A Framework for Building Spatiotemporal Applications in Java

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Erik Voges
2001

Supervised by
Associate Professor Sonia Berman

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A relatively new area of study that first received attention during the past decade explores how spatiotemporal data can be efficiently used in applications and stored in databases. Spatial data includes the locations or positions, and possibly also the size and orientation of physical objects. Temporal data is time-stamped, i.e. every piece of temporal data is associated with at least one point or interval in time. Spatiotemporal data is both spatial and temporal. It would denote, for example, where a particular object was at a given time, or when an object had a certain size.

There is a need for many spatial applications to have temporal functionality added. Often such an approach is fraught with problems, since the existing designs were specifically tailored for spatial applications, and changing those designs invariably leads to poorer performance or some other compromises. Because it is not easy to change an existing spatial or temporal system to a spatiotemporal system, it makes sense for developers to start building new systems that are optimised for handling spatiotemporal data.

Building an application from the ground up is a daunting task, but there are powerful technologies in existence that can expedite the process. One such technology is found in persistent programming languages, which relieves the application builder of the task of storing and retrieving data (e.g. on a database or file). Persistence technology was created with the aim of making complex applications such as Geographic Information Systems (GIS), Computer Aided Design (CAD) systems and Computer Aided Software Engineering (CASE) software easier to create and maintain. When using a persistent language, the same data types and operations are used for both transient and database data – there is no need for code to translate between store and memory formats or to transfer objects between memory and disk. In contrast, programmers using conventional databases need to be able to program in two languages: the database language (like SQL) and a programming language (like C++ or Java). To date there has been little work done to actually apply persistence technology to the GIS domain [MIA96, SVZ98]. One problem with a persistent programming approach in this context is that one has

to spend a great deal of time to work out how spatiotemporal data is going to be dealt with in the application, as noted in trying to build a PJama (Persistent Java) system for the local Sea Fisheries Research Institute in our department [SVZ98]. How will spatiotemporal data be represented, what methods are necessary for manipulating the data, and where would spatiotemporal data and methods be positioned in the overall structure of the program?

This thesis aims at developing a framework that can be used to make persistent spatiotemporal systems a viable alternative to conventional GIS applications. The approach adopted involves finding a suitable data model, implementing this as a PJama (Persistent Java) class library, extending this with structures to improve performance, and evaluating the result. The contribution of this thesis is thus directed towards finding answers to the following questions:

- What is needed for a library that can aid the rapid development of persistent spatiotemporal applications?
- How does a library facilitate the process of building a spatiotemporal application?
- How does using persistent Java contribute to productivity and are there any trade-offs in code quality or reusability?

Some initial research was done to find out whether there is a proven best way to handle spatiotemporal data, but in current research no particular model stood out significantly above others. One particular model was however found to be defined more clearly than most, and it seemed to be a simple and conceptually clean approach for working with spatiotemporal data [GBE+00]. It is based on good design principles that make it easy to understand and use, and thus it seemed the best candidate to implement as a framework that can be used for persistent spatiotemporal applications. The model was implemented as a package of persistent Java classes to form the basis on which such applications can be built. The package was tested both in term of its usability and its performance.

This thesis is organised into 8 chapters. Chapter 2 gives background on spatial databases and temporal databases, and also presents a few existing spatiotemporal models. The Geographic Information Systems (GIS) perspective on temporal and spatial information is also described in chapter 2, and chapter 3 provides some background on persistence technology together with previous experiences of using persistence in a GIS context. Chapter 4 contains an overview and explanation of the data types and operations of the abstract spatiotemporal model that was implemented. Some examples are given to demonstrate how its operations are applied for answering spatiotemporal queries. The implementation of the model as Persistent Java (PJama [AJDS96]) classes is discussed in chapter 5. The use of object-oriented language features such as inheritance and polymorphism is highlighted and a brief overview is given of how the final product is applied when creating a spatiotemporal application. Chapter 6 provides background on spatiotemporal indexing, and outlines the approach that was taken for creating and adding indices to

the spatiotemporal classes. It also shows how indices were applied in the optimisation of certain operations. A number of different experiments with the package are described in chapter 7. One experiment, aimed at testing the usability of the model, involved giving students a number of tasks and comparing their solutions. Other experiments measure the performance of spatiotemporal queries using different indexing strategies, and investigate when it becomes viable to add indices to moving objects. Some GUI components developed for verifying the correctness of operations and visually creating test data are also illustrated in this chapter. Chapter 8 contains a summary of the work and of the contribution of this thesis, as well as some suggestions for future work.

Chapter 2

Background

The study of spatiotemporal databases draws on a number of topics that have been extensively researched already, such as spatial databases and temporal databases. But there are also aspects of this field of study that are relatively new and unexplored, such as the development of efficient indexing techniques for spatiotemporal data. Using indices as access structures is essential in databases that handle large volumes of data.

This chapter is aimed at giving the reader an overview of the concepts and ideas developed with temporal and spatial databases. Adding a temporal dimension to spatial data is also explored from the perspective of GIS, an area in which spatiotemporal data management is becoming increasingly important. Some proposed taxonomies for handling spatiotemporal data are introduced as well as a number of experimental access methods for spatiotemporal data. Indexing techniques for spatiotemporal data are discussed in chapter 6.

2.1 Time in Databases

Scientists and philosophers had different views on time: Newton saw it as a dimension similar to space, but separate from it. Einstein considered it to be a fourth dimension, interacting with space [Gre99] Rucker argues that the passage of time is an illusion. In his philosophy, every object or human is a pattern in a fixed space-time block, existing simultaneously and always. Time only seems to pass because humans' memories work backwards [Ruc77].

In the first part of this section an introduction is given to some concepts used for describing various aspects of time. From that follows discussions of adding temporality to relational and object-oriented systems and a brief overview of temporal access methods.

2.1.1 Structure and Dimensionality of Time

A *chronon* is the smallest duration of time that can be represented in a discrete model [SO95], [Lan92]. It is a line segment, rather than a point. Although there are also continuous models, and time is generally perceived as continuous, only discrete representations can be implemented [EGV97]. Time can be bounded either in the past or the future. It is currently believed time is bounded in the past to the big bang [Haw88].

Ordinal Time

In the linear model of time, time passes in a totally ordered way – one event taking place before or after some other event. There is a degree of transitivity that applies here. If event A happened before B, and B before C, then we can deduce that A happened before C.

Figure 2.1: Linear time

In some cases, a group of events can be totally ordered, meaning all the events happened in sequence. Allowing for equality, some events could be considered as taking place at exactly the same instance. In other cases, there could merely be a partial order where only the relative order of certain subsets of events are described [Fra98]. For instance, if John builds a house in Nyanga and Joe builds a house in Khayelitsha, both of them must first build a foundation, then walls, then a roof. But whether John puts his roof on before Joe does, is of no consequence.

Branching Time

Time could also have multiple perspectives in the future or the past. There could be various possibilities of events leading up to a certain event, or some event could have multiple possible outcomes [ST97]. See Figure 2.2. This model of time is different from

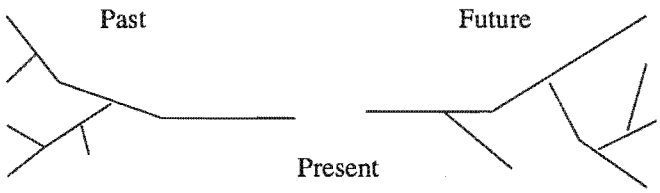

Figure 2.2: Branching time

the partially ordered time model, in that the same event can occur on various branches - as part of different scenarios [Fra98], [SO95]. Events in the partial order model occur once only.

