. The conception and implementation of the MurMur schema editor is realised at the Laboratoire de Bases de Données of the Ecole Polytechnique Fédérale de Lausanne, in Switzerland. In this context, I was integrated in the programmers' team to work on the design of the editor. After the work repartition, I was asked to propose an implementation of the data structure for the MurMur schema editor.

I present in here the results of this work.

1.2 Outline of the work

Chapters 2 and 3 deal with the theoretical background. More precisely, Chapter 2 covers the theory behind space, time, and multi-representation as well as a state-of-the-art review in data modelling. This allows to present the needed concepts to understand MurMur's objectives, while the review shows the pertinence in MurMur's approach. Chapter 3 de-

scribes the model designed by the MurMur partners in detail. It reflects the review made in order to apprehend the concept that are to be implemented in the various MurMur tools. Understanding of typical entity relationship, object-relational, and object-oriented models is assumed.

Chapter 4 assesses the work done, emphasising its practical usefulness for future applications. The usefulness is proven through examples of use, while the work done is presented as the project has been analysed.

Conclusions are drawn in Chapter 5.

Chapter 2

Modelling spatio-temporal and multi-represented information

Contents

2.1 Space and time 4

2.1.1 Discrete and continuous views 5

2.1.2 Valid- and transaction-time 7

2.1.3 Time and space integration 8

2.2 Multi-representation 10

2.3 State-of-the-art 17

2.3.1 Spatial models 17

2.3.2 Temporal models 18

2.3.3 Spatio-temporal models 19

This work is based on three main concepts. First and secondly, spatiality and temporality, discussed in Section 2.1; thirdly, multi-representation, discussed in Section 2.2.

With a coverage of the state-of-the-art, Section 2.3 offers a view of different existing data models, classified as: spatial, temporal, or spatio-temporal.

2.1 Space and time

Space and time are two basic building blocks of reality. Every one needs the notions of "Where ?" and "When ?", either to locate himself, or to apprehend the surrounding world.

Considering this fact, being able to modelize these two concepts when representing reality is capital. But space and time modelling can be done in several ways. First, one needs to decide if a continuous or discrete view is necessary, then the way time and space are in relation is to be considered.

2.1.1 Discrete and continuous views

Discrete view

For spatiality, the discrete view is the association of a spatial value, usually a geometry, to a spatial entity. Similarly, each temporal entity is associated with a lifespan describing its temporal extent.

The usual way to define these geometry and lifespan is through the use of Abstract Data Types (ADT). They form closed sets, since operation results are always elements of the set. Many researches have been done to define the "best" set of ADTs. The ISO standard, in collaboration with the OpenGIS consortium, is defining a standard for SQL3 [1]. It is a large and complete set, organised as a hierarchy and containing two kinds of types, Geometry (GM) and Topology (TP). Both describe geometric characteristics, shape and location, of the spatial feature. But whereas the first, GM, associates location with the spatial coordinate system, the second, TP, uses topological features to relate elements with others. This model thus benefits from an adapted set of ADTs for describing schematic maps, where topology precedes spatiality. An example is a schematic water distribution map, in which connections between pipes are more important than their exact location.

Formally, a spatial extent is an infinite set of points, the same holding for a temporal extent with an infinite set of instants. Most models use boundaries to represent these extents. A time interval is thus defined by its beginning and end, and a polygon by the extremities of its vertices. Constraint databases propose another solution: each extent is specified by a set of constraints, usually linear inequalities, which defines their set of points, in space or time. The downside of such constrained models is their complexity, making their use quite difficult.

Several ways of supporting discrete spatial or temporal features exist. They are presented here:

- **Attributes:** Each entity can have zero, one or multiple spatial/temporal attributes. For temporality, nearly all current data models offer at least the Date data type.

- **Entities (classes or tables):** Each entity has a peculiar attribute. It can be *geometry* for spatial entities, or *lifecycle* for temporal ones. Spatial/temporal information is then given in these attribute definitions.
- **Patterns:** The data model offers a set of predefined spatial and/or temporal classes that describe all the possible extents. These classes do not contain any thematic information. To represent real-world entities, spatial or temporal, they have to be linked to one of the predefined entities.

The third approach is the least conceptual. It forces database designers to scatter a real-world entity among multiple entities, one for thematic information, one or more for spatiality, and similarly for temporality. However this approach allows to add spatiality and temporality to an existing database just by adding new entities/classes.

Another desired feature, presented in [2], would be the possibility to characterise spatial or temporal relationships between entities, with eventually a number of thematic attributes. For example, representing crossroads and their signposts. In order to do so, a representation with relationship semantics is needed, since temporal and spatial relations do not exist when there are no entities.

Continuous view

Space and time considered in a continuous way, are characterised by the type, domain, and range of their defining function. Frequently, for space-varying data, the attributes are defined for the whole extent of the database [3]. However, this is not always meaningful, as for example the depth of a lake is relevant only in the area covered by the lake. Moreover, at the same location, two different values may coexist for the "same" space-varying attribute. For example, when measuring toxicity in a river for two different waves of pollution, two different values coexist.

The range of the functions can be any set of values. However, support for complex or multi-valued values is usually not implemented [4]. Lastly, the type of the function specifies which kind of interpolation will be used when answering queries for points where the value has not been measured. Most frequent types are:

- **Discrete**, such as mines in a field.
- **Stepwise**, e.g., constant in a region, such as for land occupation.
- **Continuous**, such as for land elevation, continuous in space and time.

A few data models offer such conceptual views. For instance, designers are enabled to choose different kinds of functions, including stepwise, linear, and user defined, in Spatio Temporal Unified Modelling Language (STUML) [5].

Most implemented data models support more logical concepts, such as grids or tesselations in some GIS. Temporal Database Management System (DBMS) usually associate time intervals to attribute values or to tuples.

Conclusion

The discrete and continuous views arise from the dual nature of perception, and consequently description: a same phenomenon can be perceived as either discrete or continuous. The mental activity of perception applies many filters to human vision, and these filters are influenced by the personality and the objectives of an individual. Therefore, data models that are used by humans should provide the choice between the two views, so that users can describe and manipulate information according to their perception of reality. Moreover, some cases require combination of the two views. This is the case for a moving car, viewed as a time-varying point.

As will be shown in Section 2.3, few models support both views, or with limitations. This one of the points targeted by the MurMur proposal, described in Section 3.2.

2.1.2 Valid- and transaction-time

Valid time is defined as the time when a value has been, is, or will be valid. It is opposed to the notion of transaction-time, which applies to the moment a fact is known to the information system. For example, the fact that someone died at 2pm on May, the 29^{th} 2000 is valid time, while his relatives knowing it only the 30^{th} is considered as transaction-time.

Handling transaction-time would mean holding in memory all values, even deprecated ones, and their time of introduction or change. However, this may be critical for some applications. Assuming a risk management application, transaction-time is very important. Assume that the risk-management application marked a region as safe from floods. Then, one day, the region is flooded. Individuals may attack the organisation, stating that they should have been warned. In this case, use of transaction-time may allow to determine if the information possessed at the time of the query could have led to another answer.

2.1.3 Time and space integration

Although space and time are concepts inherently related, they are rarely thought of and represented on equal terms. This section describes different views of space and time interactions, according to a classification done in [6].

The space-dominant view

In this view, space is considered as *absolute*, that is, infinite, homogeneous, and isotropic, with an existence fully independent of any entity it might contain. Space is viewed as a container, it can be split into layers, to combine a variety of elements, grouped by themes. Time is incorporated implicitly, every time a change occurs. As a result, a new snapshot of a layer is created every time the database is modified. However it is not known how an updated layer might affect other associated layers. The layer-based raster or vector models of GIS are examples of such space-dominant views. These models group geographic space after some sort of categorisation, while time is grouped after some sort of periodization. Constituting history is based on similarity or dissimilarity between aggregations at different points of time.

Space can be viewed as continuous or discrete. However, time, generally perceived as continuous, is represented in this model as a discrete subset of the real numbers. Therefore changes are limited to a finite number of times so that the sequence can be indexed.

The time-dominant view

When time is explicitly part of the representation, with or without reference to space, the time dominance is generated. In this case, time is viewed as *absolute*, and creates a fourth dimension, along which events, observations, or actions can be located. A time structure exist, with a temporal logic, allowing statements about actions, observations, or events to be either true of false at various points in the time structure.

Time-dominant structure can be classified into three categories:

- **Interval-based models:** Temporality is specified using regular or irregular intervals [7].
- **Point-based models:** Temporality is specified using explicit occurrences of an event, observation, or action. These models are usually implemented as time maps. In such maps, nodes are events, observations, or actions, and edges represent the relationships between these.

- **Mixed models:** Temporality is specified as a combination of the two above models.

The absolute space-time view

Both space- and time-dominant views have had an important place in research, many in the GIS community for space, and in the database world for time. The absolute space-time view is an integration of the two previous views. TEMPEST (Temporal Geographic Information System) is an effort toward such an integration, as is MurMur. The first is described in [8].

The relative space view

Here, space is defined as *relative*. This implies that space is created by relations between entities. In other words, defining a relation automatically creates space. While the concept of absolute space overemphasises the absolute location of entities, relative space focuses on the relative location within a spatial representation. This point of view is usually associated with studies of forms and patterns.

Eight binary relationships are defined in [9]. They are given in Figure 2.1, along with a visual explanation [6].

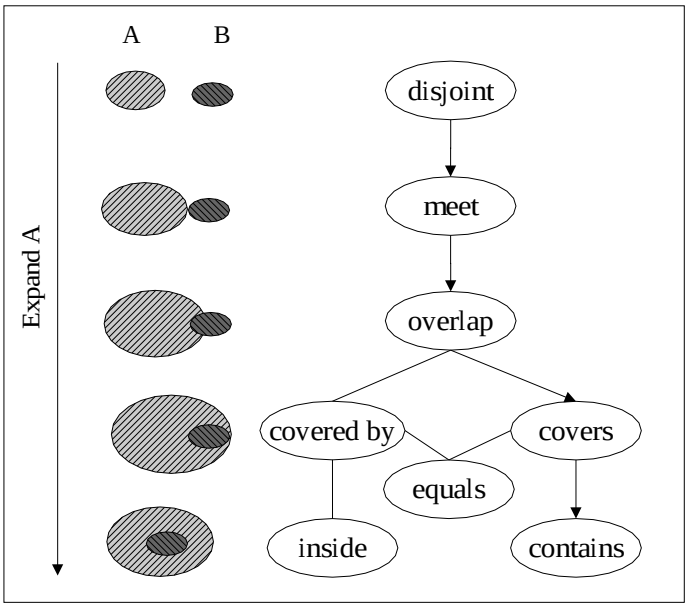

Figure 2.1: The eight binary spatial relationships, applied to an example.

The relative time view

Here time is considered as measured in relation to something, not constrained to a single dimensional axis. Cyclic time, such as the repetition of a day or a week, is an example of relative time. It is a subjective view, since it assumes a flexible structure defined in terms of relationships between events.

The relative space-time view

Space and time are viewed as coexistence (connection or togetherness) relationships between changes and events. There is no notion of space without time and reciprocally. Space may be non-Euclidean, and time non-linear.

Conclusion

While the absolute views require some sort of measurement reference, implying non-judgemental observation, the relative ones involve interpretation of processes, and changing patterns within a knowledge domain. As such, they form complementary views, and one should not be discriminated over the other one.

However, the question of integrating the two models remains. The MurMur proposal, described in Section 3.2, targets this integration.

2.2 Multiple representation features

The real world is supposed to be unique, however its representations are certainly not so. A geologist does not need the same cartographic information as a tourist. Thus different applications suppose different representations of the same phenomena. In order to represent a real-world phenomenon, multiple facets must be describe, such as:

- The information that is kept: a big company can have one human resource application, with restricted visibility for its various departments, or it can use separate applications.
- The way information is described: the colour of a car is relevant for the buyer, not for the tax office.
- The way information is organised: for the electrical company crossroad lights are entities, while they are attributes of the cross road for the road management.

- The way information is coded: currency in Euro or US Dollars for instance.
- What constraints apply: an employee is assigned to a department for the logistic, while being externally funded, he is not par of it from a financial viewpoint.
- The way information is presented: order of employees by name, seniority, or department.
- Associated spatial and temporal framework: yesterday's employee versus today's, data for this county versus data for that other one.

Current data management applications rely on a centralised representation. Views are used to derive specific representations for applications. However the view mechanism has strong limitations. In relational DBMSs, updates of the database are usually impossible from a view, due to the fact that they frequently do not rely on a 1:1 mapping between tuples in the view and in the database. In object-oriented DBMS, the view definition is further restricted because the data model rules may lead to inconsistencies in the hypothesis of complex view definitions. On the other hand, the generalisation/specialisation mechanisms found in object-relational or object-oriented DBMS provide some support for additional multiple representations, but it is not sufficient, as will be shown later.

With centralised representation mechanism, no support exists for the cases where application viewpoints are not derivable from each other. Let us consider an hospital information system, in which patients are identified by medical teams thanks to a number they carry on a bracelet, whereas the same patients are identified by the administrative staff through their social security number. If the two viewpoints do not share any other information that could provide a common identification scheme, then update propagation, such as for a patient leaving the hospital, cannot be done automatically inside the database. This would lead to an increase the maintenance costs as well as to a serious risk of inconsistencies. However, from a traditional centralised database perspective, two irreducible viewpoints cannot coexist and may be considered as a design error. From a user perspective, it is not.

The centralised paradigm is even more uncomfortable when a database results from the integration of different preexisting data sets. This situation arise frequently when designing systems that allow interoperability between different companies, or coalesce data from different sources.

Spatial databases add one more dimension to multi-representation: the resolution, used to characterise geometric features, in particular on maps. Objects may have a geometry depending on scale on a map. For example, a building can be rectangular on a 1/10,000

map, a point on a 1/25,000 one, and not represented at coarser scales. Thus physical zoom-in and zoom-out operations are inadequate. Drawing standards are changing from one scale to another one, depending on the available information, or simply the information may not be available at the requested scale. Unfortunately, there is no complete set of algorithm to do cartographic generalisation, i.e., deriving a map at some scale from a more precise one. Also, the cartographic generalisation may be a long and costly process.

Because of this, the ideal would be to store inside the database spatial objects associated to multiple representations, scale dependent. In order to broaden the scope of the discussion, the term *resolution* is best suited for spatial objects in databases, a more generic concept than maps. A multi-resolution spatial item is an item that is associated with multiple geometries.

Moving among resolutions has a potential impact on different facets of spatial objects, such as:

- multiple geometries, possibly belonging to different spatial types, like point and area,
- multiple abstraction levels that make a set of objects coexist with the object(s) that represent their aggregation,
- multiple abstraction levels that result in a hierarchical value domains for attributes,
- multiple representations in terms of thematic information, which corresponds to maintaining several viewpoints as in traditional databases.

In all these cases, it is the responsibility of the application designers and users to rely on the system functionalities, foreign keys and generalisation, to inter-relate different representations, and ensure consistency in the resulting application.

Supporting multi-representation means that both the user and the system are aware that two or more stored representations are representing the same real-world phenomenon. To achieve this, existing models must be extended with new concepts such as multi-representation links, with a well defined semantic and associated constraints and operators. Expected benefits include better real-world modelling, enhanced understanding of schema diagrams and database content, improved consistency management, automatic update propagation, and data cleaning facilities [2].

In order to represent the real-world, objects, links between events, and their static and dynamic properties (attributes and methods) are used inside the database. Since

representations vary according to different criteria, the representation space may be seen as multidimensional, where dimensions are:

- **The spatial resolution dimension:** this axis represent the spatial resolution ranges for which representations holds. Coordinates may be seen as an ordered set, for instance from the finest to the coarsest resolution.
- **The observer or viewpoint dimension:** this axis represent the different viewpoints for which representations are elaborated. Coordinates are a set of discrete non-ordered points.
- **The classification dimension:** this axis represent object instances of a given object type. Coordinates are discrete and clustered according to object types.

A point in this three-dimensional space is the representation of an object instance as a member of the population of the given type, according to a viewpoint and to a given resolution range. Multiple points may hold identical position in this space.

Each instance in the database has to be somehow characterised with respect to the axes in use in the multi-dimensional space. An instances could be stamped with pairs <coordinates, axis>that locate it within the space. In GIS terms, the axis would be metadata, and the coordinate the corresponding value. Instead of being defined at the instance level, metadata could be defined at a higher level of abstraction, automatically applying to instance of that level. This freedom of design opens a variety of approaches in supporting multiple representation according to multiple dimensions. The actual focus of research is more on one-dimensional multi-representation. With the same approach, each dimension is presented separately.

Multiple resolution

Data about the same geographic space may be managed at various resolution levels, to serve different applications need. An embedded navigation application needs data at different resolution, fine for the arrival and departure are, coarse in between. Also, multiple resolution may be a consequence of integrating data coming from varied acquisition sources.

To support multiple resolution, the following approaches have been advocated. The first one propose a single instance, multi-represented. The second propose multiple single-represented instances.

One multi-resolution instance One object in the real world translates into one instance in the database. This instance is allowed to bear multiple geometries, qualified with their resolution range. Various forms of simplification are done when moving from a finer to a coarser resolution.

Multi-resolution however does not reduce to multiple geometries. Relationships between and among objects may change. Even thematic attributes of objects or their values may change. For instance, a road may hold information about the number of lanes at a fine resolution, while this information is irrelevant at coarser resolutions.

If this integrating approach is also used for the viewpoint and classification dimension, the result is one instance holding all possible representations. This may be result in an object too cumbersome to use, due to the complexity of changes going from one representation to an other.

Many single-resolution instances One way to reduce complexity is to split the representation of a real-world object between multiple, interconnected representation, each one represented as an object instance in the database. The existence of multiple instance raises three questions:

- **How the instance are classified:** into one class in one database schema, into multiple classes in one schema, or into different classes belonging to different schema.
- **How the instances are related:** implicitly, through their identification mechanism, or explicitly through links.
- **Which properties are associated to each instance:** all properties explicitly or only properties specific to the instance, with others inherited from other instances.

Concerning classification, current proposals advise to handle multiple resolution objects through a set of single-resolution schemas. The schemas may eventually map to one physical database, or to different ones [2].

Regarding inter-instance links, the explicit method is advocated. The semantic of such a link is that the linked instance "represent the same object at different resolutions". This is close to the is-a link semantic, but it does not obey the inclusion semantic. Two types for the same object at different resolutions will generally have intersecting populations rather than one included in the other. For example, assuming a database on roads, some will disappear at coarse resolution, while other will be merged into new road objects. In this

case, several links may be needed to represent all the correspondences between the two populations.

As for properties, associating the whole set of properties that are relevant for that instance result in flexibility and self-contained manipulability. However this approach needs a number of integrity constraints to ensure that properties representation independent hold the same value in all instances. This would be time-consuming and thus more research is needed to extend the inheritance approach to object types with intersecting populations.

Multiple viewpoint

A viewpoint expresses information requirement from a given set of users. It holds the specification of both the data structure (object and relationship classes, attributes) and the rules for data usage. The discussion here is limited to the attributes part, extending to methods if using an object-oriented database. The question of change is the topic of the next section.

Two approaches are advocated to deal with multiple viewpoints. They are the same as for resolution. In the case of viewpoints, the second approach prevails.