Cyclic Time

Cyclic time is a loop for recurring processes. If time is cyclic, like the seasons of the year, transitivity breaks down. After summer comes autumn, then winter and spring, but then summer again (Fig. 2.3). Transitivity could lead to a statement like 'Spring is after summer' [Fra98].

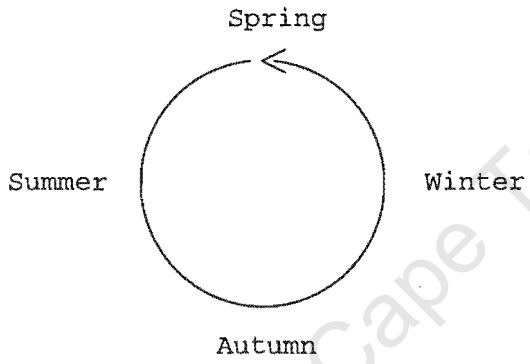

Figure 2.3: Cyclic time

Reasoning with intervals also change slightly when time is cyclic. Two cyclic intervals can complement each other, each starting at the end of the other, thus forming a whole cycle. Consequently, they could also overlap twice.

Valid Time and Transaction Time

In reality time has only one dimension, but there are often more than one type of time used in temporal database applications. An important distinction is made between *valid time* and *transaction time* ([Wor98], [SO95], [KIPS95], [ST97]).

Valid time describes when an event takes place in the real world, while transaction time describes when the event was recorded in the database. This 'temporal duality' is also seen in accounting, where nothing is erased - only amendments can be added later to balance out errors [Lan92]. Thus there is a fixed paper trail on which every single event can be traced.

There are also other kinds of time such as *user time* [SO95], [KIPS95] (defined and controlled by the database user) and *display time* [Lan92] (particularly in the field of

spatial databases and GIS - describing when data is represented on output devices such as a screen or plotter).

[KIPS95] notes a potential problem that has to do with the semantics of valid time. They use the example of and airline ticket reservation that contains information such as transaction date, issue date, scheduled flight dates, actual flight dates, and ticket validity periods. The question that arises is which of the dates represent *valid time*. For a travel agent the date of issue may be important and represent valid time, for the client the actual flight dates are probably more important. The fact is that the valid time field is chosen by whoever develops the system, but semantically it is really closer to user time. Proposed approaches to the 'multiple valid times' problem are:

- to create a new table for every different valid time, e.g to have separate tables such as *scheduled flight dates* in which the scheduled dates are valid time fields, and a table *actual flight dates*, where actual flight dates represent valid time.
- to have only user times or only valid times, that are labeled – here each user/valid time would be represented with a label which associates it with its own individual time line, e.g. all return flight dates are stored using a *return flight* label.

The latter was the recommended approach since the former could potentially create too much extra complexity in the database by increasing the number of tables, and resulting in less efficient joins. The labeled-times approach however is not without problems either, for example inconsistency problems may arise if some event gets added to the database long after it took place. Although transaction times are used to govern a situation like this, it is not clear whether the database should be allowed to change a bank balance, for example, if it finds a transaction with an earlier valid time time-stamp, or whether the user application should be required to resolve the apparent inconsistency between the different balances at the earlier and later valid times.

2.1.2 Time in Relational Databases

Relational databases are popular in business and other communities due to their appealing simplicity. However, the underlying relational data model does not support the natural variation of attributes of real-world objects over time. It has become increasingly important to have application support for time-varying information, especially since the problem of storing vast quantities of data has been resolved through technologies such as optical storage.

[SJ91] gives an overview of different approaches to adding temporal aspects to the relational algebra. They note that there are four basic design decisions that characterise the objects that these temporal algebras define. They are:

- How valid time is represented
- Whether valid-time time-stamps are associated with tuples or attributes
- Whether attribute values are atomic or allowed to be set valued
- Whether transaction time is associated with attributes, tuples or sets of tuples.

They have also developed a set of criteria that could be used to compare the various approaches (table 2.1). Out of 26 criteria they have shown that 7 are *conflicting criteria*

>	All attributes in a tuple are defined for the same interval(s)
	Consistent extension of the snapshot algebra
	Data periodicity is supported
	Each collection of legal attribute values is a legal tuple
>	Each set of legal tuples is a legal relation
	Formal semantics are well-defined
	Has the expressive power of a temporal calculus
	Includes aggregates
	Incremental semantics defined
	Intersection, θ-join, natural-join and quotient are defined
	Is, in fact, an algebra
	Model doesn't require null attribute values
	Multidimensional time-stamps are supported
	Reduces to the snapshot algebra
>	Restricts relations to first normal form
>	Supports a 3D view of historical state and operations
>	Supports basic algebraic equivalences
	Supports relations of all four classes
	Supports rollback operations
	Supports multiple stored schemas
	Supports static attributes
	Treats valid time and transaction time orthogonally
>	Tuples are time-stamped
>	Unique representation for each temporal relation
	Unisorted (not multisorted)
	Update semantics are specified

Table 2.1: Quality criteria for temporal algebras [SJ91]

(marked with a '>' in table 2.1) that could not all be satisfied by any of the evaluated algebras. The other 19 criteria could all be simultaneously satisfied, and the degree to which that is achieved is also an indication of the quality of the algebra.

2.1.3 Time in Object-Oriented Databases

The object-oriented paradigm is seen as a suitable approach for adding temporal semantics to data - OO models have facilities that are helpful in expressing the knowledge structure of applications [Wac98].

To add time to a database system, three enhancements should be done: extending the data structures, adding new operators, and having the ability to express temporal constraints [SN97].

Time-stamping can be done at object or at attribute level. If done at object level, either the object identifier can be time-stamped, or the object type (by adding a temporal attribute). When adding temporality to an existing OODBMS for example, time-stamping could only be done at type level (i.e. objects stored in the database receive an additional 'time' attribute).

More problems arise when adding temporal operators. They can either be added to the OODBMS directly, or as an additional layer on top of the existing system (i.e. an application that runs on OODBMS). In this latter case, the query language also needs to be extended to call the temporal operations. In terms of query optimisation it is not ideal for time support to be defined by the user. This is because access methods have not been employed for time-varying information [SO95].

In [SN97] a comparison is made between two approaches to temporal modeling object-oriented environments. The first presents a temporal object model (TOM) with query language for temporal database applications. The second is an extension of an existing OODBMS with temporal constructs and operations. It was found that while in the latter case, the query language was unclear and complex (temporal queries looked different from normal queries), it is not always viable to build in explicit support for temporal constructs into a system and its query language. An approach somewhere in between was suggested - certain constructs should be added into the model, but they should be extensible enough to be used not only for temporal extensions, but any other applicable domain.

2.1.4 Access Methods for Temporal Data

A B^+-tree is an index structure used for accessing ordinal data in one dimension – which is why it lends itself so well to the indexing of temporal data. There are numerous access methods for temporal data, and many are based on B^+-trees [GLOT96], [TML99].

A comparison has been made by [ST97] between different temporal indexing methods. Most of the indices in their comparison are transaction time access methods, however they do also discuss structures for valid time, as well as bi-temporal methods (involving both transaction and valid time). Some important problems are considered such as index

pagination and migration to tertiary storage.

Index pagination can greatly affect the performance of an index structure as it determines how nodes are paginated. The B^+-tree is considered a well-paginated structure as it requires $O(\log_B r)$ pages accesses when searching r objects, using a page size B.

Performance can also be improved if *logically* related data can be physically clustered together on disk. Achieving efficient clustering is more problematic in valid-time databases since deletions are done at the physical level and updates can happen anywhere in the valid-time domain. This dynamic behaviour means that although an efficient clustering can be achieved for some dataset, a number of updates could make it significantly less efficient. Transaction time and bi-temporal databases on the other hand, only need to append updates to the database, making it easier to create efficient clusters.