Since the beginning of databases, the view mechanism exists. While very poor at first, relational databases improved it a lot. These system have a very powerful restructuring ability, as arbitrary algebraic expression can be used to build a view. However, users are responsible for adding the necessary artificial and foreign keys to link related tables and for providing procedures to enforce integrity.

Object-oriented, or object-relational, database systems fail in supporting the same flexibility. This is due to the use of object identity and complex object structure that make inserting new objects, i.e., view results, inside the structure a hard to solve problem. On the other hand, generalisation/specialisation hierarchies provide some support for multi-representation. However this will be shown as insufficient in terms of expressive power, user-friendliness, and practicality.

Multiple classification

Classification is a subjective abstraction in data modelling. It allows to get rid of details, and to talk in terms of object classes, their relationships and the properties to be attached. Classification is very likely to change depending on the viewpoint adopted, or even through the evolution of properties of objects. Even from a single viewpoint, multiple classes may be used to represent a given object.

Semantic and object-oriented data models support this by providing the is-a link to define generalisation/specialisation hierarchies. However this link is not appropriate for arbitrary classification, where a population is not included in the other. Another limitation is the static aspect of current generalisation/specialisation hierarchies. Objects are not allowed to move from one class to the other and cannot be part of two leaf classes. The role concept, the result of significant research efforts, offers an alternative. It is a classification where objects acquire membership in several role classes, with the freedom to release this membership at any time [10].

Roles provide a solution to support multi-representations such that each representation is materialised into one database instance. Moving from object to roles raises the issue of inheritance.

Inheritance The reuse of the object-oriented automatic inheritance, late binding, refinement, redefinition, and overloading on inter-role links is not possible. Two basic alternatives have been proposed to replace or complement the automatic inheritance and late binding approach. The first one is known as delegation. The definition of an object/role type includes attributes whose value is not stored within the instance. They are derived from the corresponding homonym attribute in the corresponding instance belonging to another object/role type. The net effect is similar to inheritance but restricted to a subset of attributes freely chosen by the designer [11].

The second solution allows to specify the desired inheritance as part of query formulation. This provide a sort of adjustable dynamic binding. When accessing an object, the user has to specify which other object/role types can be accessed to find the desired property if not found in the type directly denoted in the query. This is defined as the scope of the query. Moreover, the user can specify in which instance population to search. This is referred as the selected viewpoint of the query. With the two above mentioned specifications the user has a complete control on which object properties have to be accessed [12].

Object creation When an object has multiple representations in distinct instances, the question of whether there are rules governing creation or migration of instances arises. For example, in proposals that assume coexistence of the base object types and multiple role types, objects must be created first in the base type. Once created, they may generate instances in role types, but cannot disappear from the base type while it is represented in a role type.

To govern the work-flow of membership behaviour, the definition of predicates can be used. They may be automatically checked, every time an object instance is modified, if

it satisfies a predicate then it gets membership in the role type. The other approach is a check on demand. Inference rules may be associated to each object/role type, specifying which other types may or may not be populated by migrating instances.

Conclusion

In this generic description of a framework to address multi-representation support, issues and solutions have been investigated. Despite similarities, different approach depending on the dimension have been shown. The approach of multi-representation may also differ between users' view and implementation. The focus was on object type, but the concern may be extended to relationships and attributes.

MurMur's answer is presented in Section 3.3.

2.3 State-of-the-art

The review presented here is adapted from [2]. It presents data models classified as spatial, temporal, or spatio-temporal.

2.3.1 Spatial models

GeO2 [13]

GeO2 is a GIS implemented by the research lab of the French mapping agency IGN on top of the O2 database management system.

The models support a unique generic spatial abstract data type, called *geometry*. A set of spatial operations and functions are attached to *geometry*. They form a closed set of operations. Each entity type may have zero, one, or several spatial attributes.

CONGOO [14]

CONGOO is a spatial data model based on an object-oriented design. The thematic concepts of the data models are the classic ones from the object-oriented approach. Spatial real-world entities are represented by objects that have a spatial type which can be elementary or composite. Elementary types are *Point*, *Line*, and *Area*. Composite types are *Set-of-Points*, *Set-of-Lines*, *Set-of-Oriented-Lines*, *Complex-Area*, and *Heterogeneous-Set*. A last type, *Alternative*, is a generic type which means that instances may be of different spatial types.

There is no spatial attributes. Therefore when a spatial real-world entity has several spatial properties, each property has to be represented by a spatial object.

Spatial classes whose instances obey the same spatial integrity constrains can be grouped, making up a layer. Layers can be divided in sub-layers. Two topological predicates allow designers to define spatial integrity constraints on the set of instances of a class, a layer, or the whole database.

Visual symbols are used on the schema to denote the main concepts.

2.3.2 Temporal models

TIMEER [15]

TIMEER is a conceptual temporal model, based on an extended Entity-Relationship (ER) model. It is upward compatible with the model it extends, and allow to design non-temporal databases as well as partly temporal ones. The notations of the ER model are extended to indicate which temporal aspect are to be captured. The indications are Ls indicating lifespan support, VT for valid-time, TT for transaction-time, LT for lifespan and transaction-time, and BT for valid- an transaction-time support. Various time granularities are supported: second, minute, hour, day, week, month, and year.

Lifespan and transaction-time support is offered for entities. For attributes, valid- and transaction-time is supported. However, temporal support cannot be added to the components of a complex attribute as it is assumed that all components change synchronously. For relationship types, the model supports lifespan, valid- and transaction-time.

TIMEER provides flexible temporal support since the database designer is a able for each modelling construct to specify whether or not it should be temporal.

T_ODMG [16]

The Object Database Management Group (ODMG) is an industry consortium devoted to establishing standards for persistence of object-oriented programming languages objects in databases. The ODMG standard includes a reference object model, an object definition language (ODL), an object query language (OQL), and bindings for several programming languages.

T_ODMG extends the ODMG data model in an upward-compatible way. A temporal dimension is added to both object attributes and relationships. A relevant feature provided by T_ODMG is the support for temporal, immutable, and static object properties.

A temporal property is a property whose value may change over time, and whose values at different time are recorded in the database. An immutable property is a property whose value cannot be modified during the object lifespan. A static property is a property whose value can change over time, but whose past values are not meaningful and thus are not stored in the database. In T_ODMG, a lifespan is associated with each class, it is assumed to be contiguous.

Temporal Support in SQL3

An temporal extension has been proposed to SQL. The latest version is known as SQL/Temporal.

Temporal support allows to locate events at a point in time or at a period in time between two points in time as measured along some time line. A granule is a period of time of the minimum duration at a given precision. For a period, the beginning bound is the least value, the last element is the element that is greater than any other element, and the ending bound is the least value greater than the last element. The functions BEGIN, LAST, and END return these values.

SQL/Temporal includes a normalisation operation that maps a set of periods to exactly one set of periods that is equivalent and minimal. The latter means that no two elements $P1$ and $P2$ of the result are such that $P1$ meets $P2$.

Rules for period type conversion and mixing of data types are also defined. Predicates PRECEDES, CONTAINS, MEETS, and OVERLAPS are defined on periods using BEGIN, END, and LAST.

2.3.3 Spatio-temporal models

Perceptory [17]

The proposed approach is to define spatial and temporal plug-ins for visual languages (PVL) that can be added to any existing database design tool.

The spatial PVL contains three basic symbols representing the spatial types: *Point*, *Line*, and *Area*. Their composition allows to define the following spatial types: a composite type (points and lines), an alternate type (point or area), a multiple representation (point, area), and a derived geometry. Three other symbols express: a generic geometry, a composite geometry, and cardinalities. A cardinality specify if the geometry is optional, multiple, or both.

A temporal PVL has been defined too. It provides two symbols representing the two basic time types: *Instant* and *Interval*. Mixing both symbols allow designers to describe spatio-temporal phenomena.

Perceptory is the implementation of such a visual spatio-temporal modelling tool. It is based on the Visio CASE tool, itself based on UML.

POLLEN [18]

POLLEN is a spatio-temporal object-oriented design method. For the space and time dimension, POLLEN offers a set of predefined abstract classes. *Point*, *Line*, and *Area* are for space and *Instant* and *Interval* for time. For the discrete view, space and time are described at the object level. A set of spatial real-world object is represented by a thematic class C and a spatial class CS that is a subclass of one of the predefined spatial classes. C and CS must be linked by a relationship. The same mechanism is employed for representing the lifecycle of temporal objects.

For the continuous view, any thematic time varying attribute, A, of an object type is represented by a method in the thematic class of the object type, which returns the value of the attribute at any instant. It is implemented by a new class CA whose instances contains the values of A for each instant (or interval). CA is linked to the thematic class and to one of the predefined temporal class. The method is the same for space-varying attributes.

For topological or synchronisation relationships, they can be represented by thematic relationships. However, constraints must be defined by application developers as methods inside classes.

STUML [5]

STUML is an extension of UML to the space and time dimension. It covers both the discrete and continuous views. The two dimensions are orthogonal and described strictly at the conceptual level.

For space and time description, two graphical symbols are provided. Metadata can be associated to each class, allowing designers to define the space and time models, to precise the space and time temporal domains, to define the interpolation functions for space- and time-varying attributes, and so on ...

STER [19]

STER is an ER model extended to the space and time dimensions. Valid and transaction time can be attached to entities, relationships, and attributes. Valid time describes the lifespan of entities and relationships, and the validity extent of attribute values.

An entity type can be spatial; i.e., it has a peculiar geometry attribute. STER supports three spatial data types for geometries: *Point*, *Line*, and *Area*. As geometries can be defined at several resolutions, an entity can have several geometries. However, STER does not support any means for describing the associated resolutions. Geometries, like any attributes, may be timestamped.

There is no spatial attribute. Therefore spatial real-world entities that have several spatial properties have to be decomposed into several spatial objects. Space-varying attributes are described at the conceptual level. They can be described for the whole database extent but not for a specific extent. Space-varying attributes can also be timestamped.

Spatial relationships are mere thematic relationships whose meaning is spatial, but without any spatial integrity constraints.

A rough translation of STER concepts to classical ER models is proposed in [19].

MADS [4]

MADS is a conceptual spatio-temporal model based on an extended ER model. The three dimensions, thematic, space, and time, are orthogonal. The thematic model is quite powerful. It supports the usual concepts of extended ER models: entity type, relationship type, attributes, specialisation/generalisation (is-a) links. Attributes may be simple or complex at any depth. They may bear optional or mandatory, mono-valued or multi-valued. An attribute may also be derived by an expression from the values of other attributes. Relationships may be n-ary or cyclic, and may bear attributes. Is-a links support refinement, redefinition, and overloading of attributes and methods.

MADS also offers specific kind of relationship types. The aggregation has a specific meaning of composition, or "part of". The generation relationship links newly created entities to entities that caused their birth. The transition relationship allows application to record the fact that a real-world entity changed in such a way that the classification of the database object(s) that it represented changed.

For the spatial dimension, MADS supports both discrete and continuous views. A hierarchy of spatial abstract data types (ADTs)is also used. It will be described in Section 3.2.1. MADS also support the discrete and continuous view for the time dimension. Another set of ADTs is used to represent temporal types. They will also be presented in Section 3.2.1.

Combining the space and time dimension allows designers to describe spatio-temporal phenomena.

Chapter 3

The MurMur project

Contents

3.1 Project objectives 24

3.2 Data model specifications 26

3.2.1 Abstract data types for space and time 26

3.2.2 Space- and time-varying information 28

3.2.3 Space, time and generalisation links 30

3.2.4 Constraints 32

3.3 Multiple representation support 35

3.3.1 Multi-representation stamps 35

3.3.2 Correspondence relationship types 38

3.4 Conclusion 44

This chapter presents the MurMur project, in which the work described in this document was realized. First, details concerning the project itself are given below, then the project objectives and the approach chosen are resumed in Section 3.1 and finally a concise description of the data model on which the work described in this document was made is given in Section 3.2.

3.1 Project objectives

Chapter 2 explained the main concepts around which MurMur is built – temporality, spatiality and multiple representations – and described the state-of-the-art in those domains. This section explains the objectives of MurMur and the approach chosen to realize them.

The project MurMur is a European-funded project, under the 5th Framework Information Society Technology Programme. The acronym MurMur stands for Multiple Representations – Multiple Resolutions. The project involves several partners which are :

- a geodata provider, the French national mapping agency, IGN,
- geodata users, represented by the French public research centre, Cemagref,
- a GIS supplier, Star Informatic, and
- academics with R&D experience in geographical databases :
  - the Université Libre de Bruxelles, ULB,
  - the Université de Lausanne, UNIL and
  - the Ecole Polytechnique Fédéral de Lausanne, EPFL.

The overall objective of MurMur is to provide support for more flexible representation schemes, such as multiple coexisting representations of the same real-world phenomena, achieving thus semantic flexibility. This will be done in particular on representations of geo-referenced data at multiple resolutions or scales, achieving cartographic flexibility, and temporal representations, thus allowing to keep track of the evolution of a phenomena. Nonetheless, along the project's evolution, research will be made and solutions will be proposed to cover non-geographic domains.

There are two main reasons for choosing the domain of geographic applications. The first is the importance of geographic data for the society of today. Up to 80% of all business and policy problems involved in creating livable communities include geographical data as a key element in their decision making process (source: NCGIA 1999 National GeoData Forum, Washington, DC, June 1999). The second reason is that the diversity of user profiles is much broader in geodata applications – from city planners to sociologists – than in conventional database applications, where the database is usually designed to serve users belonging to the same organisation.

The scientific outcome of the project will be the formal definition of an enriched spatio-temporal data model, based on features from both the object-relational and object-oriented models. The model will have the associated definition and manipulation languages, to allow the management of real-world objects at different semantic, spatial, and temporal resolutions. Targeted capabilities are:

- association of several representations to an object;
- defining topological and temporal relationships that may differ with regard to spatial resolution and temporal evolution;
- link objects corresponding to each other at different resolutions and/or different times; and, finally
- describe and query objects and their attributes at different levels of semantic, spatial, and temporal resolutions.

To realise these capabilities, a fully orthogonal model is targeted. The spatial, temporal, thematic, and multi-representation dimensions will be fully independent. This means that thematic entities may or may not be temporal, spatial, spatio-temporal, or all at the same time.

MurMur's objectives include the development, testing, and validation of middle ware applications on top of the Star-Info GIS, product of Star Informatic. This set of applications will be an interface between the users and the underlying GIS software. Design and implementation will be done so they maximise the portability to other GIS and DBMS.

These applications are to provide the following functionality :

- define and edit a schema of a spatio-temporal database using the enriched data model,
- input data into the database and let it evolve through insert, update, and delete operations, while preserving the integrity constraints on related multiple representations, and
- query the database using thematic as well as spatio-temporal criteria.

As proof of concept, this software layer will be tested in two case studies. The objective is to ensure an effective human-computer interaction and to determine the benefits of using MurMur's approach with respect to the actual solutions used to cope with the problems. The first case study concerns the problem of multi-scale databases. The goal is to efficiently

merge the already existing and independent databases, to improve their efficiency. The second case study is a risk-management application dealing with both natural risks, e.g., floods or avalanches, and industrial risks, e.g., pollutions. While this application also needs to manage multiple resolutions, here the temporal aspects, such as history, are of vital importance for risk analysis and prevention.

3.2 Data model specifications

In order to realise its objectives, MurMur needs a powerful conceptual data model. The model must allow definition of clear schema, readable and understandable by the user. It also need to answer all the issues covered in Chapter 2.

The data model described thereafter is an extension of MADS, a conceptual model realised at the EPFL, but lacking the multi-representation concept, which was presented in Section 2.3.3. Its description will be centred around the elements specific to MurMur, leaving the traditional modelling features out. The complete model description can be found in [20].

3.2.1 Abstract data types for space and time

Abstract data types (ADTs) provide MADS with the predefined spatial and temporal constructs. These ADTs consist of data structures and sets of operations. They are used for describing the spatial and temporal extents of the schema elements. Each ADT has an associated icon, to be used in schema diagrams.

Spatial ADTs define the spatial extent of the described feature. The MADS predefined SADTs are: *point*, *line*, *oriented line*, *simple area*, *simple geo*, *point set*, *line set*, *oriented line set*, *complex area*, *complex geo*, and *geo*. No support is provided for 3D, but corresponding SADTs will be added when MADS is extended. However, the design principle allow database designers to define their own SADTs, as required by their applications. Predefined operations are associated with each SADT, such as *length* for line, or *surface* for area. The SADTs' structure and icons are shown in Figure 3.1.

The *point* SADT is used to represent features materialised by a single point. A *line* is a mono-dimensional set of connected points defined by one or more linear (in)equations. Thus are included curves, polylines, and open or closed lines. An *oriented line* is a line with the notion of start point and end point. A *simple area* is a two-dimensional set of connected points that lie inside a boundary formed by one or more disjoint closed lines. If

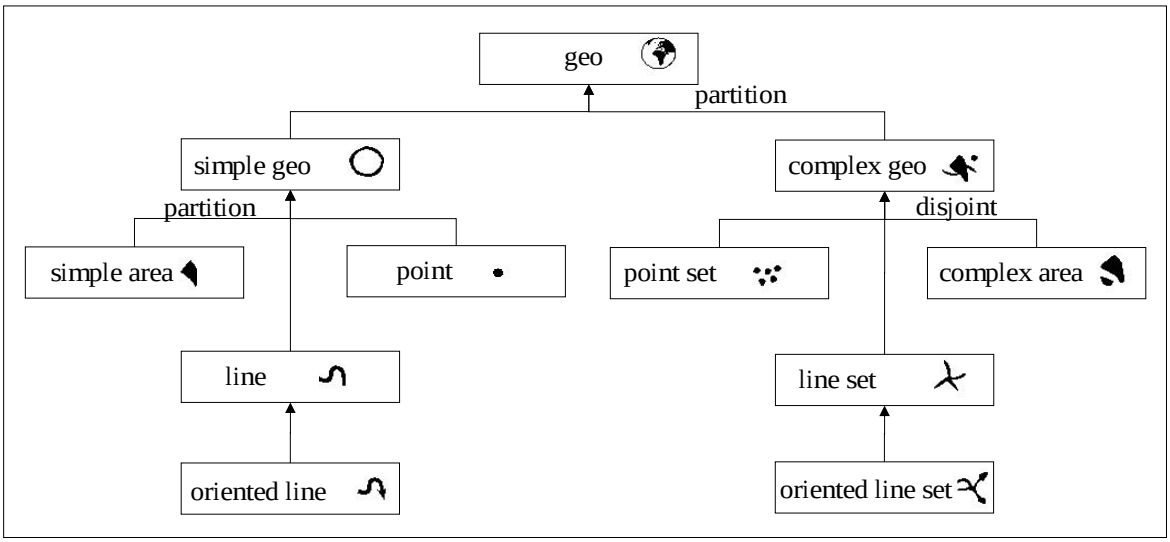

Figure 3.1: Hierarchy of MADS spatial abstract data types.

more than one line form the boundary, then one of the closed lines contains all the other ones, in a 2D sense. This means that an area may have holes but no island, neither exterior nor inside a hole. The three described SADTs are specialisation of the *simple geo*, and they form a partition of it. The *simple geo* is a used as an abstract construct and is thus never instantiated as such.