One approach to accessing bitemporal data is to use a multidimensional structure such as an R-tree [Gut84] (there are numerous variants of the R-tree [KS97], [BKSS90] and other multidimensional structures such as the X-tree [BKK96] or Pyramid-technique [BBK98]).

When viewing two dimensional time as rectangular regions (as in figure 2.4), there is a significant degree of overlap where transaction time extends to the present, i.e. when bitemporal objects have not been 'deleted'.

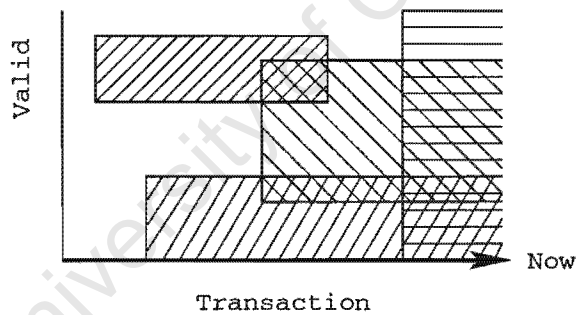

Figure 2.4: Viewing bitemporal data as rectangles

This is resolved by an approach that was proposed in [KTF98], which involves using two R-trees, one which only keeps historical data, and one which only keeps data that are *currently* valid (this 'now'-tree would only store lines that are positioned at the beginning of the transaction time interval and extends across the valid-time interval, Fig. 2.5).

In previous work when using a B^+-tree as a temporal index, the starting time and the ending time of some event would be recorded and stored either on the same or on two different nodes. A different strategy is proposed by [GLOT96] in which an interval is mapped to a two-dimensional plane, where the coordinates would be **starting time** and **duration** respectively. Temporal queries are thus made as spatial selection operations.

As an example in Fig. 2.6, a started at time 1 and ended at time 4, while c started at

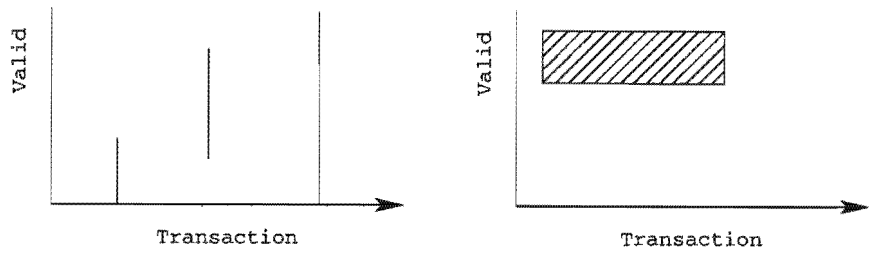

Figure 2.5: The 2-R-tree scheme uses two R-trees, one holding complete intervals (right) and one containing starting positions of intervals that extend up to 'now' (left).

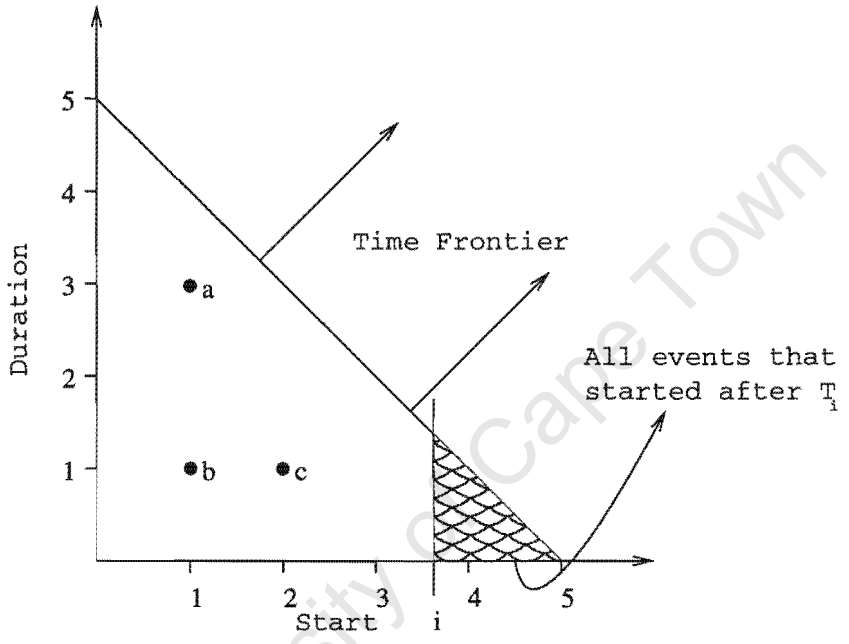

Figure 2.6: Translating time intervals to two-dimensional co-ordinates

time 2 and ended at time 3.

If a vertical line is drawn from the diagonal time frontier line to the horizontal axis at time i, then the region to the left of the vertical line will contain all events that started before time i. Similarly other queries such as "events which ended after Ti" or "events which started before Ti and have not ended" can be coloured in as regions on the two-dimensional space.

With this approach, transaction time and valid time can be 'overlapped'. Events on the interval region can be traversed either diagonally, horizontally or vertically for storage in a B^+-tree (e.g. *a*,*c*,*b*; *b*,*c*,*a*; or *a*,*b*,*c* respectively). This ordering can depend on which types of queries will be most useful, for example a vertical ordering would be faster for "What happened before $T1$" type queries.

2.2 Spatial Databases

Traditional database management systems offer a fixed set of data types that can be used to store values in columns in tables. These types are inadequate for handling spatial data, as spatial objects need to be decomposed into coordinates, and would often span over several rows. As a result, even simple queries such as finding adjacent regions are very cumbersome and inefficient. This led researchers to taking an abstract view of spatial entities and operations, and the development of spatial query languages followed.

A spatial database has been defined as a database system that includes spatial data types in its data model and query language, supports spatial data types in its implementation, and includes methods for performing spatial joins and for spatial indexing [Gő94]. Spatial databases are the underlying database technology used in Geographic Information Systems and other applications. This section briefly looks at defining spatial data types, relationships between spatial objects, and spatial access methods.

2.2.1 Spatial Data Types

There are two main aspects that need to be considered when modeling spatial data [Gő94], and they are

1. the characteristics of the space in which the spatial objects are defined, and
2. how individual spatial objects are represented in that space.

Data can be defined for a regular grid-like space which is evenly divided up into cells (raster data) or for a space that is divided into a number of dynamically defined polygons (vector data). Either way, the data that can be represented in a database will always have an upper bound on its accuracy, i.e. a mapping needs to be made from the continuous reality to a discrete representation. Examples of two approaches for ordering the space into discrete topologies are the use of *realms* as described in [GS93], and using *simplexes* and *simplical complexes* [Wor98].

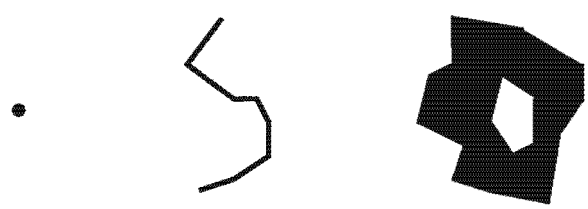

Figure 2.7: Spatial objects in 2D: point, line and region

The point, line and region objects illustrated in figure 2.7 are examples of simple spatial objects in 2D. They can be defined in many ways, but if there is a well defined discrete space in which they exist, the problem of mapping to discrete is resolved at the modeling level, rather than when the spatial database is implemented.

The distinction made in [Lan92] between objects and features are discussed in section 2.4.2 where features are described as compositions of objects, which represent entities as they are in the real world.

2.2.2 Relationships between Spatial Objects

Different classes of spatial relationships are distinguished in [Gő94], namely

- `Topological relationships:` *e.g.contains, disjoint, overlaps*
- `Direction relationships:` *e.g.north_of, behind, above*
- `Metric relationships:` *e.g.distance* $>= 100, 5$ *times larger*

Spatial data models need to offer a number of operations that deal with the spatial relationships among objects.

For one-dimensional space a complete set of 13 possible relationships between intervals have been defined in [All83].

There are many more possible relationships in two-dimensional space. The *dimension of intersection* is also considered in [CFvO93] and from a total of 256 possibilities, 52 relationships are found to be valid. They prove that the five relationships *touch, in, cross, overlap* and *disjoint* are mutually exclusive, and together with 3 operators that define the border lines of regions and begin- and endpoints of lines, the 52 valid spatial relationships can be described.

2.2.3 Spatial Access Methods

The most important function of spatial indices is to facilitate the selection of spatial objects (often with a specific relation to some given spatial value) from among a larger set of spatial objects.

Spatial indices could either be dedicated structures which are built for handling spatial data, or they could be more general one-dimensional structures such as B^+-trees for which spatial data is *mapped* to the one-dimensional domain. [Gő94]

Objects stored in a spatial index are approximated, i.e. simple geometric shapes are used to roughly define the shape and position of the object. These simple shapes are more efficient to use inside the structure for locating objects and calculating relations (rather than the actual geometry of the object).

Some indices make use of continuous approximation, which is based on the coordinates of the spatial object (e.g. bounding box or bounding sphere [KS97]), and others use grid approximations where the entire space is divided up into grid cells which contain spatial objects (or parts thereof). This is illustrated in figure 2.8.

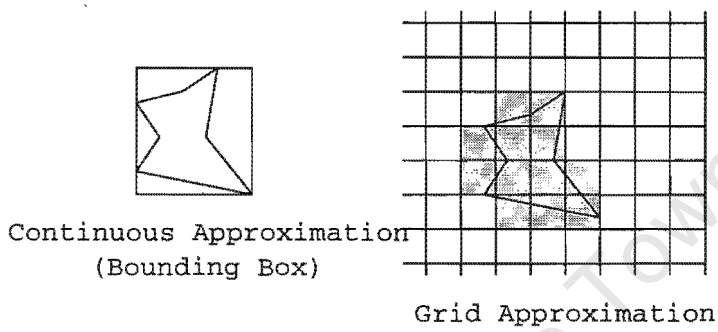

Figure 2.8: Approximations of Spatial Objects

A *filter and refine* strategy is applied, where the simple approximations are used to filter out the vast majority of data not relevant to the selection query, and the small selected subset is refined to deliver an accurate result.

Most spatial indices keep objects in *buckets* which usually correspond to pages in secondary memory. An index can either be *clustering*, where physical objects are kept on consecutive pages on disk, or a *secondary index* which only holds pointers to the indexed objects.

In the following sections, two methods for indexing point data are outlined (Kd-tree and Grid File) and one method for indexing regions (R-tree) as examples of spatial access methods.

The Kd-tree

This structure is a binary tree in which each node contains a key from one of k dimensions (used to index k-dimensional space, hence the name kd-tree).

The space is split up recursively with respect to each dimension in turn. For example the root node would split dimension 0, the next level split dimension 1, etc. When the final dimension has been split, the next level starts from dimension 0 and cycles through again. Figure 2.9 illustrates how a kd-tree divides up two-dimensional space.

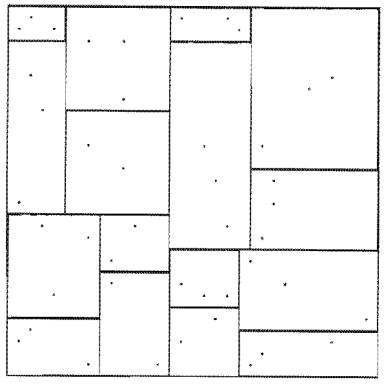

Figure 2.9: A kd-tree over two-dimensional space

Another variant called the KDB-tree [Rob81] allows for each cell in the partition to correspond to a bucket, and also uses pagination in the binary tree (index) itself.

The Grid File

The space partitioned by a Grid File is split up by an irregular grid into a number of cells. The positions of the split lines are kept in scales, and there is one scale for each dimension. The split lines extend across the entire space, which is characteristic of the partitioning used by this access method.

A k-dimensional array called the *directory* contains logical pointers that point to buckets. One or more of these pointers can point to the same bucket, depending on how many cells in the partition a region consists of (see Fig. 2.10).

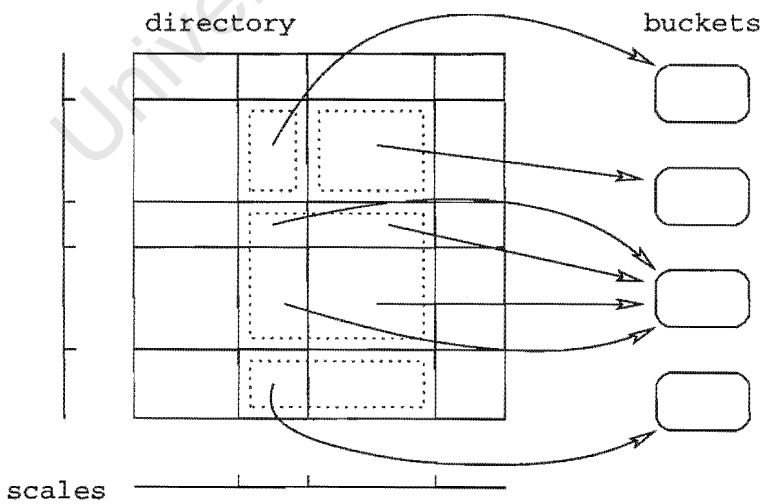

Figure 2.10: A Grid File's Structure

To look up a bucket that contains a particular point, the scales are used to find the page on disk that contains the directory entry for the cell containing the point (the directory is kept in a set of pages on secondary storage). Since the scales are relatively small structures they can be kept in memory. In a second page access the bucket containing the point is retrieved, making this a rather efficient procedure.

The R-Tree

For some set of rectangles that need to be indexed by the R-Tree (typically the bounding rectangles of more complex spatial objects), each rectangle is stored in a leaf node of the tree [Gut84]. Each key in a higher level node contains a rectangle that bounds all rectangles in its child nodes (see Fig. 2.11).

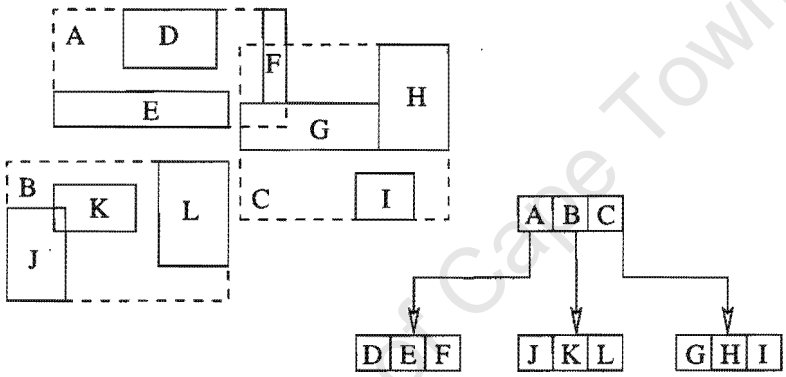

Figure 2.11: An R-Tree's Structure

This structure allows for overlapping bucket regions, but that is also the source of some inefficiency. The more regions overlap, the more paths would have to be traversed down the tree to perform some selection query. A number of remedies have been suggested to reduce overlap, and authors came up with variants of R-Trees accordingly.

The R^+-Tree, for example, uses *clipping* to completely avoid overlap. This means though that regions may have to be split up a number of times, leading to a different complexity problem [SRF87].