The other SADTs are used to represent spatially homogeneous sets. These include *point set*, *line set* and its specialisation in *oriented line set*. A *complex area* is a set of non overlapping simple areas. A *complex geo* is a generalisation of the heterogeneous sets and may include points, lines, and areas. Finally, *geo* is the most generic SADT, generalising both simple geo and complex geo SADTs. It is an abstract class and is thus never instantiated as such.

Temporal ADTs define a time extent and are used for timestamping. The two notions of valid- and transaction-time were discussed in Section 2.1.2. MADS currently supports only valid time. Thus timestamping objects, relationships, and attributes allow to provide information on their life cycles, expressing when they were, are, or will be valid.

MADS predefined TADTs are shown in Figure 3.2. An *instant* refers to a single point in time. An *interval* is a set of contiguous instants. The *instant/interval sets* define, respectively, a set of disjoint instants/intervals. The *simple time* TADT is a generalisation of instants and intervals and the *complex time* is a generalisation of the instant and interval sets. The *temporal element* is a generalisation of the simple and complex time TADTs.

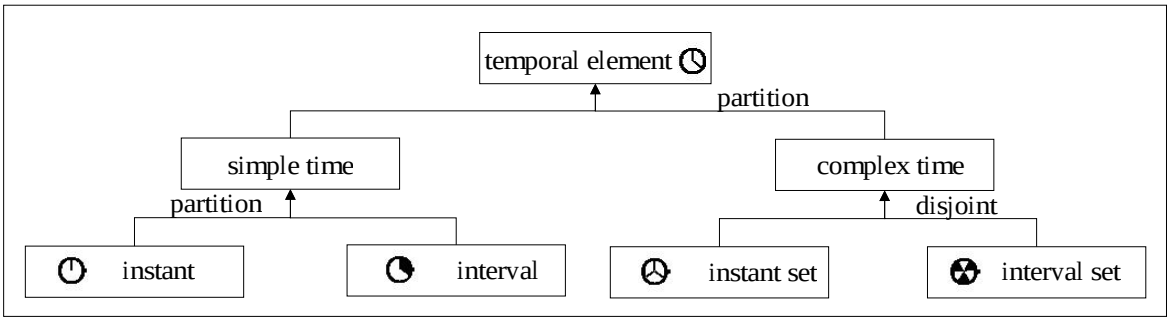

Figure 3.2: Hierarchy of MADS temporal abstract data types.

The generic spatial and temporal ADTs are used as constructs to deal with spatial or temporal elements whose instances may have different types. For example, consider a *Town* object definition, the instances could be either a point or an area depending on the size of the town, thus a simple geo SADT would be used.

To complete these ADTs definitions, two global parameters have been defined, DBSPACE and DBTIME. They are used to define the limits of, respectively, the spatial extent and time frame of the application. These extents are not infinite and have to be know to check the validity of the data or to correctly infer space/time extents associated to the data.

3.2.2 Space- and time-varying information

The ADTs just described support a discrete view of space and time, by characterising an element through its position in space and time. However support for a continuous view is needed, such as to allow description of phenomena that exist over a certain space and/or time. The approach proposed by MurMur to support the continuous view are discussed below, first for space and then for time.

With respect to space, attributes can be roughly categorised as follow:

- Continuous fields describing a property of the space itself like humidity. At most one value exists at a given instant and place for each of these fields.
- Continuous fields describing a property of a one- or two-dimensional object, whose value depend on the object, the location inside it and the instant, such as toxicity level in a pollution cloud. The value exist only inside the object, and if objects overlap, each point inside the intersection will have a value for each object.

The approach chosen in MADS supports both kinds of continuous fields. Space-varying attributes can be defined on the space extent of DBSPACE, while the value can be from any domain of values of a SADT.

Variation, either spatial or temporal, is usually done according either to a continuous, stepwise, or discrete function. Each type of function is supported by MADS and their types define the kind of interpolation, if any, that can be used to compute the value of the attribute. Space-varying attributes are represented in schema diagrams using the notation *f(spatial extent icon)*.

With respect to time, attributes can be roughly categorised as:

Static attributes: Attributes that have a value which does not depend on time. The value once entered in the database is not expected to change, a date of birth being a good example.

Time-varying attributes: The value of such an attribute may change over time, but the database only keeps track of the latest value. No visual notation is used on the diagram to differentiate static and time-varying attributes.

Timestamped attributes: These attributes have time-varying values and the database keeps track of their evolutions over time. The term *evolution* denotes that past, present, and future values may be recorded. Timestamped attribute are represented in schema diagrams using the notation *f(temporal icon)*.

Time-varying attributes can be defined on the time extent of DBTIME while the value can be from any domain of values of a TADT.

Finally, information may vary both in space and time. The principle of orthogonality allow this model to perfectly represent such cases, as for example mean of rainfall values for a year over a country.

To enforce orthogonality, MADS allows, in extension to attributes, to describe object or relationship types as spatial or temporal. The spatiality of an object or relationship type is described by a predefined attribute, *Geometry*, whose domain is a SADT. In diagrams, spatial object/relationship types are denoted by showing the icon of the corresponding SADT along with the name of the type.

Similarly, object and relationship types may be timestamped. This allows to keep track of the life cycle of their instances from creation to deletion, with activations and suspensions in between. Here also a predefined attribute is used, *LifeCycle*, which describes the temporal behaviour of instances. MADS, as opposed to other simpler models, allows object and relationship types to live more than a single continuous span. To realise that, the *LifeCycle* is a function from DBTIME to STATUS, where STATUS is a predefined domain made up of four values: not-yet-existing, active, suspended, and disabled. The first one concerns an object known to exist at a later time. Active objects can be accessed and modified, in the actual time frame, while suspended only allow properties to be accessed but not changed. Finally, disabled concerns objects which existed but are not accessible anymore within the disabled time frame. Only a subset of all possible transitions is allowed; they are: not-yet-existing $\rightarrow$ active, active $\longleftrightarrow$ suspended, active $\rightarrow$ disabled and suspended $\rightarrow$ disabled. Temporal object or relationship types are denoted in the schema diagram by showing the icon of the TADT on the left side of their name.

Spatio-temporal types are types whose geometry is timestamped. They are denoted in a schema diagram by putting the corresponding TADT icon on the right of the SADT one.

3.2.3 Space, time and generalisation links

The use of generalisation links is possible between spatial and non-spatial object and relationship types. A non-spatial type can have a spatial subtype. Conversely, a spatial type only has spatial subtypes.

The same applies to temporality. However, a precision is needed for a non-temporal super-type having a temporal subtype. In this case, the inherited information of the subtype is going to disappear from the database if it gets suspended. This would cause a problem when moving back to active, as the information would have to be entered once again. To deal with that, the information coming from the super-type is kept in the

subtype when the supertype is removed from the database. With respect to generalisation/specialisation, object or relationship timestamping makes it necessary to introduce a differentiation between two structures. Let us define *cluster* as the set of subtypes specialising a super-type according to a given criterion. This cluster is called *static* if a sub-type cannot change of category, it is *dynamic* otherwise. In a static cluster where the super-type is covered by the subtypes, the life cycle of an object is the same in the sub and supertypes. For a dynamic cluster, a redefinition of the timestamp is needed for keeping track of specific subtypes life cycles. In Figure 3.3 the various cases are shown, static cluster in (a), dynamic in (b) and (c) representing a temporal subtype with a non temporal super-type.

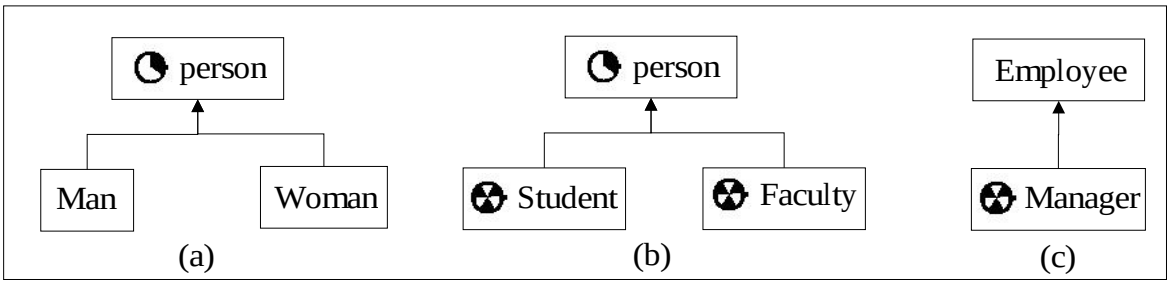

Figure 3.3: Timestamped objects and generalisation.

Several types of inheritance are defined in MADS. They copy the traditional mechanisms of the object-oriented paradigm: refinement, redefinition, and overloading. In the following description, the term *property* denotes either attributes or methods. Examples are given in Figure 3.4.

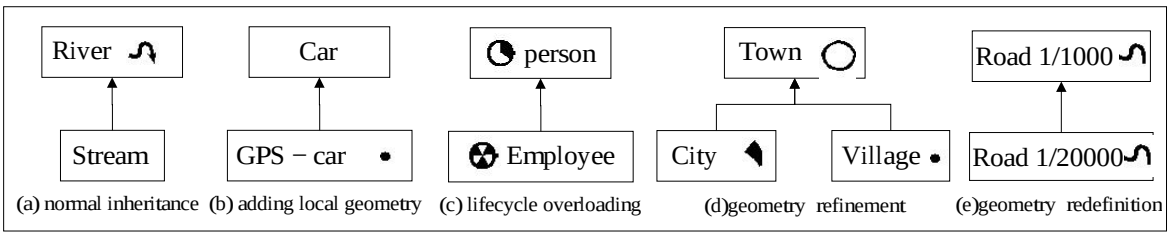

Figure 3.4: Inheritance, refinement, redefinition and overloading.

Refining is useful when a property has a smaller domain in the subtype than in the super-type. *Redefining* or *overloading* an inherited attribute aims at a different objective. Here a new property is created in the subtype, with the same name. This allows to keep track of both properties, one inherited and one local. It makes association of multiple

geometries or life cycles to the same object possible. These two methods differ in one aspects. The domain of values of a redefined property must be a subtype of the one in the super-type, thus allowing dynamic binding. For overloading, the domain of values is not restricted and thus it prevents dynamic binding.

The problem of *multiple inheritance*, which arises when a subtype has multiple supertypes and they each have a property with the same name, is solved in MADS by using fully qualified naming. This means that reference to the property is made by calling: `SupertypeName.PropertyName`.

Finally, since MADS support generalisation links between relationship types, the behaviour of roles with respect to inheritance must be defined. This is analogous to the behaviour of attributes, as described above.

3.2.4 Constraints

Since spatial and temporal features can be over objects, relationships, and attributes, issues of consistency arise. Space does not automatically generates constraints, a spatial object may have attributes, spatial or not, and if they are, their spatiality may not be linked to that of the object. On the other hand, the definition of spatial integrity constraints should be easy. Time, instead, is seen as prone to generate constraints. We describe next how classical constraints are revisited to take temporality and spatiality into account. Then specific constraints for spatial and temporal entities will be discussed.

Cardinalities and identifiers

In MADS, the traditional cardinality notion is completed with the specification of a temporal cardinality. This allow to constrain the evolution of the values over time. A temporal cardinality written *h(min,max)*, defines a constraint on the minimum and maximum number of values that an attribute can take over the associated life cycle. If no timestamping is defined, then a infinite life cycle is assumed. Unfortunately, such a definition has two possible interpretations:

- The number of changes is constrained. Enforcement of the constraint can be assured by triggering an event when the maximum number of updates is reached.
- The number of values is constrained. The ensemble of values has a maximum size, however the number of changes inside this ensemble is not limited. To enforce this, triggering on updates is again possible.

A similar case arises for role cardinalities in relationship types. Here the temporal cardinality defines the minimum and maximum number of relationships that an object can participate into during its life cycle. The interpretation of the temporal cardinality leads to the same ambiguity than on attributes. But the fact that relationships have an OID may lead to different policies.

These two issues require a specific investigation to assess user understanding and preferred requirements. It is left for further analysis.

To prevent dangling references, e.g., a relationship linking nonexistent objects, MADS enforces the policy that whenever the instance of an object is deleted, the instances of relationships defined on the objects are deleted too, even in the case of a timestamped relationship. With the use of this policy, no dangling references can be created, and thus MADS has no implicit constraints on object/relationship timestamping.

MADS allows timestamped relationships to link non-timestamped object types. In this case, the meaning is that the relation keeps track of all past, present, and future instances concerning currently-valid objects. Also, non-timestamped relationship types may link timestamped object types, allowing to keep track of all past, present, and future objects and only currently-valid relationships.

Similarly, the notion of identifier must be revisited for timestamped object and relationship types. If a timestamped object has an identifier, it can be interpreted in two ways:

- the identifier designates a single object at any point in time, but may designate multiple objects at different point in time, it is then called *temporal identifier*, or
- the identifier will only ever be associated with one object, it is then called *static identifier*.

MADS adopts the temporal identifier by default, but a facility allows the user to choose between the two.

Spatial constraints

Several spatial constraints may be associated to attributes, object, and relationship types. As example, two categories are given:

- spatiality in a set of related objects will lead to constraints, such as no overlapping or adjacency.

- spatiality of an attribute may be constrained by the spatiality of the object (or relationship) type it belongs to. For example, the capitol of a city must be spatially contained in the city.

In order to enforce and verify these constraints, the notions of envelope and interior are very useful. They are implemented as methods on the predefined spatial types. While these two notions are easily defined on simple geo, and its subtypes, the case of complex geo is much more complex. MADS extended the notion to the complex area, but the work for general composite objects remains to be studied.

Temporal constraints

In MADS, no a-priori constraints are enforced on timestamps. But they may be defined either, explicitly, using temporal expressions based on a calculus that includes Allen's operators, or, implicitly, by referring to predefined constraint types, such as inclusion, covering, and equality. An *inclusion* on an attribute, for example, would enforce that its temporal values be included in the temporal values of the object it is member of. A *covering* would mean, within the same example, that at any instant of the life cycle of the temporal object a value exists for the attribute. Finally, an *equality* constraints on two attributes states that the union of the temporal elements of both are equal.

The application of temporality on is-a links redefines the inclusion constraints. If both object (or relationship) types are temporal, then the temporal values must be the same. If the supertype is temporal, its instances must be active when the subtype instances exist. If only the subtype is temporal, then for the instance to be active the supertype instance must exist.

Role cardinalities also need to be redefined. For temporal object types in non-temporal relationship types, refer to the part about cardinalities and identifiers in Section 3.2.4. In the case of a temporal relationship type linking non-temporal object types, MADS defines the temporal cardinality as applying to the active instances of the relation itself. To deal with temporal object and relationship types together, the two previous cases are combined.

The last issue is the coexistence of temporal and non-temporal facts. If a temporal attribute is defined on a non-temporal relationship type, for example, then its history of values is lost when the relationship is deleted. For a temporal object type with non-temporal attributes, when making a retroactive query, the current value of the attribute is returned but no value is returned for proactive queries.

3.3 Multiple representation support

After showing how the MurMur data model answers the needs of spatiality and temporality in Section 3.2, this section concerns the way the data model answers the need of multi-representation. As multiple representation was discussed in Section 2.2, the discussion starts immediately with the data model perspective.

Multi-representation in MADS is supported in two different ways. The first one, stamping, defines multi-represented object/relationship types, attributes and links. This approach, discussed in Section 3.3.1, may be appropriate whenever the different representations of a real-world phenomenon are close enough to each other so that they can be integrated into a unique, parameterised definition. The second way, discussed in Section 3.3.2, is based on a new kind of relationship types, *correspondence relationships*. They allow to independently define representations and nevertheless relate them together. This approach is to be used when representations of the same phenomenon are so diverse that they cannot be integrated in a common definitional framework.

The notion of multi-representation, linked to viewpoints and resolutions means that querying the database is always done with a particular viewpoint and resolution, or a set of them. This will be referred as the query stamp.

3.3.1 Multi-representation stamps

These stamps are used to attach to specific schema elements a set of viewpoints and resolutions. We first discuss stamps on object types and attributes, and then stamps on relationship types. discussed separately.

Stamps on object types and attributes

Stamping object types allows associating several representations to the same real-world phenomenon. Different viewpoints and resolutions are used to map these representations. All characteristics of object types are subject to multi-representation: the global visibility as well as any attributes and values. In particular, the geometry of an object is frequently multi-represented, according to different spatial resolutions. The life cycle of an object also supports representation stamping. The stamping of an attribute concerns all its characteristics such as visibility, value, value domain, cardinalities, timestamping, as well as space and time variability.

Stamping is used to separate multi-represented object types from the ones that provide a real-world representation independent from viewpoint and resolution. Such object types,

their instances and attributes should always be visible, whatever the query stamp is. But their visibility can still be restricted as will be explained later.

Stamps on object types can be restricted at the instance level, to limit visibility of an instance to a subset of the object type's visibility. This is useful for situation where multiple representations share the same description, but not the same instances. An instance must be visible for at least one of the stamps of its object type.

Similar visibility restrictions can be applied to attributes. In the absence of restrictions at the attribute level, stamps defined at the object level extend over all attributes, providing thus the same visibility. This is also valid for instance stamps at the object level, thus replacing the object type stamp. These rules simply state that an attribute cannot be visible when its object type is not. Thus when stamping an attribute, the stamp must be included in the set of stamps of the object type. When restrictions exist both at the attribute level and at the instance level, the domain of visibility is the intersection of the domains of these two restrictions.

Visibility restrictions can be independently defined on several attributes within the same object type. However, the set of restrictions must preserve consistency of the data structure of the object type definition for each stamp. Consistency here means that "the attribute tree built by pruning according to the stamp filter must be a sub-tree of the original integrated attribute tree (with the object type as root) such that: for every attribute in the sub-tree its parent attribute in the original tree must also be in the sub-tree, and for every complex attribute in the original tree that appears in the sub-tree at least one of its original component attributes must be in the sub-tree" [20].

Sometimes an attribute having multiple stamps may hold different values, one for each stamp. The exact behaviour is precised with another stamp, the *rstamp*, which defines if values vary or not according to representations.

For complex attributes, stamp restriction can be performed at all levels. The restriction of a complex attribute applies to all its components. They can also restrict this implicit stamp by keeping a subset of it.

When considering multiple representations and generalisation/specialisation hierarchies together, the question of whether stamps are inherited or not arise, thus making it a kind of visibility property. Assuming that stamping is part of the attribute specification, it follows that a subtype inherits the stamps from the super-type. This is explained by the fact that since the stamp of an attribute must be part of the object's stamps, and attributes are inherited, with all their characteristics, then the object necessarily inherits the stamps of the supertype. However, as with attributes, the visibility might be restricted to a subset of the original visibility of the supertype. Here again, consistency problems arise. A subset

of object types for any given stamp must form a properly defined is-a hierarchy. If this cannot be done with stamping then correspondence links should be used, as explained in Section 3.3.2.

Stamps on relationship types

In this section the particularities of stamping relationship types are discussed. The rules for stamping and for restricting these stamps are the same as for object types. Attributes bear the stamps of the relationship type, as is the case for attributes of object types. An instance can also restrict the visibility, but must be visible at least for one of the stamps.

However, by definition a relationship type is a link between at least two object types. This means that to see the link both objects must be visible. Thus, in order to access a relationship type, and some of its objects, two at least, a query must have a stamp that holds for the relationship and a stamp per linked object. Of course these stamps may be the same.