The R*-Tree uses a 'forced reinsert' function which periodically removes regions from the tree and re-inserts them in such a way that overlap is minimised. The R*-Tree seems to have the best performance in documented experiments [BKSS90].

2.3 Spatiotemporal Models

The data model defines the conceptual core of a system, and it consists of data types and operations, together with rules/constraints to ensure that operations produce valid results. This section presents some spatiotemporal models, which bring together some of the spatial and temporal concepts discussed earlier, and forms background for the model presented in chapter 4.

2.3.1 STDM for Object Oriented GIS

The primary focus of STDM [Wac98] is creating GIS on object-oriented systems.

The model they propose makes the distinction between `Event` classes and `State` classes, the former describing an event and when it occurred, and the latter describing what has changed or is changing.

Spatial objects are modeled as `Boundary` objects which form a vector-based partitioning of the space.

Time and space are fused together in *space-time paths* which are arrangements of `Event` and `State` object instances connected together to form a representation of an entity's lifespan.

The inheritance mechanism is used to apply incremental modification to objects as follows:

```
newObject X
  inherits oldObject Y
  includes modification M
```

In other words when a space-time path needs to be updated, the modification is added to a new class definition which inherits the existing space-time path.

The spatiotemporal representation depends on the observer's view. Their position on the space-time path will determine what is present, past, or future, and their perception of it changes as they move along the path.

This model is not aimed at being a generic spatiotemporal taxonomy, rather to provide a model sufficient for creating an object-oriented GIS system that serves as a 'proof of concept' to ideas presented in [Wac98]. The way in which inheritance is applied is quite novel, but it restricts this model to be implemented only on object-oriented systems.

2.3.2 A Generic Spatio-bitemporal Model for Geographic Information

In [Wor98] a representation is given for an object model based on topological algebra and bi-temporal elements (BTE), forming a high-level set of conceptual spatiotemporal classes called ST-complexes.

Spatial and temporal aspects of this model are first described separately and then the unified model is presented.

Bitemporal Model

Measuring events in two orthogonal temporal dimensions, a bitemporal element is a collection of combinations of transaction- and valid-time, forming regions on a two dimensional plane where the two temporal dimensions form the axes. (Fig. 2.12)

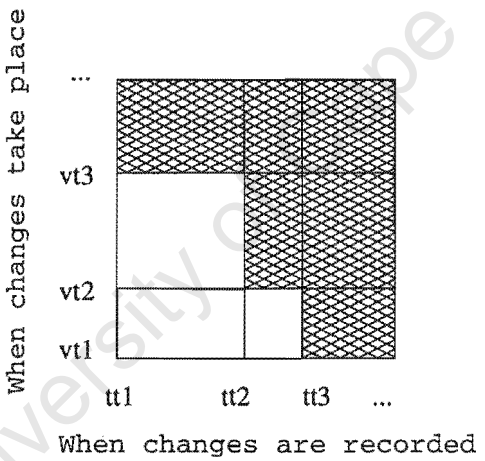

Figure 2.12: Bi-temporal element: valid time + transaction time

Spatial Model

All pure spatial objects are assumed to be in the Euclidean Plane (2-dimensional) and made up of simplexes and simplical complexes.

Definitions:

A 0-simplex is a single point, a 1-simplex is a set of all the points on a line between two endpoints (including the endpoints), and a 2-simplex is the set of all the points on the

Figure 2.13: Spatial objects defined by topological algebra

boundary of and inside the triangle formed by three non-colinear points (Fig. 2.13). The convex hull of any subset of nodes of a simplex is called a face.

A simplical complex, C, is a finite set of simplexes such that a face of C is also in C, and that the intersection of two simplexes in C is either empty or a face of both simplexes (Fig. 2.14).

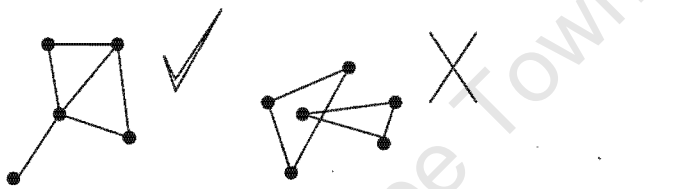

Figure 2.14: Examples of a valid and an invalid simplical complex

Representation of purely spatial objects
An equivalence relation is defined for simplical complexes such that two simplical complexes are S-equivalent if and only if they represent exactly the same spatial object. This eliminates ambiguity and the need to cater for the many distinct possibilities of representing an object with simplical complexes.

Operations on spatial objects
Some pure spatial operations that can be defined on simplical complexes are classed as set-theoretic: *union*, *intersection*, *set difference*, *set membership*, *equality* and *subset*; topological: *boundary*, *connectedness*, *inside/outside*; and Euclidean: *distance*, *bearing*, *area*, *perimeter*.

Unified Spatio-Bitemporal model

ST-simplexes, definitions
This is an ordered pair of a BTE and a simplex, for possible projections onto the spatial or temporal domain.

ST-complexes
These are groups of ST-simplexes with three types of constraints on them. Firstly, each

simplex (spatial projection of ST-simplex) must be distinct, secondly, together they must form a simplical complex, and thirdly, their faces must have as much temporal reference as themselves.

ST-objects
Just as for spatial objects, a representation can be defined for ST-complexes so that any two ST-complexes identifying the same object in 4-space (valid- & transaction time + 2D space) belong to an equivalence relation.

Spatio-bitemporal operations
Assuming that purely spatial and bitemporal operations have been defined, operations on spatio-bitemporal objects can be defined in terms of operators on ST-complexes which have been lifted through some equivalence classes.

The general model introduced here relies heavily on the theory of simplical complexes. Future work include the detailed calculation of representation of ST-operations. Also some longer term objectives are to implement and test the model, and translate some of the above constructs into constructs and algorithm that perform efficiently.

2.3.3 The "MADS" Spatiotemporal Conceptual Model

The authors of [PSZ99] aimed to provide a good conceptual model that could be used as a foundation on which to build spatio-temporal applications. They likened it to the same situation when users needed to bridge the gap between DBMS's and applications, with the aid of entity relationship models and CASE tools. There are no 'CASE tools' for designing spatio-temporal databases, because there is no agreed-upon underlying conceptual model to build it on.

Abstract Data Types for Space and Time

In order to keep the conceptual model clean and simple, i.e. not overloading the schema with unnecessary detail, abstract data types (ADT) have been defined to encapsulate spatial and temporal properties. The ADT's are organised in inheritance hierarchies that also contain set ADT's (e.g. point set, line set) and generic types (e.g. simple geo, complex geo). The generic abstract types contain simpler abstract types types like point, line, etc. and they are useful when there is incomplete knowledge about the exact shape of an object. The ADT's are extended by types appropriately defined for particular user applications.

$DBSPACE$ and $DBTIME$ are two system parameters that describe the extent of the

spatial domain and the temporal domain respectively. To represent time-varying information, there are two domains called *SPACEZONES* and *TIMEZONES*, which contain sets of all spatial elements and all temporal elements (intervals, instants, etc) respectively.

The system allocates a **spatial** and **temporal** component to every object or relationship to ensure orthogonality. Another attribute, **status**, is a function from *DBTIME* to *STATUS*, which can have values "not-yet-active, active, suspended, disabled".

Overview of Components in the Model

The MADS model is an object+relationship conceptual model, i.e. it describes objects, the relationships between objects, as well as constraints. More detailed descriptions of how these are defined are given below. The following definition is for an attribute structure. It is described as a "..higher-level ADT for an attribute." and is used for composing complex objects. An item (object, relationship or attribute) could thus have an attribute that is defined by such a structure, which could in turn contain a number of objects/attributes.

Attribute Structures

Attributes of complex objects can recursively be decomposed into other complex objects with attributes. The concept of a structure is defined to hold all the information of an attribute independently from its containing object.
Structure s = (name, cardinality, spacezone, timezone, domain), where
name is the name of the structure,
cardinality indicates whether values of the structure form a set, bag or list. If it is a time-varying structure, it indicates the min/max cardinality at any given instant in time.
spacezone holds the spatial domain over which the structure is defined, if it is a space-varying structure.
timezone the same as spacezone except for the temporal domain.
domain is the defined domain of the structure, which can either be elementary if the structure is atomic (int, string) or complex otherwise.

Object Types

An object type can either be spatial, temporal, both, or neither. It is the item used for storing both attributes and information.
Object type O=(name, geometry, status, attributes, methods, domain, super, pop),

where
geometry is a spatial attribute for which its domain is specified as one of the spatial ADTs,
status is a time-varying attribute belonging to the domain of the function from $DBTIME$ to $STATUS$,
attributes is the structures of the (set of) attributes and inherited attributes (which may be empty sets). Inheritance and redefinition of attributes allows for useful scaling of geometries, or keep information about the lifecycle of objects.
super is the set of supertypes of O (could be empty),
pop is the population of the object type. Each object has an object-id and for temporal objects the population is time-varying.

Relationship Types

R=(name, kind, roles, geometry, status, attributes, methods, domain, pop),where
kind is the type of relationship which consists of a tuple (type, constraint, membership). The type field indicates what type of relationship it is, e.g. it may be a spatiotemporal relationship such as AlwaysInside or SometimesCross. The constraint field is an optional predicate over the geometries or the histories of the objects linked by the relationship, and membership is a boolean flag indicating whether tuples of objects satisfying the constraint also defines relationships of this type.
roles defines the types of objects involved, their cardinality and collection type (e.g. set, bag or list), and role names, which should always be unique.

Constraints

Spatial constraints between objects and their spatial attributes are not always necessary, although they should be easily defined. Time constraints however, follow certain assumptions:

- An attribute can only be valid during the lifetime of the object it belongs to.
- A complex attribute's lifespan is the union of its components' lifespans.
- A relationship is contained in the intersection of related objects' lifespans.

An existing implementation of the framework, MADS, does not implement any of these constraints as it can be limiting in expressive power.

2.4 Time in Geographic Information Systems

Geographic Information Systems (GIS) have been around for many years. GIS can basically be described as specialised databases systems used for the storage and analysis of geographical data.

This section comments on the relationship between time and space, with particular relevance to GIS. Some issues such as 'identity' and 'scale' are raised and a number of simple ideas that have been suggested for giving histories to cartographic data are presented.

2.4.1 Cartographic Time Model

As briefly mentioned in section 2.1, there are many different views that scientists and philosophers had on time. Rather than turn to philosophy or try to represent things totally accurately as they are in reality, only essential characteristics of time are distilled, to have a generic and pragmatic representation of spatiotemporality. This is referred to as 'cartographic time' [Lan92].

Cartographic time is considered a fourth cartographic dimension. Dimensions are inherently spatial constructs, but working with time alone is easier than working with space alone, because time is considered one-dimensional. As in Newton's theory, cartographic time is recorded separately from space, although temporal and spatial dimensions do interact. Individual models can then be designed to determine how the interaction between absolute spatial and temporal dimensions takes place.

Temporal boundaries are formed when two adjacent states differ, i.e. where change takes place. Temporal objects are in many ways parallel with spatial objects, as shown below:

	in space	in time
Overall configuration	map	state
Separated by	lines	events
Sampling units	cells	hours/days/decades...etc
Meaningful units	objects	versions
Separators between units	boundaries	mutations
Size measured by	length, area	duration
Position described by	coordinate	date
Contiguous neighbours	adjacent objects	previous/next versions
Max. num of neighbours	infinite	2

Cartographic maps have 'states'. States are made up by some configuration of temporal objects. Each object has a particular version, which can be changed by 'mutation'. When

any one or more objects mutate, it is an 'event' which causes the map to change state. This is illustrated in figure 2.15.

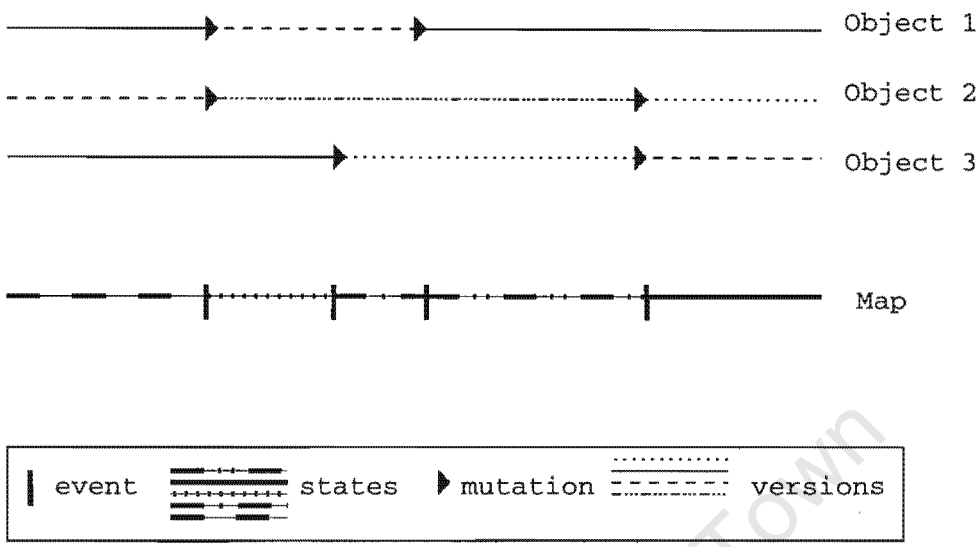

Figure 2.15: The states of cartographic maps (from [Lan92]).

Other issues that researchers have identified when dealing with time in GIS relate to multi-faceted time, identity, and scale.

Three Facets of Cartographic Time

Change at each of these levels are recorded as world time, database time and display time respectively, where world time and database time are more commonly referred to as valid time and transaction time. [SO95], [KIPS95], [Lan92]

Constant Identity

Identity which defines entities is established when they are born or reincarnated, or when they die. There is some granularity of change involved here to address the question "After how much change or what kind of change does identity get altered?" It is therefore important to define which elements of an entity make up its identity [CP00].

Scale Independence of Spatiotemporality

Rates of change are different for different entities. It is suggested that temporal information should be captured and stored in the finest resolution required (similar to spatial

information). A poor choice of resolution or 'scale' can cause misleading information , for example making it seem as though two objects co-existed at the same time, while there is a small time gap between their lifetimes. Several theories suggest that temporal and spatial scale are loosely associated, and that the two be scaled in tandem (something like room/minute, city/day, region/week). There is still poor understanding of the mechanisms relating phenomena at different scales - as evidenced by Chaos science [Kel94].

2.4.2 Representing Features

A GIS distinguishes between objects and features. Features are semantic groups of objects – for instance Washington state is made up of a main region on the continent plus a number of islands. A park can include roads, camp sites, etc. but all together constituting a feature. An object in this text is considered to be a fixed fragment of space, of which non-spatial attributes can change over time. It is in effect referred to as a location. [1]

Temporal Topology of Objects versus Features

Features can be born, move over time and die, whereas objects cannot since they are bound by their location. Objects fill their spatial as well as their temporal partitions exhaustively. Features do not fill their spatial partitionings, and their lifetimes are also independent of temporal partitionings. An object's lifespan can be represented by a sequence of changes in its attribute values, where the lifespan of a feature includes attribute changes, as well as updates with respect to shape and location.