Particular relationship types have specific behaviour. Topological relationship types are used to link spatial object types. Therefore, stamps on these relationship types depend on the stamps of the *Geometry* attribute of the object types they link. Spatially-constrained relationship types can be stamped with one or more stamp:

- **No stamp:** the relationship type is always visible.
- **A single stamp:** the *Geometry* attributes of linked object types must have this stamp.
- **Several stamps:** the *Geometry* attributes of linked object types must be visible in the intersection. To link object types having disjoint stamps on their *Geometry* attribute, a relationship type must be stamped with the conjunction of these stamps.

The criteria defining topological criteria for relationship types also bear the stamp of the relationship. In this way multiple criteria may be defined on topological relationship types. When an instance of such a relation has restricted its implicit stamp, and the criteria are also stamped, only the criteria corresponding to the stamps of the instance are applied.

Aggregation relationship types may also be stamped with one or several stamps. When the aggregation is defined between objects having disjoint stamps, it is a correspondence relationship type, as defined in the next section.

3.3.2 Correspondence relationship types

Correspondence relationship type relate multiple representations of the same real-world phenomenon. These relationships can be of different kind and link more than two representations. Correspondence relationship types are not representation stamped. The instances they link have different stamps. As with other relationship types, to see a correspondence relation, the query stamp must hold at least a stamp for each of the two linked instances.

Aggregation relationship type

They are binary relationship types, allowing the description of composition relations between concepts of different viewpoints and different resolutions. In cartographic generalisation, for example, spatial simplification is often achieved by the aggregation of objects. Different criteria might be considered for aggregation:

- **Semantic criteria:** for example, cultivated zones are aggregated because they have the same land use.
- **Spatial criteria:** for example, buildings near each other are aggregated as part of the same neighbourhood.
- **Semantic and spatial criteria:** for example, a collection of buildings and sport grounds are aggregated into a university object.

An example of spatial and semantic aggregation is shown in Figure 3.5.

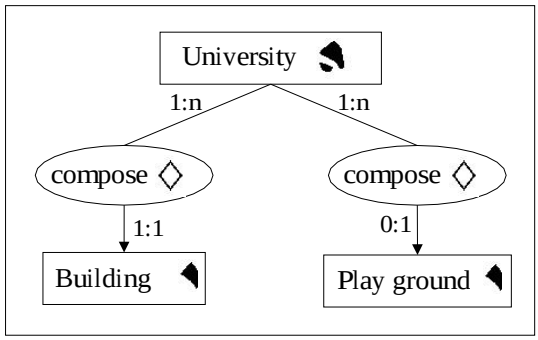

Figure 3.5: Example of aggregation relationship type.

SetToSet relationship type

The semantic of such a relationship is that a group of objects correspond to another group of objects. Traditional relationship types do not capture this kind of correspondence because they only allow to link one instance of each participating object type. An additional cardinality is specified for roles of SetToSet relationship types, defining how many object instances the role may link to one relationship instance.

Cartographic generalisation frequently leads to this kind of correspondence. Typification is a spatial simplification operation that computes one or several objects from a set of objects, such that the geometry looks similar to the original set, but not necessarily with a 1:1 mapping. Object resulting from typification have no real-world counterpart: the important information is more the spatial configuration, the shape than the exact position of objects.

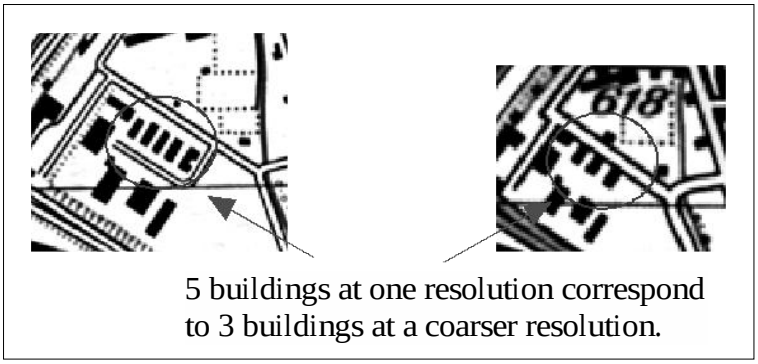

Figure 3.6: An example of SetToSet correspondence.

Figure 3.6 is an example of such a cartographic typification. Figure 3.7 shows a possible modelisation of the example, with an instance declaration.

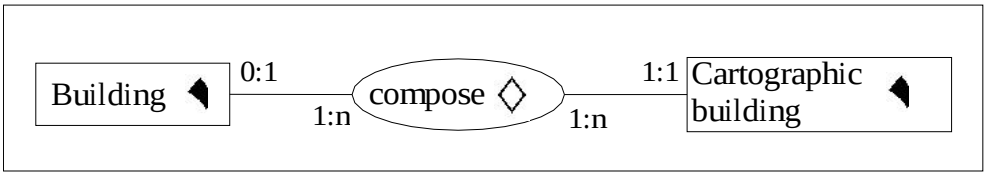

Figure 3.7: An example of a SetToSet relationship.

Identity relationship type

It is a binary relation linking instances with the same OID. It has a special semantics: the linked instances are two representations of the same real-world entity.

Cardinalities on the roles of the identity relationship type indicate the correspondence of the populations of the object types:

- **0:1-0:1:** the populations intersect.
- **0:1-1:1:** the population of one object is included in the population of the other.
- **1:1-1:1:** the populations are identical.

The identity relationship type offers an alternative to the description of multi-represented object types using representation stamps.

With the cardinality of 0:1-1:1, the semantic of an identity relation is the same as with the is-a link, inclusion. For 0:1-0:1, the semantic is the same as with maybe links, intersection. However, several reasons made the definition of identity relationship types necessary:

- The is-a and maybe links do not allow definition of attributes and methods. However, correspondence relationships may need such attributes and methods definitions.
- No link allows to define equivalence of populations.
- The is-a link is not suited to describe correspondence relation due to its inheritance and dynamic binding behaviour. This is discussed below.

The classical generalisation link is inadequate for representing different viewpoints and resolutions. If we consider representing objects at different resolutions, obtained by cartographic generalisation, some objects would no longer be represented because they are below the resolution threshold. With an is-a link to represent correspondence, the population inclusion constraint implies to choose as super-type the most precise resolution, and the least precise as the subtype. However, in cartographic generalisation, some features are lost, those that are not relevant for the new resolution. The subtype in this case would then possess fewer attributes than the super-type and thus the inheritance of attributes is not suited for this case.

Finally, MADS represent spatial information through an attribute, *Geometry*. In order to describe an object with a different spatiality, the geometry attribute would have to be redefined in the subtype. Because of the dynamic binding mechanism in is-a links, the value returned by default would the one of the subtype. This is not always relevant.

Hierarchy of correspondence relationship types

Correspondences between two object types and between their instances may not always be described with a single relationship type. To be able to convey those cases, the correspondence relationship types are organised into a generalisation/specialisation hierarchy. This hierarchy allows the definition of generic relationships spanning over multiple correspondences between object types. This hierarchy is designed to allow relationship types in an inheritance hierarchy to change kind, something impossible otherwise.

The classification criteria used to construct the hierarchy are:

- object cardinality,
- relationship cardinality,
- differentiated roles, indicating whether roles are differentiated, such as in an inclusion, where the roles would be *includes* and *is included in*,
- same OID, and
- arity, indicating whether the relationship can posses any number of roles or not.

The obtained hierarchy is presented in Figure 3.8

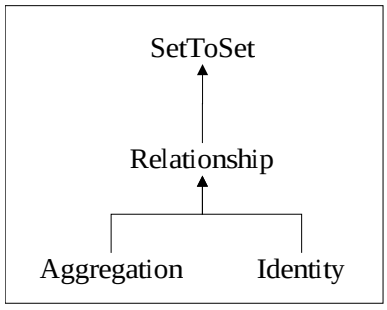

Figure 3.8: Hierarchy of correspondence relationship types.

Hierarchical constraints The link between a relationship type and a subtype is the classic generalisation/specialisation link with inheritance, possible redeclaration of attributes and methods, and dynamic binding. In this context, dynamic binding will allow direct access to the most specific relationship type defined for each object at the time the relation is manipulated.

In addition to attributes and methods, roles can also be inherited or redefined as follow:

- The cardinality of a role can be redefined, with the restriction that the redefined cardinality interval must be inside the inherited one.
- The object type linked may be redefined, provided it is a subtype of the original object type.
- A new role may be added.

However allowing relationship subtypes of a different kind from the one of their supertype implies the definition of additional constraints:

1. The definition of an aggregation or an identity as a subtype requires that the supertype is binary.
2. The definition of an identity relationship as subtype of another relationship type implicitly redefines the cardinality of roles as either (1:1-0:1), (0:1-0:1), or (1:1-1:1). This is due to the fact that two objects with the same OID cannot coexist in the same object type.

The constraints on cardinalities apply at different levels:

- at the instance level, indicating into how many relationship instances one instance of the object type can participate.
- at the population level, inducing set relationships, as defined before.

Defining an identity as a relationship subtype of SetToSet relationship, aggregation, or association, constraints the redefinition of cardinalities and object linked with the following rules:

3. If only the cardinalities of the roles are redefined, then they can only be redefined as (0:1-0:1).
4. If the cardinalities of the object type linked are redefined, then they need to obey rule 2.

To describe multiple correspondence types between object types, the relationship hierarchy can be used in different ways:

1. The object types having multiple correspondence types are divided into subtypes having homogeneous correspondence type. The correspondence relationship type defined between the two super-types is then redefined between each corresponding subtypes.

2. A correspondence relationship hierarchy is defined containing a relationship for each possible correspondence type.

3. Only the most general correspondence relationship type is defined.

These three approaches are not equivalent in terms of their expressive power. They are given in order of decreasing power. However, a schema using the first method would be much more complex to read than with the last one. Decision on the model to use is left to the user, MADS schema supports the three of them.

Information associated with correspondence relationship types These relationships allow to relate multiple instances representing the same real-world phenomenon. To allow evolution of the database and propagation of changes between representations, supplementary information must be associated with each correspondence relationship type. This information can be of three type.

- **Correspondence between populations:** The aim is to be able, when adding or modifying an instance to an object type, to determine if this instance must be part of the linked instances. The specification can be made on different criterion:
  - Spatial, e.g., parking with an area <1000 m^2.
  - Semantic, e.g., rivers that are polluted with quicksilver.
  - Topologic, e.g., buildings next to a night club.
  - Temporal, e.g., buildings owned by the city in 1995.
- **Correspondence between sets of linked instances:** The aim is to specify correspondence conditions between sets of linked instances. A logic predicate is used to specify the corresponding rule.

- **Correspondence between attribute values:** To maintain integrity of the database, it is important to have mechanisms to control the attributes describing the same information. This can be done *a priori* by defining derived attributes, or *a posteriori* by using integrity constraints.

3.4 Conclusion

This chapter describes the MADS conceptual model. Three important characteristics of the model are resumed here:

1. The model is conceptual, it has been defined as a tool for information modelling free of implementation concerns. However, the definition of the MADS model was made targeting translation toward specific DBMS.

2. The model is orthogonal. The independence of the features makes them freely combinable. This allows maximal expression power and fine-grain modelling.

Chapter 4

Implementation of the data structure

Contents

4.1 Proof of concept 46

4.1.1 Independent data structure 46

4.1.2 Example of schema editing 47

4.1.3 Validation 50

4.2 General analysis 55

4.3 Functional analysis 56

4.3.1 The schema and other elements 56

4.3.2 The domains 59

4.4 Implementation 60

4.4.1 Package `mads.tstructure.core` 61

4.4.2 Package `mads.tstructure.domains` 66

4.4.3 Package `mads.tstructure.utils` 68

4.4.4 Package `mads.tstructure.utils.exceptions` 68

This chapter describes the implementation of the data structure created to answer the needs of the schema editor, described in [21], and those of MADS's syntax, given in Appendix A. First, Section 4.1 describes the importance of the data structure. The following sections, 4.2, 4.3, and 4.4 are the adapted reproduction of the text that was written for the European Commission in [22].

Section 4.1 is organised as a proof of concept justifying the work done on the structure. Some of the information developed in this section will be better understood after reading the analysis of the implementation process. However, in order to provide for the reader an interesting point of view, justification is given before tedious analysis details.

Section 4.2 analyses the creation of the data structure, independent from the graphical interface, and proposes a first sub division of the problem. It also includes a general description of the impact of multi representation and of the syntactic constraints on the structure itself. Then a more detailed analysis is given in Section 4.3, taking every schema element into account and precising the design choices made for each one. Finally, the implementation details are given last, in Section 4.4, and cover the way the language chosen, Java, models the realization of the data structure.

As for code documentation, the language used provide a tool, *javadoc*, which produces HTML documentation from the comments included in the code. This documentation can be found on the CD-ROM described in Appendix B.

4.1 Proof of concept

The goal of this section is to show why the data structure, as it is implemented, is a basic construct of the schema editor, and of future applications. This will be discussed in Section 4.1.1. Then, an example of schema editing, centred around the role of the structure, will be given in Section 4.1.2. At last, validation of a schema will be described in detail in Section 4.1.3. As the only action completely performed inside the structure, independently from any application using it, its role is critical.

4.1.1 Independent data structure

At first, the question of "Why implement a data structure?" may arise. A syntax is developed already, and thus provides a clear structure of a MurMur schema organisation. While this syntax is sufficient to produce a tool checking for syntactic correctness in a produced MurMur schema, it is a static construct. This means that the schema has to be completely written before it can be checked. The goal with an implemented data structure is to allow dynamic checking throughout the whole creation of a schema. In order to realise that, an integration with the applications is targeted. This means that the implemented data structure will use the object-oriented model and the same language.

Now that the usefulness of implementing a data structure has been shown, one may ask "Why an independent data structure ?". With only the schema editor in mind, to create

the structure inside the editor application would seem, first, much easier and, second, much more efficient in terms of programming and at run-time. However, as explained in Chapter 3, MurMur's objectives are more than just a schema editor tool, which would give a very short lifespan to the model. As they are defined now, the next tools considered in the project will need to work based on existing schemas. While the notion of schema is frequently associated to a visual representation for users, computers do not care about the visual part. As such, building applications on top of a data structure, a programmed one, instead of on top of another application makes perfect sense. First in terms of complexity, dealing with logical components is easier than with graphical ones. Second in terms of efficiency, as running one application on top of another would be very demanding for the computer, especially with the weight of graphical Java applications.

4.1.2 Example of schema editing

In order to design schema, MurMur provides a schema editing tool. Description of a simple schema realisation is done. The focus is on the structure interaction, not on designing a perfect schema. The example is limited due to the schema editor being in its early stage of development.

First, an object is created as shown in Figure 4.1. The first step is to click on the object creation icon (1), then to click in the drawing window (2), thus effectively creating an object. The name of the object may then be set (3). During this operation, the structure proceeds by creating an object, registering it inside the schema and then sets its name. A first check is performed at that stage, to ensure that the chosen name does not duplicate an existing one. A second object is then created following the same procedure. Then a relationship is created with the same operations, excepted that the icon used to draw is the relationship one, on the left of the object type.

Next, properties of the two objects are updated, making them spatial. A right or double click on the object opens the property window. In this window, the second tab is used to specify the spatiality. Behind the scene, the structure creates a new attribute, with the predefined name *Geometry* containing the necessary information. The property window and its spatiality tab can be seen in Figure 4.2.

The next step is the creation of roles to link the two object types to the relationship. This is done by right clicking on the roles entry of the relationship. This pops up an *add role* menu and then a role window is opened, allowing to choose the related object. This can be seen in Figure 4.3. After the role is confirmed, it is created inside the structure. First only object types are proposed as link, thus preventing a role from linking two relationships.

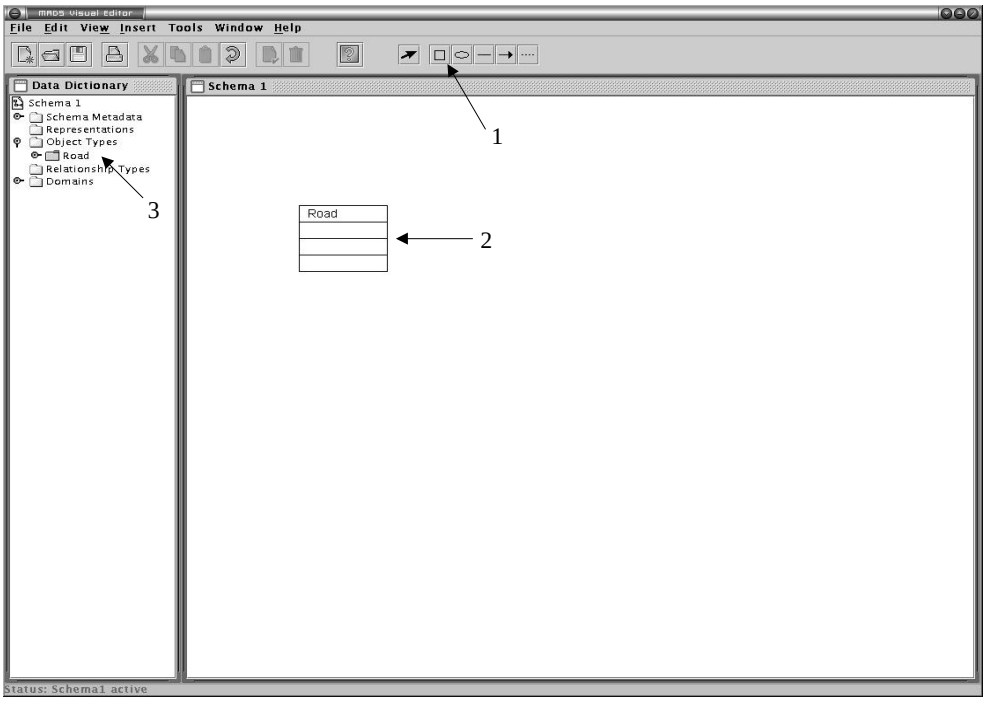

Figure 4.1: Schema editing: object creation

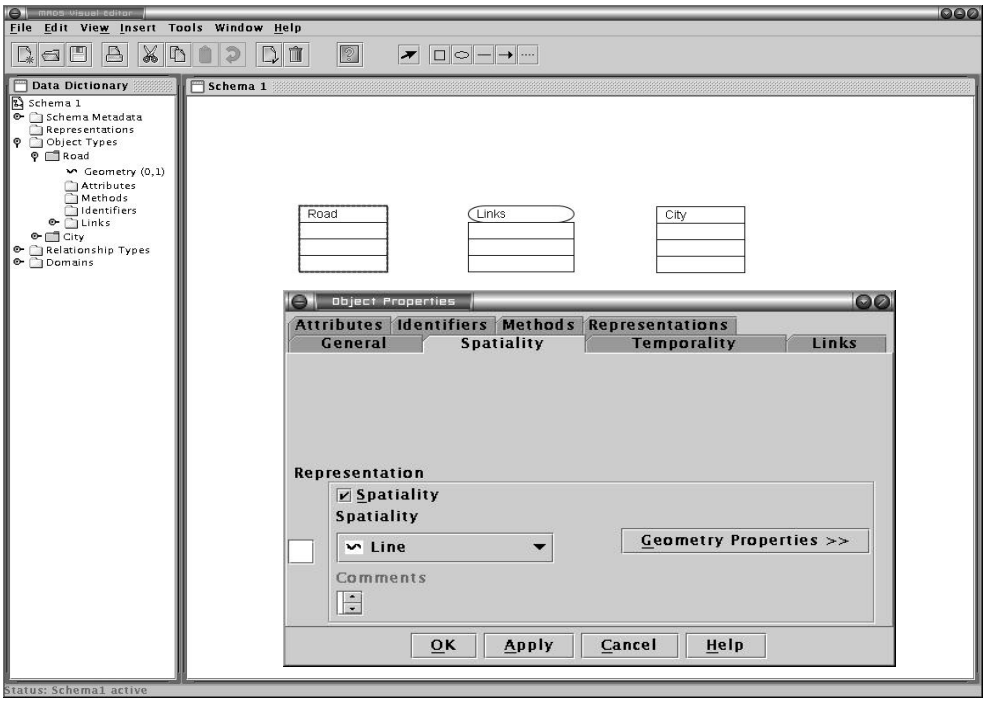

Figure 4.2: Schema editing: object properties

Then the role is created inside the relationship and is registered also in the object type. A cardinality is immediately attached to the role.

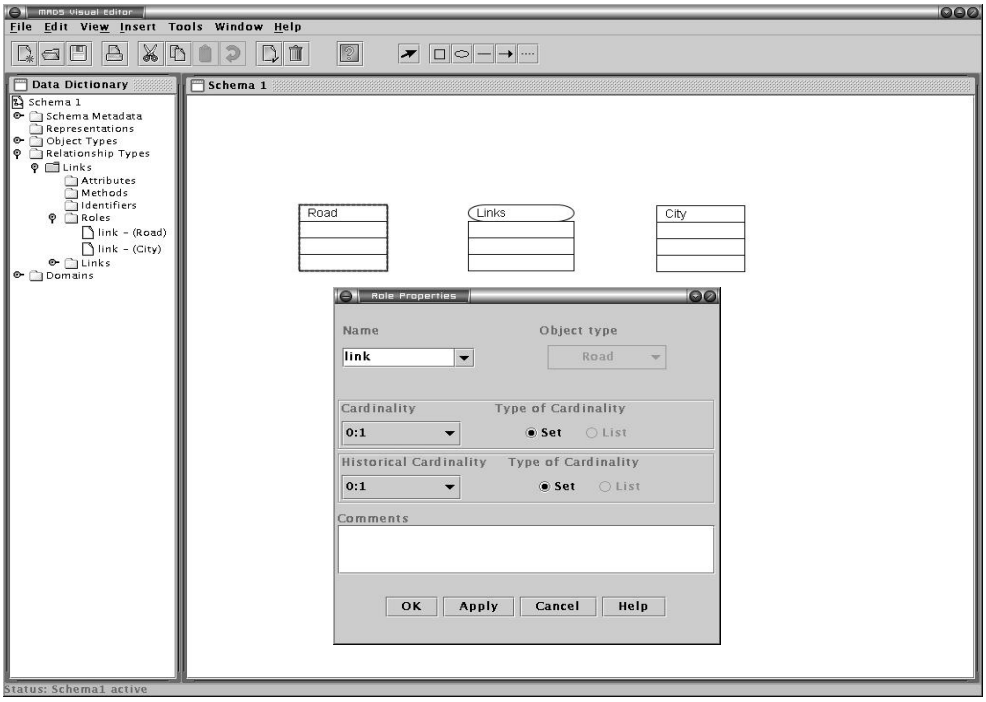

Figure 4.3: Schema editing: role properties

The last operation that will be done on the schema is creation of an attribute. This is done by right clicking in the tree on *attributes* under the chosen object. This pops up the attribute property window. This is shown in Figure 4.4. The structure adds the attribute inside the object type, giving it a name. This name must be unique inside the object type. When the attribute is defined, a cardinality is attached to it, as within the role creation.

This concludes the example. The goal was to show how the structure is integrated to the editor. Its is a storage for all schema information. It is either seen on screen, or through the various dialog windows.

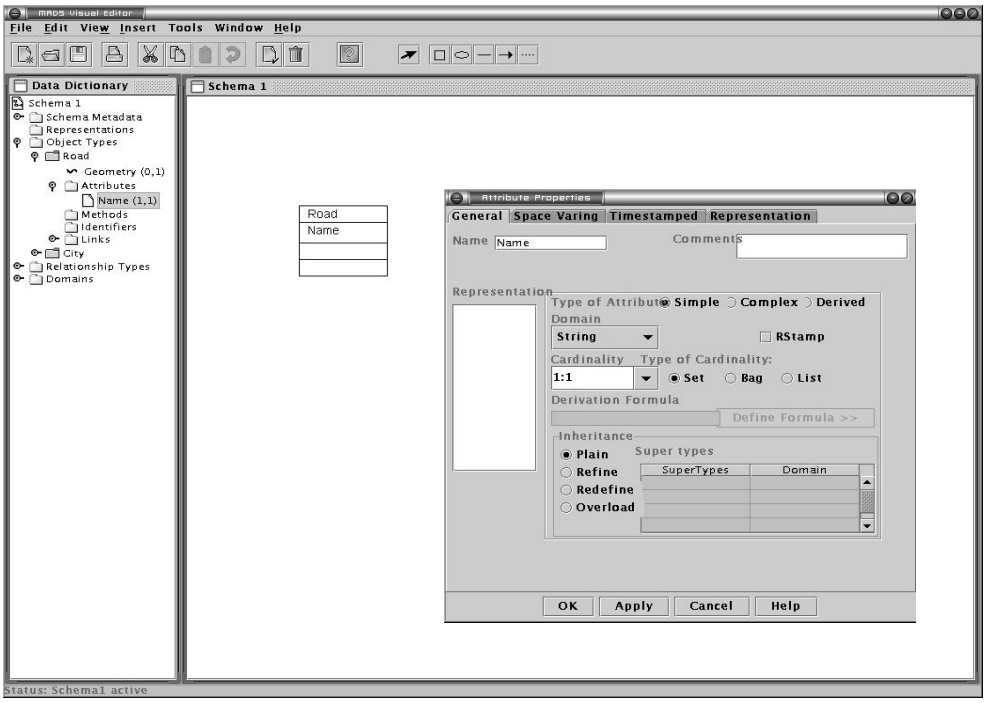

Figure 4.4: Schema editing: attribute properties

4.1.3 Validation

The validation process is very important, it ensure that a stored schema respect the syntax of the MurMur data model. Validation is done, inside the editor, before exporting and before saving a schema. More details on the design choices of its implementation are given in the following sections. Here, attention is on the sequence of actions of validation.

In order to validate a schema, an option exist in the tool menu. When validation is called it is always on the schema element. This element has a validate method with a public access. The sequence diagram of schema validation is shown in Figure 4.5 and Figure 4.6.

On these schema, a white box indicates the lifespan of a method in an instance and a grayed box indicates branching execution paths. The test to determine if the branch is taken or not is indicated on the schema. When two grayed boxes are linked by an arrow, this indicates that the execution path will take only one of the available branches. Arrows indicates a method call, the name of the method is indicated on the arrow, followed by its return type.

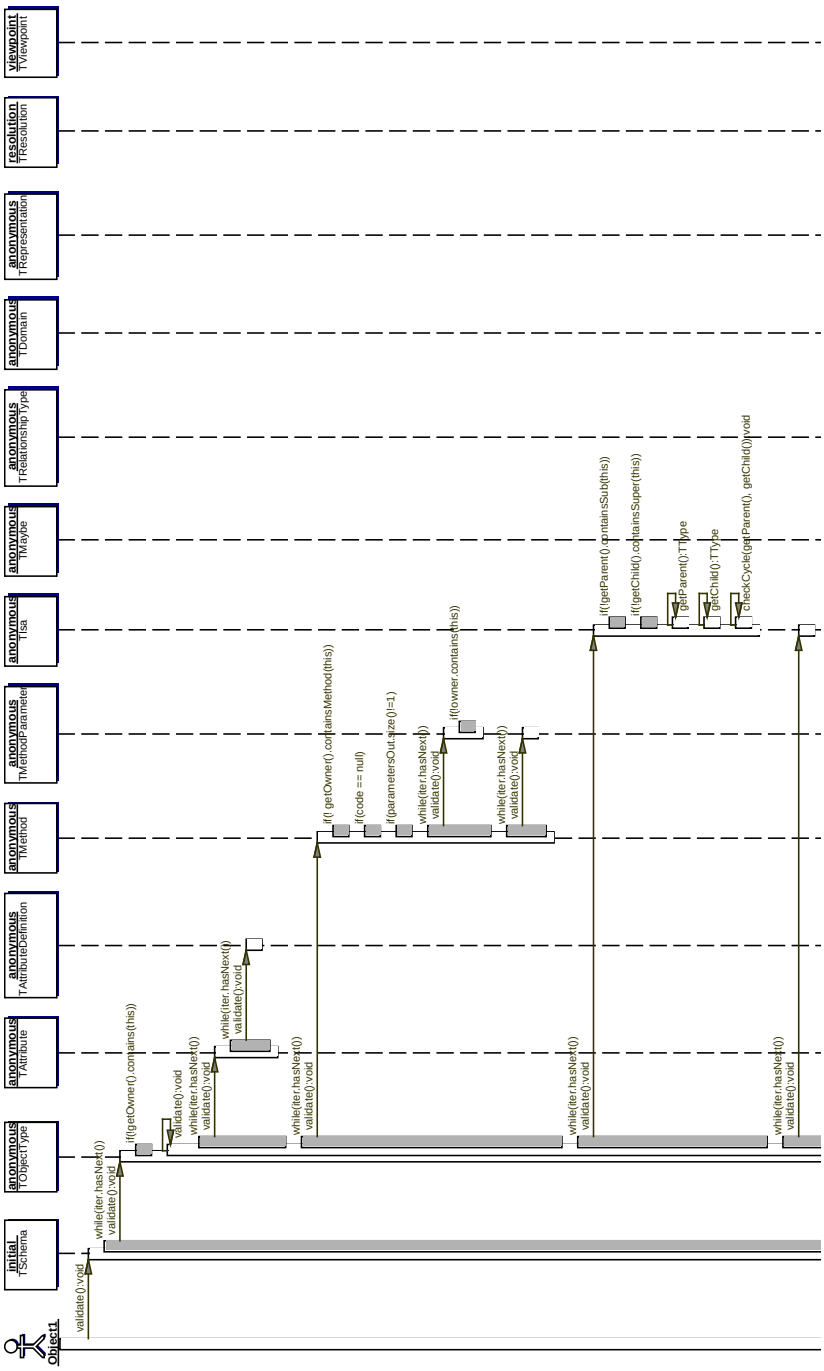

Figure 4.5: Sequence diagram of `validate` in a schema (1)

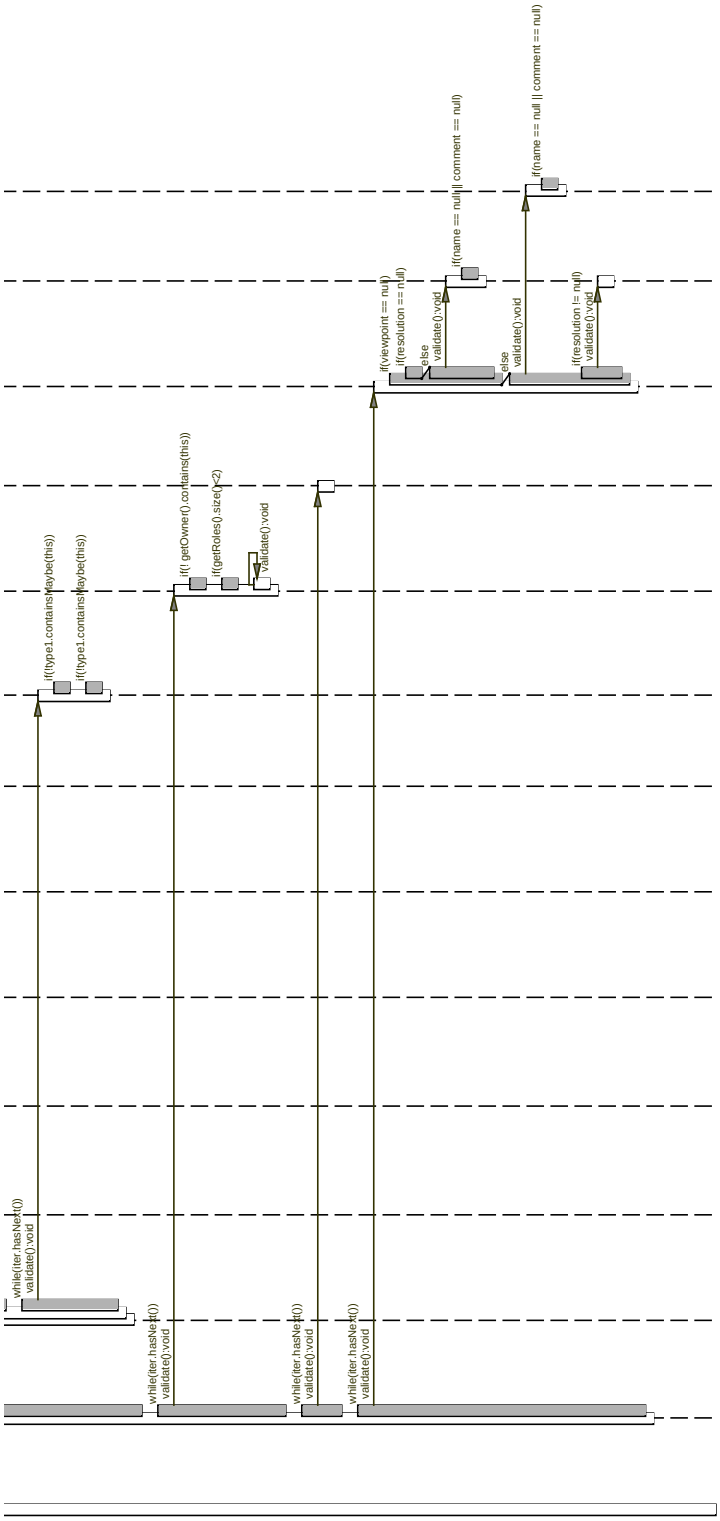

Figure 4.6: Sequence diagram of `validate` in a schema (2)

The validation proceeds top-down, starting from the schema components. First, object types are validated. In order to do that, the `validate` method is called in the object type class. It first checks if the object is correctly registered inside the schema. Then the object tests if its attributes and methods are valid. In order to be valid, an attribute checks that its definitions are valid. For methods, checks are performed on their parameters. Detail of the tests performed to ensure validation of properties can be found on the diagrams.

After properties, links are validated, it concerns here only is-a and maybe links. The roles will be checked during relationship types validation. For is-a links, the performed checks include ensuring that both linked types still exist and that no the inheritance hierarchy does not cycle. For maybe links, existence of the linked types is checked.

After objects, relationship types are validated. The behaviour is close to the one of object types. A specific validation, for a topological relationship can be found in Figure 4.7.

The cross relationship type is binary topological relationships having predefined role names. First, check is made that the roles use the correct names. Then validation checks that no more than two roles are defined, and that they link spatial object types. Finally the last check on roles is that at least two are defined. This organisation of roles validation is due to the way relationships are organised in classes. This is explained in Section 4.4.

The attribute and methods validation works exactly in the same way as for object types, the same for is-a and maybe links.

Going back to the schema validation, it is achieved by checking that representations are valid.

The way an invalid element is reported is describe in Section 4.4.

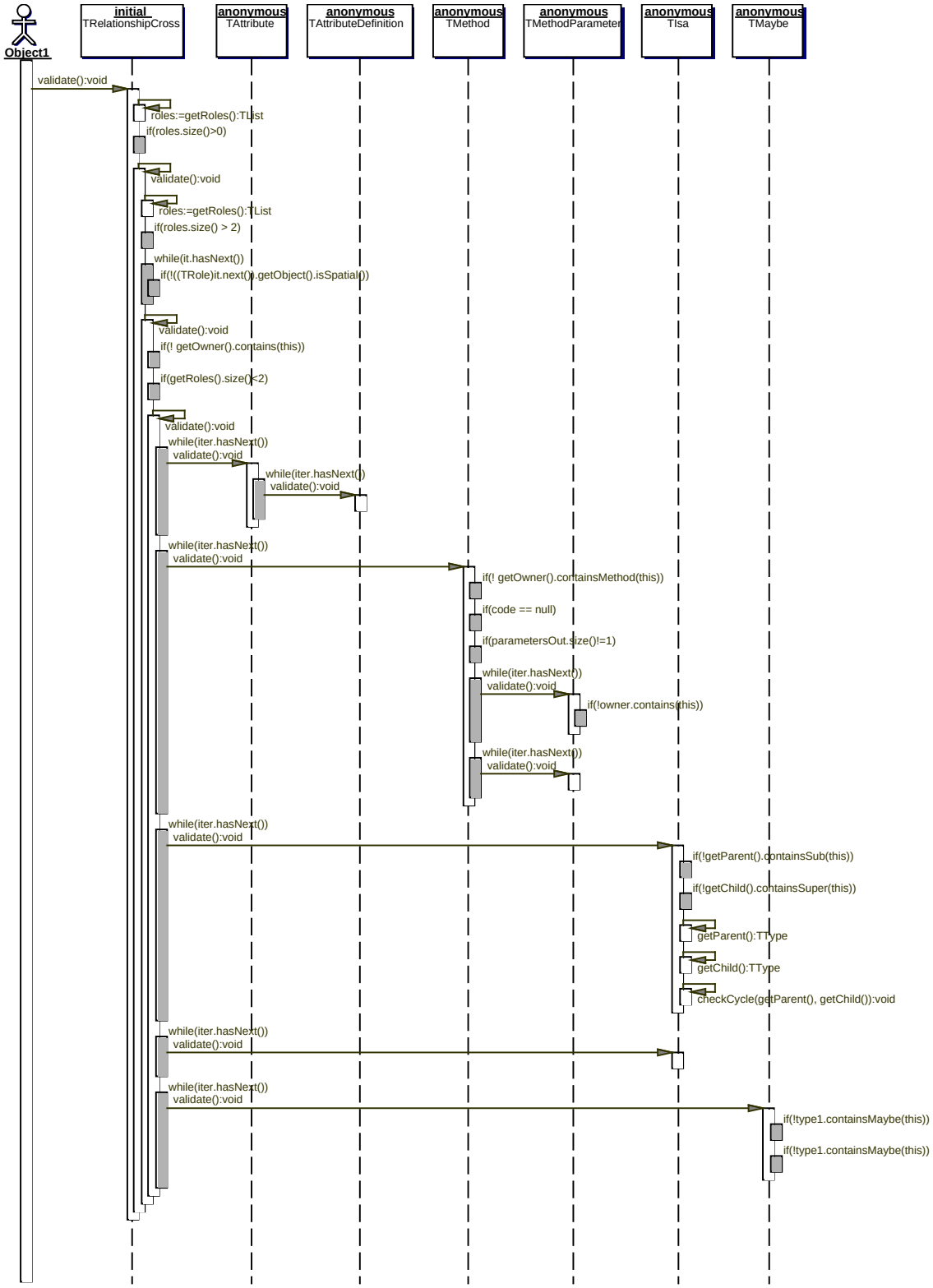

Figure 4.7: Cross relationship type validation

4.2 General analysis

The schema editor desired for MurMur is a graphical one. The application is designed to allow the user to fully use the features of the MADS model. A fundamental notion that drives the implementation is that the structure is to be used in more than one tool, and in most cases only the schema is important, not its graphical representation. Decision was thus taken to design the data structure and the graphical interface separately. Doing so allowed to design a structure close to the specifications of the syntax. Also features dependent on the syntax, but independent from the interface, such as exporting or validation, would take advantage of being designed upon the data structure only, without interference from the graphical part. This allowed to use the grammar associated to the data definition language of MADS and to define a data structure on the same model. Therefore, the analysis of the structure is going to follow the tree-like organisation of the MADS grammar.