A Representation Strategy

A complete set of all objects over time can be stored, and a feature-based structure can be composed from it. An object based structure is thus complementary to the higher level feature based structure. A feature at time Ti can be seen as a time-slice of all objects composing that feature at that time. Object level representation includes all nodes, chains and polygons which comprise objects over time, while feature level only shows entities included at given time slices. Features at different time slices are likely to intersect, but this is important for error checking and change detection.

[1] According to the American National Standard, **feature** = defined entity and its object representation, **object** = digital representation of all or part of an entity, **entity** = real-world phenomenon that can not be subdivided into phenomena of the same kind [Lan92].

2.4.3 Conceptions of Time

Some common viewpoints on how to treat cartographic time are discussed in this section.

Space-time Cube

Some researchers describe a 3D spatiotemporal model where two dimensions are spatial and one temporal [MWH]. Operations on such a cube would be complex (referencing a point, tracing a vector, cross-section, trimming) and it would become increasingly more difficult as the volume of data grows. It is still too taxing on existing hardware, and there are still too many questions and problems that need to be resolved with this approach, making it impractical for GIS over the next few years.

Sequent Snapshots

Snapshots can describe what exists at time T1, and at time T2, but in order to find out how they differ, the two snapshots must be compared exhaustively. This approach capture states, but not events. The underlying meaning of objects are hidden, therefore it is also not possible to enforce any type of rules for integrity. Much redundant storage is used, because much remains unchanged from one time to the next.

Base State with Amendments

Descriptions of changes are stored, not whole snapshots, thus greatly reducing storage requirements. This springs from a vector-based philosophy, where events are stored as they occur, not at regular sampling intervals. Errors can be trapped as improbable events can be identified, and a temporal structure is present, unlike the snapshot model (previous and next forms of a version can be easily found). However it is then costly to obtain a complete picture at any point in time other than the base state (typically either 'now' or 'the earliest recorded state').

Space-time Composite

With this approach 3D space and time are collapsed onto a 2D plane where objects are decomposed from their surrounding environments when they have changed. Each component of the composite has its own history of changes associated with it. To compose a single time slice from the composite, the history list of each component is traversed until events are found which fall within the required time slice. One side-effect of this approach

is that decomposition can also pose storage problems if the composite is decomposed into too many small bits. Also, if an object is updated and split into two, that object's identifier must be replaced by one or both of the new identifiers throughout the entire database, wherever it has been referenced.

2.4.4 New Trends in GIS

GIS is an important tool used by governments and other large organisations that need to analyse vast quantities of spatial data. Naturally, the technologies used in such systems have been developed and improved over time. However, since a GIS requires a number of different technologies, they have become very large, expensive, complex systems.

The databases in which spatial information is stored have undergone numerous improvements. Not only has hardware efficiency increased almost exponentially, but there has also been great changes in software engineering. A new trend in GIS technology is to have open systems that can have data transported easily from one platform to another. Interfaces are being built for legacy systems to connect to the new, more modularly designed systems.

As the large GIS systems are broken up into modules, the data that is transferred between modules are required to be more independent. And as the data is "emancipated", smaller systems can be written to use and analyse the geographical data. Therefore it can be argued that the big complex GIS systems of the 20th century will soon disappear before a multitude of small network-based applications that operate on data that is independent of any particular application or platform.

There are many new mainstream technologies such as XML, SVG (Scalable Vector Graphics), and VML (Vector Markup Language) that is ideal for using in modular GIS systems. [Hub00], [AK00]

Internet technology, which is relatively uncomplicated, would not have been used by so many people had it not been for its simplicity. With the number of mobile devices increasing rapidly (cellular phones, PDAs) the demand for 'personal GIS' is also increasing. In order for personal devices to be *geographically enabled*, the technology must be simple and lightweight, and the geographic data sent and received must be application independent.

Heavyweight GIS can then be focused on analysis of spatial information, with the aim of aiding decision-making at a high level. Where previously information from various sources had to be integrated in the minds of decision-makers, GIS could aim to perform such integration through a simple, yet powerful framework.

But whether using a large complex system or a small distributed device, there are nu-

merous potential problems with recording and analysing spatiotemporal data. For small devices, storage space could soon become a problem, and for large systems, querying huge volumes of data may be slow and inefficient. What is needed is a simple and effective way to write systems that can answer questions about what happened where and when. It would also be ideal if such a technology could be used for a wide spectrum of systems ranging from large to small.

Chapter 3

Persistent Object Systems

The use of persistence technology can greatly enhance the development of complex applications such as GIS, CAD and CASE systems. It facilitates the process of storing data and thus saves the application programmer time and effort that would have been spent on building a storage facility into the application. It has not taken off commercially however, partly because little experimental data is available on the use of persistent application systems (PAS). In [AM95] Atkinson and Morrison wrote:

> "The pay-off from investing in good persistent technology is only apparent when a large application is used over a long period. Hence it is difficult to properly evaluate this approach without large investment in building up a realistic load and writing a reasonable volume of application software ... "

This thesis aims at facilitating the construction of GIS systems through the use of persistence technology. This chapter gives a brief background on orthogonal persistence and an outline of previous experiences that suggest there is a need from some structure when building complex spatiotemporal applications using persistence.

3.1 Persistent Systems and their Advantages

When working with large data sets, application programmers have the responsibility of ensuring that whatever application they design has the ability to store information for extended periods of time. In other words, whether information is stored in a file or in a database, that functionality has to be added by the programmer.

In most cases this requires that the programmer understands at least two different models - one for writing the application, and one for storing the data. There are three kinds of

mappings which has to take place in such a two - tiered architecture. One is designing the program to interact with the real world in a particular way, another is the process of designing the database to store data that represents the real-world situation, and a third is the interface between the application and the database (Fig. 3.1).

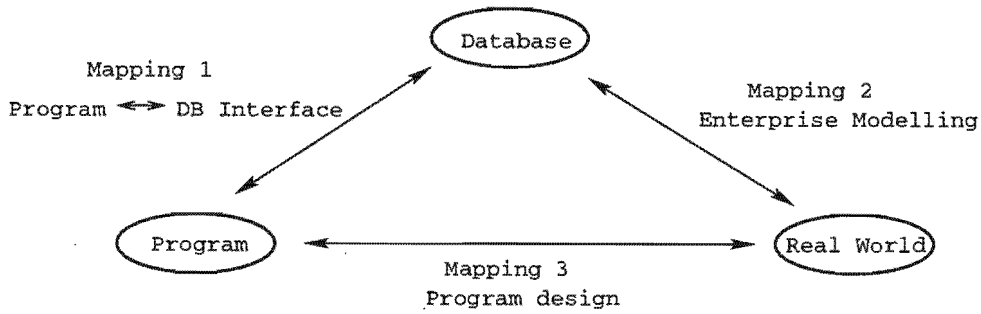

Figure 3.1: A Two-level System with 3 Mappings

A persistent system eliminates the need for a separate database facility. Objects in a persistent system have the ability to automatically retain their state for long periods of time - past the execution time of the program. The programmer does therefore not have to worry about explicitly storing data for an application. As a result, the programmer only needs to manage one mapping - from the real world requirements to a suitable application (Fig. 3.2).

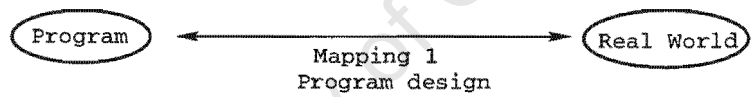

Figure 3.2: A Persistent System with 2 Mappings

Persistence can be applied to any kind of data, including the source code of programs. Since some parts of a program can be made persistent, it is possible that one can write code which contains both text and links to persistent values. Such a program is called a *hyper-program* since it is no longer flat in structure [ZKM00]. Some benefits of hyper programming are that associations between source and compiled versions can be enforced, and that access path information for different components does not have to be known or explicitly stated.