The structure was named *TStructure*, after the structure realized in preliminary work on MADS. The introduction of multi-representation changed so much the initial structure that nothing else was used from the old TStructure, and it was actually seen as a pivotal point in the structure. The complexity of this concept made for a clear separation between elements supporting multi-representation and those that did not. All these can be grouped and are part of the definition of domains. Even if they have attributes and methods, like object or relationship types, their definition, without multi-representation, is much simpler. As such, building the data structure was split in two steps: the domain part, with all related elements, on one side and the rest of the elements on the other side, named general part further on.

Syntactic constraints are a capital component of the MADS model. These constraints, induced by the grammar, range from naming rules to acyclicity of the inheritance graph. Including these in the structure was critical to provide integrity of the schema. The inclusion of constraints can be done at two different levels. Either as an error checking, when the user can add, update, delete or remove an element of a schema, or as rules to be checked during global validation of the schema. The first possibility induces that the user must deal with error messages asking for correction, thus sometimes impeding the schema modification. However the second choice would result in a lot of time spent correcting mistakes at validation time. According to these considerations, the decision was to include constraints as soon as possible in all aspects, to ensure that global validation be as fast as possible, checking only the more complex issues. A top-down approach was designed, making an element valid by checking the validity of its component, starting on the schema itself.

Representation dependent elements, under the chosen top-down approach consist of three levels. The top level is the schema element, qualified as the syntactic parent of all other elements. The next level consist of representations, object and relationship types. The third level is made of components of object and relationship types, including is-a links, maybes and roles.

The domain part is built around the two distinct concepts of predefined domains on one side and user defined ones on the other.

4.3 Functional analysis

In this part, along the detailed description of elements, a coverage of the constraints will be done for each of the elements and thus the rules for validation, derived from the constraints, will be described. First the general part is described, then the one concerning domains.

4.3.1 The schema and other elements

Schema and metadata

The schema is the container of all other elements, including domains. As such it is defined as the top element. On another level, only the schema maintains metadata, such as its creation or modification dates. This meta data must also support customisation from the user. It was deemed useful to have the notion of a predefined schema, containing all predefined MADS elements. For now this notion only concerns predefined domains, as will be explained in Section 4.3.2.

On the side of constraints and validation, no specific rule exists for a schema, thus making it valid if its elements are valid.

Schema components

The second level contains representations, object types and relationship types.

Representations This notion is different from the other two. Representations are defined by their viewpoint and resolution. An empty resolution or viewpoint (noted NULL) is allowed inside a representation, making the definition valid for all resolutions or viewpoints respectively. Validation is done by checking that no more than one parameter of a representation is empty and that no two representations regroup the same viewpoint and resolution.

Object types and relationship types They have similar features. They both have attributes and methods, and can be linked together. They also use the notion of identifiers, working in the same way as keys in classical DBMS. As for the differences, object types must have a unique name inside the schema, while no such rule exists for relationship types. Still the main difference resides in the way they handle roles. These are created and defined by the relationship type, and simply bind an object type to it. Thus the definition of relationship types and object types was split between two levels, one regrouping common characteristics, called the type notion, the other used to define differences.

For relationship types, there are other levels, since constraints are different depending on the kind of relationship type. For example, all spatial relationship types are regrouped under the common definition of topological relationship type, while the synchronisation relationship type regroups all temporal relationship types.

For validation, since the naming rule is enforced at every change of the name for an object type, it does not need to be checked again. There are no other constraints on object types, and thus no validation rule needed. For relationship types, one check performed is that they link at least two object types, through a role. For binary relationships, no more than two roles are allowed. For relationship types with predefined names on roles, a check is performed to ensure that they are used. Also topological relationship types must link spatial object types, while synchronisation one must link temporal object types. Validation then proceeds by checking all components of types. Roles are checked only descending from the relationship type, not from the object type.

Type components and links

The third level is split in two parts. The first one concerns attributes and methods, the second concerns links between types.

Attributes and methods Both are components of types, as such they have common characteristics, which were regrouped under the *property* concept. This mainly concerns inheritance behaviour.

The attribute definition, being representation driven is much more complex than the method definition. The attributes are split into three categories:

- **simple**, bound to a domain giving the set of values it can take,
- **complex**, composed of other attributes, without restriction on their categories,
- **derived**, defined by referencing an attribute from another type, simple or complex.

Attributes have a cardinality, can be functionally dependent on another attribute. They may be either spatial- or temporal-varying. All these characteristics are representation dependent. For example, an attribute may be complex under one representation and derived for another one. This lead to the creation of the attribute definition notion, to manage definitions driven by representations.

Methods are defined by a code and a set of parameters. These parameters do not support multiple representations and thus are separated from the attribute definition. Inside methods, parameters are organised in two categories: input and output parameters. Output parameters are used to determine the return type of the method.

Properties of types must satisfy naming constraints stating that their name must be unique inside the type they are defined in. This constraint is enforced every time a naming operation occurs. For validation, a check needs to be done on the derived attribute to ensure that the reference exist and is reachable. The rule for referencing is that the referenced attribute must be reachable with a path composed of types and links. The same holds for functional dependency. In methods, validation is performed by checking that they have exactly one output parameter.

Is-a links, maybe links and roles The common point of all three is that they link two types together and can be regrouped. As such, common characteristics are defined under the link notion.

Is-a links and roles are oriented, going from child to parent for the is-a and from relationship type to object type for the role, while the maybe link is completely symmetric. Is-a and maybe links connect similar types, either object types or relationship types.

Roles have three notions of cardinality linked to them. The first two are the classical ones, cardinality on the object and relationship types. The other, optional, concerns the historical cardinality on the object type. Another feature of roles is their inheritance. As properties, roles can be inherited and behave in the same way.

A grouping notion exists on all links. This grouping obeys a constraint, different from the syntactic constraint. It is a semantic constraint which rules the way instances are split inside the linked types. An example would be the generalisation of *Person* in *Man* and *Woman*. There a *partition* exist, instances of Person must be either Man or Woman, and cannot be both obviously. This constraint notion is included in the group notion, which defines common characteristics for groups of links. Is-a links adds the notion of predicate, thus allowing definition of clusters. This predicate is a semantic constraint based on an attribute of the linked supertype.

The linking of same types by is-a and maybe links is enforced at creation, and every time a target of the link changes. Another check performed at creation time ensures that the newly created link does not duplicate an existing one, or reverse an is-a link.

The check for cycle in the inheritance hierarchy is done at validation because it could be a time consuming creation check in a complex schema. Otherwise, no other check is performed at validation time, thus links, excepted is-a, are valid due to the way they are created.

4.3.2 The domains

As said in Section 4.2, domains can be predefined or user-defined ones. A common feature of all domains and their components is that they do not support multiple representations.

A general naming constraint on domains is that no two domains can have the same name inside the schema they are defined in. This is enforced at each naming operation.

Predefined domains

The MADS predefined domains are split into three categories: basic, spatial and temporal. Basic domains are defined on standard data structure such as Boolean, integers, reals and strings. Spatial domains are based on the MADS Spatial Abstract Data Types described in Section 3.2.1. Similarly, the temporal domains are based on the MADS Temporal Abstract Data Types described in Section 3.2.1.

Predefined domains are considered valid by definition and are constrained by the fact that the user cannot modify them.

User-defined domains

They are split into three categories: intervals, enumerated, and structured. Intervals and enumerations are domains defined on the predefined basic domains, Booleans excepted. They inherit all properties from these domains and additional methods can be defined on them. Methods on domain, although simpler due to the absence of multi-representation, support the same features as methods of types. These domains define, through a series of values or lower and upper bounds, a restriction on a predefined basic domain.

Structured domains are more complex, supporting attributes in addition to methods. Since these attributes do not need to support multi-representation, they are simpler that attributes of types. But they follow the same structure: simple, complex or derived; and have the same properties. They can also take place in an inheritance hierarchy.

A constraint imposed on user-defined domains is that, before they can be deleted from the schema, they must not be used by an element, for example in a simple attribute. A constraint on intervals and enumerations is that the values given must be compatible with the basic domain they are specified on. Otherwise they are valid if their methods and, in the case of structured domains, their attributes are valid.

4.4 Implementation

The language chosen for implementation is Java, an object-oriented programming language from Sun Microsystems. This language has the advantage to provide classical functionality of programming languages, while in the same time having a very complete library of functions allowing to design graphical user interfaces. All these libraries are also platform independent, meaning that a Java program runs in any operating system.

Among the many features of this language is the notion of package. A package is like a container for classes, providing a name-space encapsulation, based on the directory hierarchy of the code. This concept provides specific access rules between member classes. This notion was used to follow the project separation described before. A common root name was chosen for all the packages: `mads.tstructure`. This name is then extended with `core` for the package concerning the schema and its elements, with `domains` for the part concerning domains, and finally a `utils` package is used to store all classes not part of the structure, but used by it as tools.

This package typically includes the exceptions defined on the structure. This exception throwing mechanism is used to deal with constraints. This allow the program to signal an incorrect behaviour without terminating the application. Exception definitions are regrouped in a separate package, `exceptions` inside `utils`. For validation, the first version of the editor only reports the first error found. This was done to allow for more important features to be developed. A more complex support will be provided in future versions.

The description of the implementation includes the various class diagrams which were used to design the hierarchy of the structure. Some conventions are used on these diagrams:

- An italic class name indicates an abstract class or an interface.
- Names in the upper right corner indicates the super class when it is not on the diagram.
- underlined indicates that the member is static, that is, a class variable.

- List of symbols preceding class members:

  —▷ : indicating that the class inherits from the pointed class.

  ⟶ : indicating that the class has a reference to the pointed class.

  \+ : indicating that the member has public visibility.

  \# : indicating that the member has protected visibility.

  \- : indicating that the member has private visibility.

  : indicating the default visibility, package.

More complete description of the methods and fields of classes can be found on a CD-ROM. An HTML documentation is provided as well as the source code. Description of the CD-ROM can be found in Appendix B.

4.4.1 Package `mads.tstructure.core`

This package contains all schema elements, including the schema itself. The first class diagram, shown in Figure 4.8, present a view of the general organisation of this package.

Schema and meta data

The class `TSchema` holds the definition of the schema, its name and eventual comments. This class uses lists to store all the elements of a schema: representations, domains, object types, and relationship types. It also stores the predefined schema, as a static member. When a new schema is created, it uses the predefined schema to initialise itself, for example getting all predefined domains. Metadata is dealt with in the `TMetadata` class, which allows definition of custom properties and storage of an history of the schema. This is not implemented in the first version of the editor.

Validation of a schema is realised by validating all its components.

Schema components

Representations The representation construct is stored in the `TRepresentation` class. It uses a viewpoint, defined in `TViewpoint`, and a resolution, defined in `TResolution`. The schema editor needs to be aware of all existing viewpoints and resolutions, since other applications may also use this information. This is done by managing a list, as a static class member, in each of the two classes. All representations existing on a schema are stored in `TSchema`. `TRepresentation`, `TViewpoint`, and `TResolution` store a name and

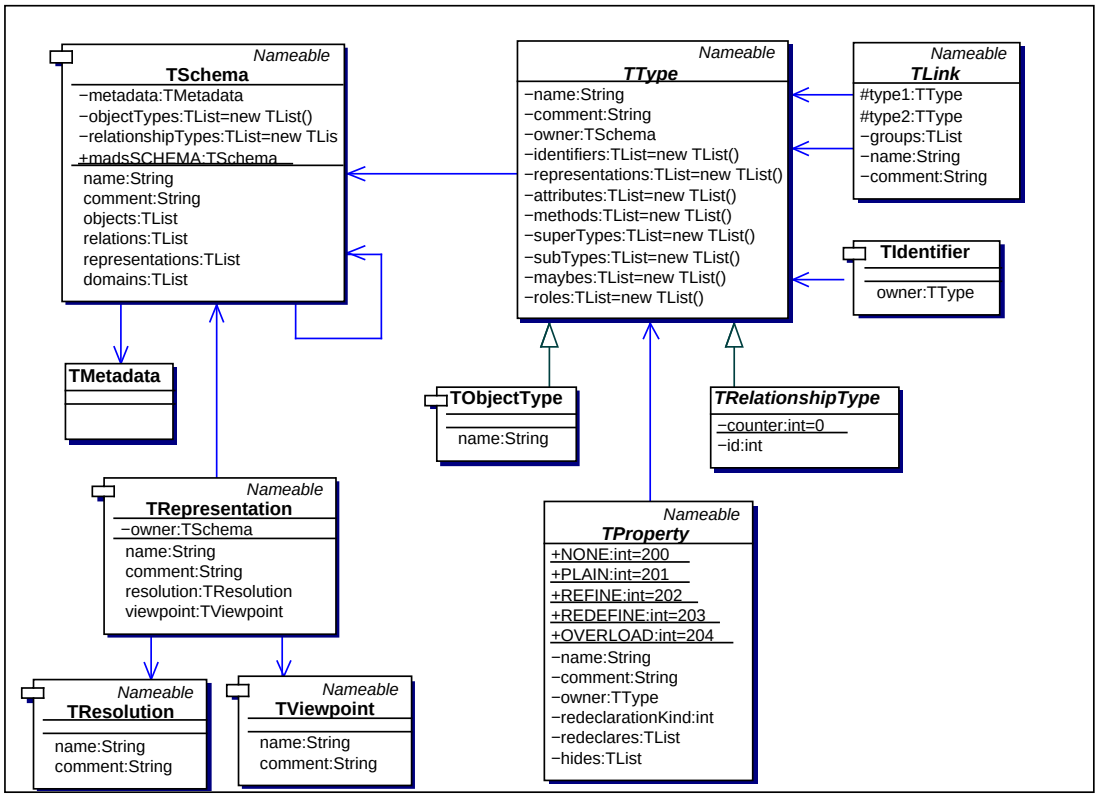

Figure 4.8: General class diagram of `mads.tstructure.core` package.

comment, to be used as textual recognition and description. The class diagram of this structure can be found in Figure 4.8.

In the first version of the editor, all viewpoints and resolutions declarations are valid. As for representations, they can have at most one null element specified, either resolution or viewpoint.

The definition and use of representations is not supported in the first version of the editor.

Object types and relationship types The type notion is managed by the `TType` class. It provides name and comment fields, and deals with the attributes, methods, maybe and is-a links, representations, and identifiers in lists. A `TIdentifier` class is defined to handle identifier definitions.

Specific characteristics of object types are managed in `TObjectType`, a class inheriting from `TType`. This class does not add any fields, it mainly specifies methods with constraints corresponding to object type specifications, such as the naming one.

For relationship types, their common properties are grouped in the `TRelationshipType` class, also inheriting from `TType`.

However, each kind of relationship type is declared in a separate class, according to a hierarchy based on constraints. The hierarchy has nothing to do with the correspondence hierarchy specified in Section 3.3.2. For example, all topological relationship types are declared as subclasses of `TRelationshipTopological`, itself inheriting from `TRelationshipType`. The same structure exists for synchronisation relationship types with the class `TRelationshipSynchronization`. A class diagram of the relationship types hierarchy is shown in Figure 4.9. The class diagrams of topological and synchronisation relationship types can be found in Figure 4.10.

These classes enforce all the necessary constraints. For example, topological and synchronisation relations are binary relationship types, and this is enforced at validation by checking that exactly two roles are defined. All relationship types with predefined role names check at validation that only roles with these names are used.

Type components and links

Attributes and methods The common characteristics, under the property concept are represented inside the `TProperty` class.

For the attribute, a `TAttribute` class is defined, inheriting from `TProperty`. The attribute definition holds a list of `TAttributeDefinition`, linked to representations. This

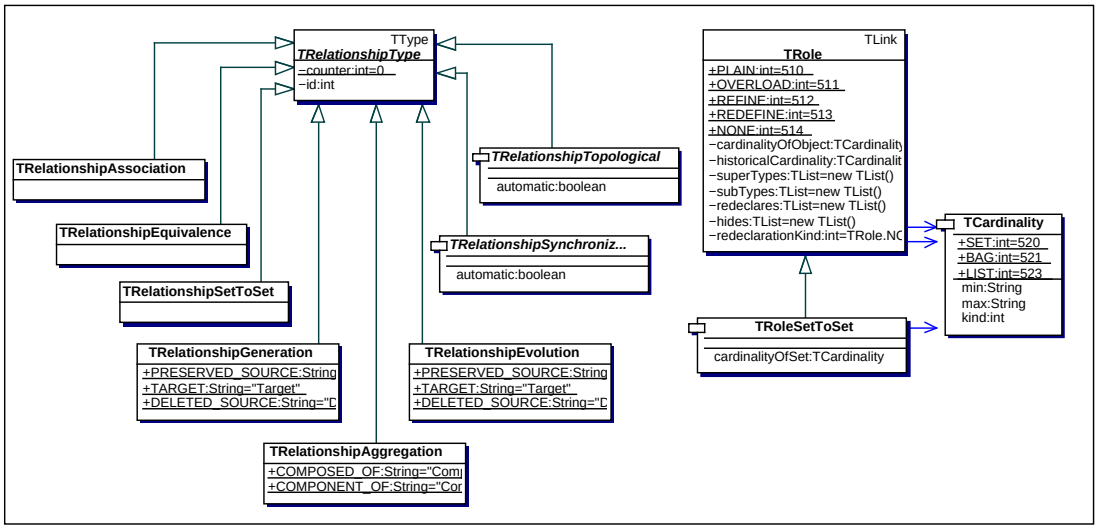

Figure 4.9: Relationship types class diagram.

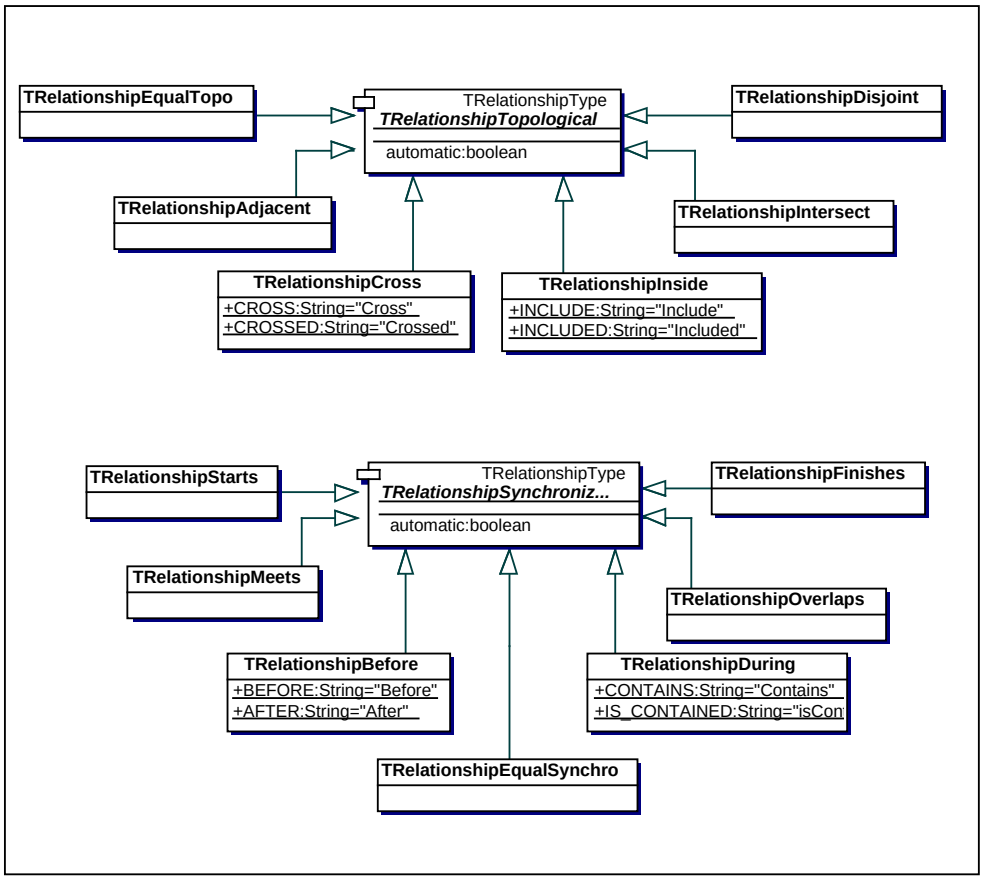

Figure 4.10: Topological and synchronisation relationship types class diagrams.

definition actually contains all properties common to the three kinds of attributes. It is then subclassed in `TAttributeSimple`, `TAttributeComplex`, and `TAttributeDerived` to allow handling of specific properties. For the cardinality, a `TCardinality` class is defined. To deal with space and time variations, a `TVariance` class is defined, subclassed in `TSpatialVariance` and `TTimeVariance`. The class diagram showing this structure can be found in Figure 4.11.

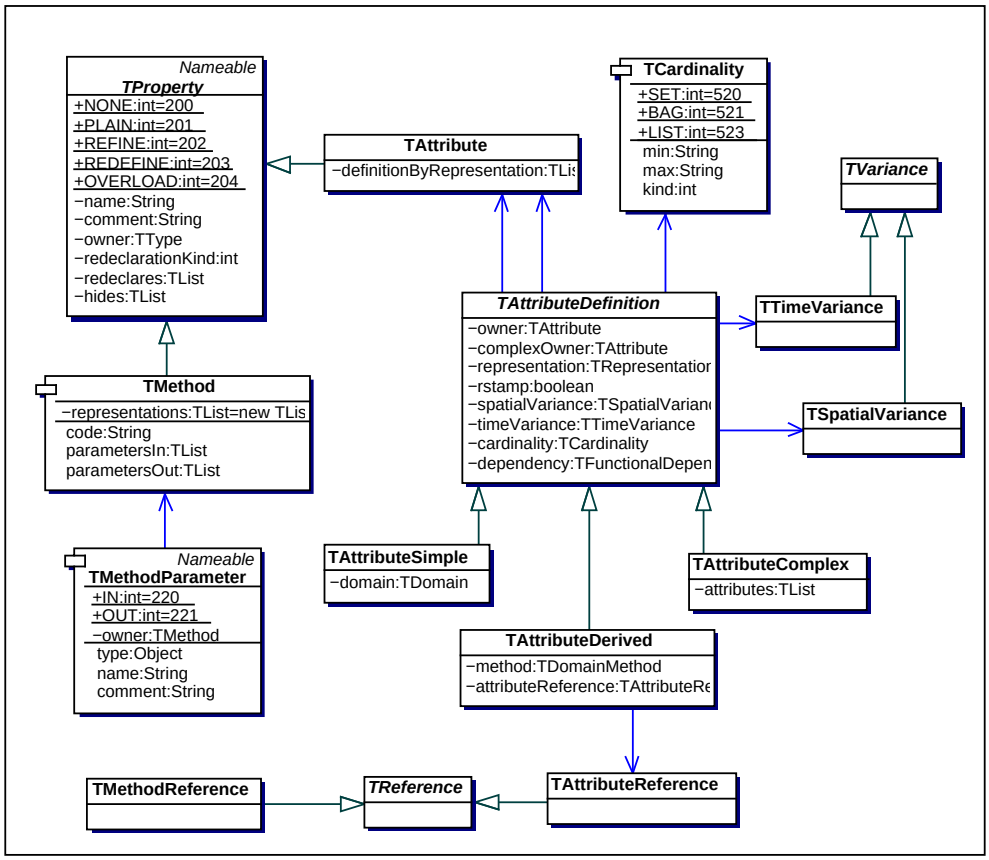

Figure 4.11: Properties class diagram.

The `TMethod` class contains all characteristics specific to methods, while the class `TMethodParameter` is responsible of parameters handling.

Is-a links, maybe links and roles The link concept is defined in the `TLink` class. The three kind of links, inheriting from this class, are defined in `TIsa`, `TMaybe`, and `TRole`. The role definition is valid, and simplified, for all relationship types, except SetToSet relationships. Since this relationship type allows definition of a second cardinality on the role, a `TRoleSetToSet` class has been created, inheriting from `TRole`.

For the grouping notion, the class `TGroup` is used. It is not supported, nor fully implemented in the first version of the editor. However the class structure already exists. The `TMaybeGroup` and `TRoleGroup` classes both inherits from the `TGroup` class, just adding validation by checking that only, respectively, maybe links and roles are part of the groups. For is-a links, the `TIsaGroupCluster` class is defined. In addition to grouping, it deals with the predicate that can be set on such groups. This predicate concept is dealt with in the `TPredicate` class.

The class diagram for links and groups can be found in Figure 4.12.

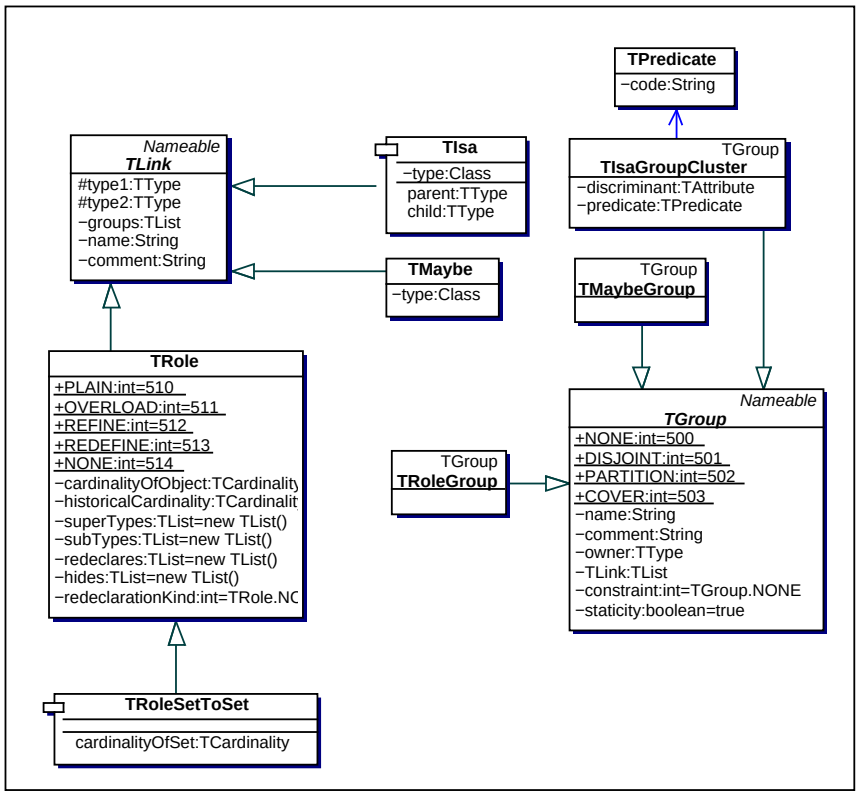

Figure 4.12: Links and groups class diagram.

4.4.2 Package `mads.tstructure.domains`

All domains are declared as inheriting from the `TDomain` class. This class was designed to interface this package with the core package. As such, the `TDomain` class is abstract and can never be instantiated. The class diagram giving the general organisation of the package can be found in Figure 4.13.

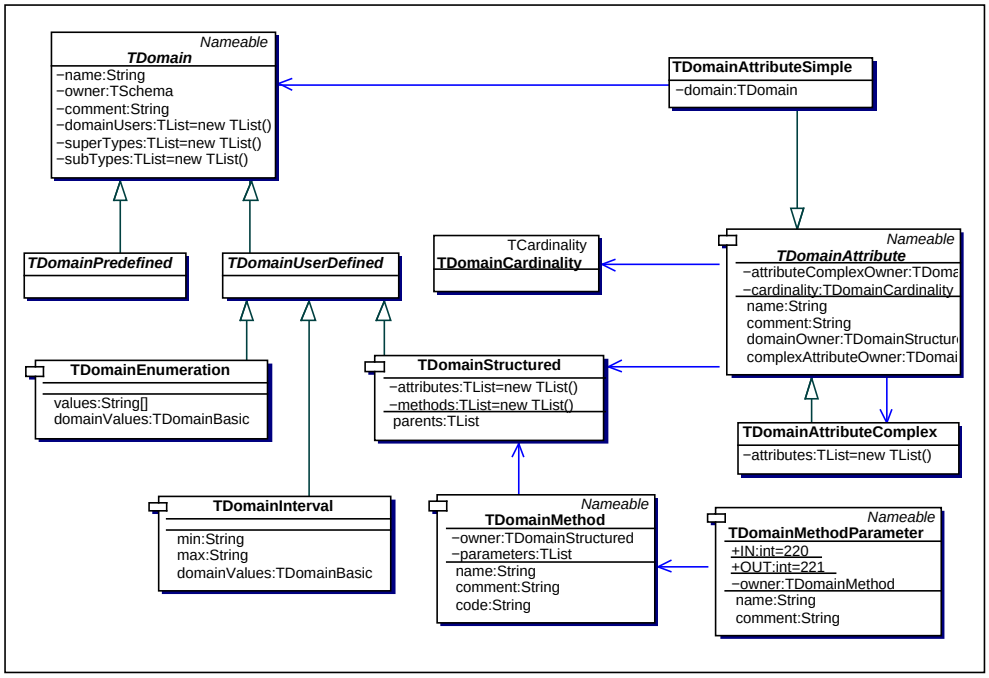

Figure 4.13: General class diagram of `mads.tstructure.domains` package.

Predefined domains

The predefined domains have their common properties grouped in the `TDomainPredefined` class, which are mainly restriction to `setXXX` methods prototyped in `TDomain`. The three groups of predefined domains are defined in `TDomainBasic`, `TDomainTemporal`, and `TDomainSpatial`. These classes use static members to handle all the predefined domains definitions.

The class diagram showing this organisation is given in Figure 4.14.

User-defined domains

The common characteristics of user-defined domains are regrouped in the `TDomainUserDefined` class. To handle domain methods, a `TDomainMethod` class is defined, along with a `TDomainMethodParameter` class.

Interval domains are defined in the `TDomainInterval` class, enumerated ones in `TDomainEnumeration`.

Structured domains are dealt within `TDomainStructured`. Their attributes are defined inside `TDomainAttribute`. This class is subclassed in `TDomainAttributeSimple` and `TDomainAttributeComplex`. The notion of cardinality is dealt inside

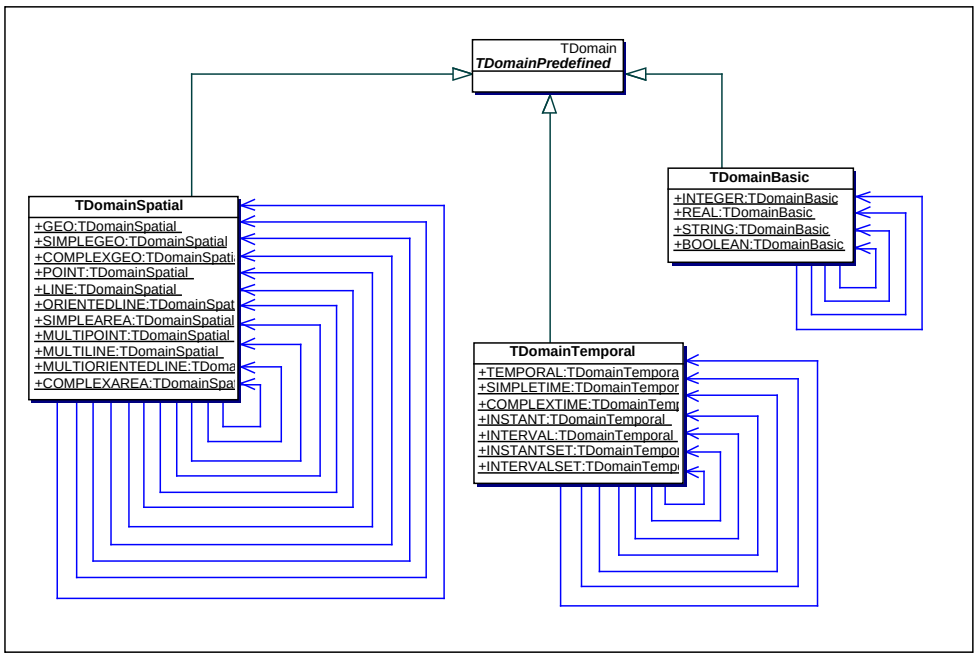

Figure 4.14: Predefined domains class diagram.

`TDomainCardinality`. This class was added to preserve package independency, but it is a simple redeclaration of the `TCardinality` class.

4.4.3 Package `mads.tstructure.utils`

This package contains classes used throughout the whole structure. Currently it is composed of one interface and one class.

The interface, `Nameable`, specifies two setters and getters, for name and comment. This interface was created to define conventions on naming methods. The class, `TList`, is an extension of the Java class `LinkedList`. It mainly adds a search by name feature. This was made possible by the use of the `Nameable` interface.

4.4.4 Package `mads.tstructure.utils.exceptions`

This package contains all classes created to be used as exceptions. They are used throughout the whole structure to serve as error message, while allowing a chance for recovery. They all derived from a `MADSException` class, which is an extension of the Java `Exception` class.

The `ElementException` class is the super-class of all exceptions coming from a wrong behaviour associated with an element. It is subclassed in three exceptions:

- `ElementAlreadyExistsException`, thrown when a definition duplicates an element, e.g., when defining a representation with the same viewpoint and resolution as another one;
- `ElementNotFoundException`, when the search for an element does not return anything; and
- `MaybeException`, when an invalid interrogation of a maybe link is done.

The `DefinitionException` class regroups all exception handling occurring during an element definition, for example, trying to link with an is-a link a relationship type and an object type. Six subclasses are defined:

- `IsaDefException`, thrown when an is-a link is wrongly defined.
- `MaybeDefException`, thrown when a maybe link is wrongly defined.
- `RoleDefException`, thrown when a role is wrongly defined.
- `EnumeratedDefException`, thrown when an enumerated domain is wrongly defined.
- `IntervalDefException`, thrown when an interval domain is wrongly defined.
- `AmbiguousDefException`, thrown when an element definition is ambiguous in some way.

The third group of exceptions is organised under the `InvalidException` class. It consists of three subclasses:

- `InvalidNameException`, thrown when a naming constraint is broken.
- `InvalidDeleteException`, thrown when trying to delete an element which cannot be deleted due to some constraint.
- `InvalidElementException`, thrown during validation, if an element does not satisfy some validation rule.

The class diagram of this organisation can be found in Figure 4.15.

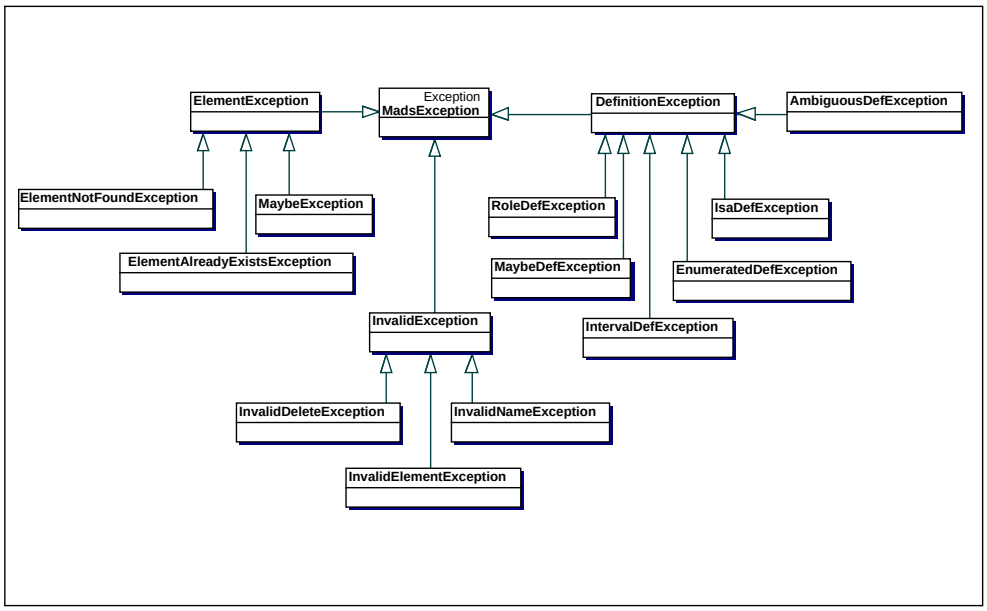

Figure 4.15: Class diagram of `mads.tstructure.utils.exceptions` package.

Chapter 5

Conclusions

Contents

5.1 Personal challenges 72

5.2 Issues to be considered for future works 72

A vast amount of literature exists on advanced data modelling. Spatiality, temporality, and multi-representation, all are concepts under development by research groups. As shown in Chapter 2, multiple ways of developing them may be envisaged and their expressive power is unquestionable. However, when considering models integrating the three above mentioned concepts, the number of available studies is rather low.

MurMur proposes a complex space modelling. It includes continuous and discrete views and allows spatial relationships. As far as temporality is concerned, the model does not confirm the concept of transaction-time. Entities, relationships, or attributes may otherwise be temporal. Multi-representation allows for multiple cartographic resolutions or semantic viewpoints. Even without considering the integration, MurMur offers a fairly complete data model. Its expressive power is further strengthened by the way all these concepts are approached orthogonally. The scientific community can certainly benefit from the advanced proposal of the MurMur project.

The data model of MurMur allows schema designing with a conceptual view easily understandable by end-users. Its characteristics, in order to be properly used, need to be implemented in graphical application tools. The schema editor targeted by the MurMur partners should provide this.

5.1 Personal challenges

Sharing a unique view of a project incorporating multi-representation is quite hard to achieve although the specifications of the data model have been agreed upon by the MurMur partners. The way the MurMur schema editor should present information to the user is still subject to discussions.

While the data structure was supposed to be implemented independently, its complexity forced me to take into account the general application design. This is why the structure is not complete at the present time, and why the existing implementation may even be modified before the first release of the schema editor. Although this increased the challenge, it made it much more interesting.

In this context, my work inside the programmers' team at the EPFL has been a very enriching experience. Firstly, from a scientific point of view as I learned a lot more in the field of data modelling, and in the efforts needed to translate conceptual modelling in a end-user tool. Secondly, from a personal way, as dealing with people having different backgrounds is always challenging.

5.2 Issues to be considered for future works

The history of computer research has repeatedly shown that the best theoretical solution does not always achieve success. In order for MurMur proposals to be accepted, the schema editor must be a user-friendly tool. An implemented data structure incorporating all the features of the data model will not be sufficient for the solution to be recognised among other available state-of-the-art proposals.

Along with schema editing, another very important feature in databases is the ability to extract information, i.e., to query the data set. In order to realise that, a query language must be specified and implemented. With only a schema editor and the related database, the user would find no use for the data model. The realisation of a good and powerful querying tool is another key toward MurMur's success. Here the complexity of the data model and its semantic makes the realisation of a user-friendly tool challenging.

Another important issue is the way MurMur schemas can be translated into existing Database Management Systems (DBMS). Since MurMur is about creating a tool for modelling conceptual schemas on top of existing solutions, it needs an existing platform to be used as permanent data storage. The quality of this export into multiple DBMSs will also bear strong influence on MurMur's success.

Appendix A

MADS syntax

This annex present the MADS data definition language 6.0.

Symbols used:

- [ ]: optional monovaluted;
- { }: optional multivaluted;
- { }*: mandatory multivaluted;
- 'symbol' : symbol is a literal
- |: logical or;
- *italic*: terminal symbol;
- **bold**: keyword; and
- (*text *): text is comment.

MADS DDL

Schema = **schema** *SchemaName*
['(*' *Comment* '*)'] (*Any text serving as a comment*)
'{'

[Representation]	(*This defines the set of allowed (viewpoint, resolution) couples to be used in conjunction with the schema. A couple may include a NULL symbol for either viewpoint or resolution, meaning that the specified viewpoint (resp. resolution) is valid for every allowed resolution (resp. viewpoint). The values that are specified for viewpoint (resp. resolution) implicitly provide an extensional definition of the DViewpoint (resp. DResolution) domain. If the representation clause is omitted, the associated schema is fully mono-representation, meaning that no representation specification may be stated in any definition within the schema.*)
{ UserDefDomain }	(*This defines the set of new domains that are defined for use within this schema *)
{ ObjectType }+	(*This defines the set of object types. There should be at least one for the schema to be validated*)
{ RelationshipType }	(*This defines the set of relationship types. The set may be empty.*)
{ Maybe }	(*This defines the set of maybe links. The set may be empty.*)
{ Cluster }	(*This defines the set of clusters. The set may be empty.*)
{ GroupConstraint }	(*This defines the set of group constraints. The set may be empty.*)
'}' ;	

Representation = **representation**
[Comment]
'{' { '(' *Viewpoint* ',' *Resolution* ')" }+ '}'

(*At this point we keep this as a set of <viewpoint,resolution>pairs. Later we shall include the possibility to have a more precise definition of resolutions. Indeed, variability in semantic and geometric resolution may independently result in sets of resolution values where the ordering is significant (e.g., more detailed or more precise). Thus, the generic value domain for resolutions is a set of ordered sets. We shall therefore include the specification of the ordering and the specification of how moving from one element to the next is to be performed, i.e. either by replacement or by interpolation. In the latter case we need to allow specification of the interpolation function to be used.*)

(********** The following lines define domains **********)

UserDefDomain = **domain** *DomainName*
[Comment]
'{' (StructuredDomain — Enumeration — Interval) '}' ;

StructuredDomain = [D-Supertypes] ComplexAttribute { Method };

(*Constraint: cycles -ISA cycles and composition cycles- are not allowed in the definition of domains.*)

D-Supertypes = isa *DomainName* { ',' *DomainName* } ;

Enumeration = **enumeration** (**Integer** | **Real** | **String**) '(' Value {',' Value }+ ')' ;
(*Constraint: values must belong to the BasicDomain that is specified*);

Interval = **interval** (**Integer** | **Real** | **String**) '[' Value ',' Value ']';
(*Constraint: values must belong to the BasicDomain that is specified*);

Domain = UserDefDomain | PredefinedDomain;
(* for use in the definition of SimpleAttribute *)

PredefinedDomain = BasicDomain | SpatialDomain | TemporalDomain ;

BasicDomain = **Integer** | **Real** | **Boolean** | **String** ;

SpatialDomain = **Geo** | **SimpleGeo** | **ComplexGeo** | **Point** | **Line** | **OrientedLine**
| **SimpleArea** | **MultiPoint** | **MultiLine** | **MultiOrientedLine**
| **ComplexArea** ;

TemporalDomain = **Temporal** | **SimpleTime** | **ComplexTime** | **Instant** | **Interval**
| **InstantSet** | **IntervalSet** ;

(********** The following lines define object types **********)
ObjectType = **object** *ObjectName*
[Comment]
'{'
[Representation] (*This is used to state for which representations this object type is relevant. These representations are a subset of the representations defined at the schema level. If omitted, the object type is relevant for all representations defined at the schema level*)
[O-Supertypes]
[TypeProperties] (*May only be omitted if the O-Supertypes clause is present*)
{ Identifier }
'}' ;

O-Supertypes = **isa** *ObjectName* { ',' *ObjectName* } ;

TypeProperties = [LifeCycle] (*If specified, the object/relationship type is temporal*)
[Geometry] (*If specified, the object/relationship type is spatial*)
{ Attribute }
{ Method } ;

LifeCycle = **lifecycle** (LifecycleDeclaration | Derived Attribute)
[Redeclaration TypeName '.lifecycle' {',' TypeName '.lifecycle'}] ;
(*Redeclaration specifies that this definition acts as a redeclaration of the lifecycle(s) inherited via is-a links*)
(*Unlike geometry and any other attributes, Lifecyle is not multirepresented, The lifecycle evolves differently from how other attributes evolve. Lifecycle evolution is governed by create, suspend, reactivate, delete and kill operations, while evolution of other attributes (including geometry) is governed by insert, modify and delete operations*)

LifecycleDeclaration = (**Instant** | **Interval** | **InstantSet** | **IntervalSet** | **Temporal**)
(**DBTIME** | ValueDefinition)
(*A lifecycle declaration first specifies the temporal type associated to active periods. Next, it specifies the time extent in which the lifecycle of any instance of the object/relationship type must be contained. This is either DBTIME or a value of a temporal domain computed from another attribute*)

ValueDefinition = (Function '(' [AttributeReference {'.'MethodCall}
{',' AttributeReference {'.'MethodCall} }] ')')
| (AttributeReference '.' MethodCall)
| *Value* ;

Function = **sum** | **avg** | **min** | **max** | **union** | **count** | ? ;

AttributeReference = { (*RoleName* | TypeName)'.' } *AttributeName* { '.' *AttributeName* } ;

TypeName = *ObjectName* | *RelationshipName* ;

MethodCall = *MethodName*, '(' [Argument { ',' Argument}] ')' ;

Argument = ValueDefinition ;

DerivedAttribute = ' = ' ValueDefinition ;

Redeclaration = **refines** | **redefines** | **overloads** ;
Geometry = **geometry** { [Representation] [GeometryDeclaration | DerivedAttribute]
[Redeclaration TypeName '.geometry' {',' TypeName '.geometry'}] }+ ;
(* In case of several definitions depending on representation, at least the first one must include either the GeometryDeclaration or the DerivedAttribute specification *)

GeometryDeclaration = SpatialDomain [Timestamped] [Rstamped] ;

(*A geometry definition can be either explicitly defined (declared) or derived from some other spatial attribute definition. A geometry declaration specifies its domain, whether geometry values have to be timestamped, and whether they have to be stamped with the corresponding resolution. The latter makes sense only for multi-resolution geometries. If a derivation formula is used, the timestamped and Rstamped specifications are also derived by the formula. Redeclaration specifies that this definition of geometry acts as a redeclaration of the geometry (geometries) inherited via is-a links. Any component of a geometry definition may be representation dependent. Using the format above, the designer may first define what is common to all representations and then have any representation dependent item preceded by the corresponding Representation specification.*)

TimeStamped = **timestamped** (**DBTIME** | AttributeReference)
(**discrete** | **stepwise** | (**continuous** [*MethodName*]))
(*A timestamping specification includes the definition of the type of the function that characterizes value evolution. The definition of a continuous function may include the identification of an interpolation method. *)

RStamped = **rstamped** ;

(********** The following lines define attributes **********)

Attribute = **attribute** *AttributeName*
[Comment]
{ [Representation]
[AttributeDeclaration | DerivedAttribute]
[Redeclaration AttributeReference { ',' AttributeReference }]
[Dependency] }+ ;
(* In case of several definitions depending on representation, at least the first one must include either the AttributeDeclaration or the DerivedAttribute specification *)

AttributeDeclaration = [CardinalitySBL]
[SpaceVarying]
[Timestamped]
[Rstamped]
[*DomainName* | ComplexAttribute]

CardinalitySBL = '(' *IntegerValue* ',' (*IntegerValue* | ('n' [**set** | **bag** | **list**])) ')' ;
(*The keyword, set, bag, or list, is relevant only when the maximum cardinality is greater than 1*)

SpaceVarying = **spacevarying** (**DBSPACE** | AttributeReference)
(**discrete** | **stepwise** | (**continuous** [*MethodName*]))
(*A space-varying specification includes the definition of the type of the function that characterizes value evolution. The definition of a continuous function may include the identification of an interpolation method. *)

ComplexAttribute = '{' Attribute {',' Attribute } '}' ;

Dependency = **depends on** *AttributeName* {'.' *AttributeName* }
{',' *AttributeName* {'.' *AttributeName* } } ;

(********** The following lines define methods and indentifiers **********)
Method = **method**
MethodName '(' { MethodParameter { ',' MethodParameter } } ')' [Type]
(*Type is the type of the result *)
[Representation]
[Redeclaration TypeName '.' *MethodName* { ',' TypeName '.' *MethodName* }] ;

MethodParameter = ParameterName Type ;

Type = DomainName | ObjectName | RelationshipName ;

Identifier = **Id** Key { ',' Key } ;

Key = '(' AttributeReference { ',' AttributeReference } ')' ;

(********** The following lines define relationship types **********)
RelationshipType = **relationship** *RelationshipName*
[Comment]
'{'
[Representation] (*This is used to state for which representations this relationship type is relevant. These representations are a subset of the representations defined at the schema level. If omitted it means that the relationship type is relevant for all representations defined at the schema level*)
[R-Supertypes]
RelationType
[TypeProperties]
{ Identifier }
'}' ;

R-Supertypes = **isa** *RelationshipName* ',' *RelationshipName* ;

RelationType = StructuralRel | SynchroRel | TopoRel | GenerationRel | EvolutionRel
| UserDefRel ;

StructuralRel = (**association** Role {',' Role }+)
| (**aggregation** Role ',' Role) (* aggregation relationships have predefined role names*)
| (**equivalence** Role ',' Role) (*objects linked by an equivalence relationship must have the same oid*)
| (**settoset** StoSRole {',' StoSRole }+) ;

Role = **role** [*RoleName*] CardinalitySL [**histcard** CardinalitySL] *ObjectName*
[Redeclaration RoleReference { ',' RoleReference }] ;
(* histcard denotes so-called historical cardinalities. They have to define a larger range than the instant cardinality associated to the role *)

CardinalitySL = '(' *IntegerValue* ',' (*IntegerValue* — ('n' [**set** — **list**])) ')' ;
(* The set or list keywords are relevant only when the maximum cardinality is greater than 1 *)

RoleReference = *RoleName*
| (TypeName '.' *RoleName*)
| (*ObjectName* '.' *RelationshipName*) ;

SToSRole = **role** [*RoleName*]
CardinalitySBL [**histcard** CardinalitySBL]
CardinalitySBL [**histcard** CardinalitySBL] *ObjectName*
[Redeclaration RoleReference { ',' RoleReference }] ;
(* An StoSRole has two cardinality specifications as it links sets of object instances*)

SynchroRel = (**before** | **equal** | **meets** | **overlaps** | **during** | **starts** | **finishes**)
[**automatic**] Role ',' Role ;
(* These relationships link temporal objects*)
(* The automatic clause defines a relationship type whose instances are automatically generated by the system as soon as the geometry of two instances of the linked object types satisfy the synchronization predicate of the relationship *)
(* before and during relationships have predefined role names*)

TopoRel = (**disjoint** | **adjacent** | **intersect** | **cross** | **inside** | **equal**)
[**automatic**] Role ',' Role ;
(* These relationships link spatial objects *)
(* The automatic clause defines a relationship type whose instances are automatically generated by the system as soon as the geometry of two instances of the linked object types satisfy the topological predicate of the relationship *)
(* inside relationships have predefined role names*)

GenerationRel = **generation** (Role — StoSRole) ',' (Role — StoSRole) + ;
(* The relationship has predefined role names: preservedsource, deletedsource, and target. The preservedsource role states that instances of this source type continue to exist - active - after evolution. If it is not the case, the source type must be temporal *)

EvolutionRel = **evolution** Role ',' Role + ;
(* The relationship has predefined role names: preservedsource, deletedsource, and target. The preservedsource role states that instances of this source type continue to exist - active - after evolution. If it is not the case, the source type must be temporal *)
(* These relationships link object instances with the same oid *)

UserDefRel = **constrainedrelationship** ValueDefinition Operator ValueDefinition
Role {',' Role }+;

Operator = '=' | '<>' | '<' | '>' | '<=' | '>=' ;

(********** The following lines define Maybe, Cluster, and Group Constraint ***********)

Maybe = **maybe** TypeName ',' TypeName [Comment] ;
(* Both TypeName must be of the same kind, either object names or relationship names *)

Cluster = **cluster** *ClusterName* **of** TypeName '(' TypeName {',' TypeName}+ ')'
[Comment]
[**on** QualifiedAttribute { Predicate { ',' Predicate } }] ;
[Constraint] ;
(* All TypeName must be of the same kind, either object names or relationship names *)

QualifiedAttribute = [TypeName '.'] *AttributeName* { '.' *AttributeName* } ;
Predicate = (OperatorPredicate | IntervalPredicate) '→' TypeName ;

OperatorPredicate = **when** Operator *Value* ;

IntervalPredicate = **when inside** *Value* ',' *Value* ;

GroupConstraint = RoleConstraint — MaybeConstraint ;

RoleConstraint = **role-constraint** *ConstraintName* Constraint on *ObjectName*
'(' QualifiedRoleName { ',' QualifiedRoleName }+ ')' ;
Constraint = **cover** — **disjoint** — **partition** ;

QualifiedRoleName = (*RoleName* — *RelationshipName*) ['.' *ObjectName*] ;

MaybeConstraint = **maybe-constraint** *ConstraintName* Constraint
on TypeName '(' TypeName {',' TypeName}+ ')' ;
(* All TypeName must be of the same kind, either object names or relationship names *)

Appendix B

Contents of the CD-ROM

The CD-ROM can be obtained from the director of this work Prof. Esteban Zimányi, Informatic Department, Faculty of Applied Sciences, Université Libre de Bruxelles.

It contains the following:

- **The javadoc**: It is presented in the *documentation* folder. It is an HTML documentation and the main file is *index.html*.
- **The source code**: All the Java source files are presented in the *sourcecode* folder. They are presented according to the package organisation described in Chapter 4.

The source code is property of the MurMur project.

Bibliography

[1] ISO TC 211/WG 2. Geographic information - Spatial schema. Technical report, ISO TC 211 N924, May 2000.

[2] C. Parent et al. MurMur Project - Workpackage 2 - Delivrable 4 - State of the Art review. Technical Report UNIL/CP/MM-WP2-DLA-004/V4.0, Université de Lausanne, September 2000.

[3] D. Pfoser and N. Tryfona. Requirements, definitions and notations for spatio-temporal applications environments. Technical Report CH-98-09, Chorocronos, November 1998.

[4] C. Parent, S. Spaccapietra, and E. Zimányi. Spatio-temporal conceptual models: Data structures + space + time. In *Proceedings of the 7th ACM Symposium on Advances in Geographic Information Systems*, ACM GIS'99, pages 26–33, Kansas City, USA, 1999.

[5] R. Price, K. Ramamohanarao, and B. Srinivasan. Spatio-temporal extensions to unified modeling language. In *Proceedings of the Workshop on Spatio-Temporal Data Models and Languages*, IEEE DEXA'99, Florence, Italy, 1999.

[6] M. Wachowicz. *Object-Oriented Design for Temporal GIS*. London: Taylor & Francis, 1999.

[7] J.F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM*, volume 26, pages 832–843, 1983.

[8] D. Peuquet. It's about time: a conceptual framework for the representation of temporal dynamics in geographic information systems. In *Annals of the Association of American Geographers*, volume 84(3), pages 441–461, 1994.

[9] K.K. Al-Taha and M.J. Egenhofer. Reasoning about gradual changes of topological relationships. In I. Campari, U. Formentini, and A.U. Frank, editors, *Theories and*

Methods of Spatio-Temporal Reasoning in Geographic Space, pages 196–219. London: Springer-Verlag, 1992.

[10] W.W. Chu and G. Zhang. Associations and roles in object-oriented modeling. In *Proceedings of the 16th International Conference on Conceptual Modeling*, ER'97, pages 257–270, Los Angeles, California, USA, 1997.

[11] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. In *ACM Transactions on Information Systems*, volume 14(3), pages 268–296, 1996.

[12] M. Gentile. *An object-oriented approach to manage the multiple representations of real entities*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1996.

[13] B. David, L. Raynal, and G. Schorter. Geo2: Why objects in a geographical dbms ? In *Proceedings of the 3rd International Symposium on Advances in Spatial Databases*, SSD'93, pages 264–276, Springer, 1993.

[14] D. Pantazis. Con.g.o.o.: A conceptual formalism for geographic database design. In M. Craglia and H. Couclelis, editors, *Geographic Information Research – Bridging the Atlantic*, pages 348–367. Taylor & Francis, 1997.

[15] H. Gregersen and C.S. Jensen. Conceptual modelling of time-varying information. Technical Report TR-35, TimeCenter, September 1998.

[16] E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extendign the odmg object model with time. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP'98, pages 41–66, Brussels, Belgium, 1998.

[17] Y. Bédard. Visual modelling of spatial databases: Towards spatial pvl and uml. In *Geomatica*, volume 53(2), pages 169–186, 1999.

[18] O. Gayte, T. Libourel, J.P. Cheylan, and S. Lardon. *Conception des systmes d'information sur l'environnement*. Hermès, Paris, 1997.

[19] N. Tryfona and C.S. Jensen. Conceptual data modeling for spatiotemporal applications. In *GeoInformatica*, volume 3(3), pages 245–268, 1999.

[20] S. Spaccapietra et al. MurMur Project - Workpackage 2 - Delivrable 5 - Data Model Specification. Technical Report EPFL/SS/MM-WP2-DLA-005/V4.0, Ecole Polytechnique Fédérale de Lausanne, September 2000.

[21] E. Gillet, S. Monties, et al. MurMur Project - Workpackage 4 - Delivrable 10 - Schema Editor specification. Technical Report STAR/EEO/MM-WP4-DLA-010/V4.0, Ecole Polytechnique Fédérale de Lausanne, December 2000.

[22] S. Spaccapietra et al. MurMur Project - Workpackage 4 - Delivrable 11 - Schema Editor. Technical Report EPFL/SS/MM-WP4-DLA-011/V3.0, Ecole Polytechnique Fédérale de Lausanne, March 2001.