3.2 Design Criteria for Persistent Systems

As mentioned above, using a subsystem for data storage such as a file system or database, involves more complexity than is found in a well-designed persistent system. It was necessary to have some criteria against which to evaluate the design of a persistent programming language, and the following design principles are given in [AM95]:

- **Persistence Independence**: A program has the same form regardless of whether its data is retained past execution time. It looks the same whether it manipulates long-term or short-term data.
- **Data Type Orthogonality**: All data types can have the full range of longevity, and there should be no special cases where any data types are not allowed to be long-lived.
- **Persistence by Reachability (Transitive Persistence)**: Objects that are reachable from persistent *roots* will have longevity and are not explicitly declared as persistent. Thus the manner in which persistent objects are identified and provided in the system is unrelated to the type system.

These rules are aimed at ensuring that an *Orthogonally Persistent* language (i.e. one that applies all three rules) is both easy to use and powerful enough to free the application programmer from having to explicitly manage data storage.

3.3 Orthogonally Persistent Java

PJama [AJDS96] is an orthogonally persistent version of the Java programming language designed according to the three principles mentioned above. No changes have been made from the original Java language specification, only the Virtual Machine has been modified to enable objects to persist. The version used in this thesis (PJama 1.6.4) has persistent roots implicitly defined as static attributes of classes. Using this more subtle definition of persistent roots, it is easier to port the code written for a persistent application to run on any other virtual machine in a non-persistent environment.

The PJama Virtual Machine makes a distinction between two modes of executing a class file, namely initial execution, or resumed execution. If the class file had never been loaded before, it would have an initial execution after which its current state is stored on the persistent store. Every time it is subsequently run, the class will be resumed from the store.

For large systems it would be necessary to run more than one Virtual Machine or multiple threads of a virtual machine on a persistent store in order to provide various users with simultaneous access to data. Therefore the OPJ specification [AJDS96] states that Virtual Machine transactions (those units of OPJ computations between any two suspended computations) should conform to the standard ACID transactional properties.

The OPJ specification also allows for evolution tools to be used either on-line or in an off-line mode (i.e. when the store is not being accessed). Such tools are used to make structural changes to classes that are already persistent in the store. This is particularly

useful with the development of a persistent application, during which the structures of classes are likely to be updated regularly.

3.4 Previously Recorded Experiences with Persistent Java

The kinds of environments in which persistence has been used and tested are illustrated in this section.

3.4.1 Persistent Java GIS Project at the University of Glasgow

A project called JUGGLE, conducted at the University of Glasgow, was aimed at investigating the feasibility of using Java for developing GIS, and to measure how easy it is to integrate Java with legacy systems [MIA96].

In an initial phase an application was developed that could display road carriageway data of the Ordinance Survey of Great Britain (OS(GB)). A working prototype was developed within two weeks and it was found an easy task, even though the developers had no prior knowledge of Java. The OS(GB) data was then stored in a persistent store which resulted in great improvements in performance since data did not have to be read in a different format and converted to Java objects – it was kept in the store as Java objects. Converting to a persistent store was also accomplished quickly and efficiently [MIA96].

Integrating Java with legacy systems was found to be more difficult though, because Java could not be used as an embedded language. What the authors did instead was to write native methods that called existing code from within a Java program. It was not the optimal solution they had intended, but they succeeded in reaching the goals of this phase which were to link Java code with the legacy code, and to create "mirror classes" in Java that could emulate the behaviour of the legacy code.

The conclusions reached from this project were that Java is a language well suited for the rapid development of complex applications, and that its strong type-checking and error-handling facilities together with persistence creates a powerful development platform.

3.4.2 N-dimensional Indexing in Persistent Java

In a different project [VO97], conducted at the University of Cape Town, a Persistent Java implementation was created of the GiST (Generalised Search Tree) structure, a

generic indexing structure developed at the University of California, Berkeley [HNP95]. The project started with the porting of basic GiST classes from C++ to PJama. Afterwards a B^+-tree and an n-dimensional R^*-tree were implemented using these classes, and the R^*-tree was used to index geographic data on informal housing settlements in the Cape Town region. A GUI was also developed for viewing the data and the index structure. The resulting library was a generic indexing tool that could be used for various types of indexing in complex applications, such as GIS. The development of indexing methods using the PJama GiST library is aimed at reducing the time required for building GIS applications, which rely heavily on efficient spatial indexing mechanisms.

All the code was developed using the SUN JDK version 1.1.6 and it was converted to Persistent Java only during a final stage. This conversion was achieved through the addition of only a few lines of code and was found to be remarkably easy. The rest of the code that ran on JDK 1.1.6 did not need to be changed or amended at all.

3.4.3 Comparison between Persistent Java and a Relational Database (VIBES Project)

A subsequent project at the University of Cape Town investigated the relative benefits of using the PJama GiST library for building a spatiotemporal application, and compared it with the use of a relational system [SVZ98]. Two systems were simultaneously implemented for querying oceanographic research data of the South African Sea Fisheries Research Institute (SFRI), one using a relational database, and the other using the PJama GiST library.

It was found that Persistent Java was much slower with bulk data loading routines, however query times were significantly faster. Of greater concern than performance, however, was the large amount of time required to develop each system (relational & PJama). In their conclusion the authors wrote that the relational approach was preferable because it uses a well-structured query language (SQL) and the design of a database in the relational format is much easier than object-oriented design. The developer that creates a system using persistent Java not only needs to have command of the Java language, but must also implement all structures in which data will be stored, and all routines used for querying the data. As a result of this experiment the need for a package to facilitate persistent GIS construction in PJama was clear.

3.5 Conclusion

Persistence is directed towards a simpler, less complicated development environment, and combined with a robust and high-productivity language such as Java it holds great poten-

tial for the fast creation of complex applications. The persistent Java environment does however lack any form of data structure as can be found, for example, in the relational data model. Although a much richer variety of data structures are possible in Java, there are no guiding principles for application programmers on how to go about organising their data. This brings about the need for libraries containing tools and structures that can form frameworks on which persistent applications are built.

Chapter 4

Spatiotemporal Model

In chapter 2 some background was given on a few spatiotemporal models and on the related technologies of temporal and spatial databases. The spatiotemporal (ST) models that were presented are quite simplistic in their type structures, for example the model of [Wor98] (section 2.3.2) contains only a single spatiotemporal type, the ST complex. The models were designed for different purposes e.g. [Wac98] in section 2.3.1 was created specifically for object-oriented GIS.

The model chosen for this project is that of [GBE+00] because it is built on a conceptually clean foundation with enough expressive power and not too much complexity. The model was designed with the intention of creating a sound foundation for building spatiotemporal capability into a DMBS. It can be used to develop a simple yet powerful language for querying spatiotemporal data.

A distinction is made between continuously moving objects and discretely moving objects in [GBE+00]. Discretely moving objects, like land parcels for example, change at instants with a certain amount of time elapsing between changes. It is argued that it is easier to implement systems that model *discretely moving objects* since spatial and temporal values can be stored in separate columns in relational tables.

Systems that model *continuously moving objects* however are harder to implement, and that is the aim of the model [GBE+00] presented in the rest of this chapter. The implementation of the model in Persistent Java is discussed in chapter 5.

The description of the model is structured as follows: in the next section, their approach to developing spatiotemporal models is discussed, and in section 4.2 the basic structure and design goals are explained. Sections 4.3 and 4.4 define the type system and operations respectively, and finally some illustrative examples of how the model is used are given in section 4.5.

4.1 Abstract vs. Discrete Modeling

Before the model in [GBE+00] was developed, some of the authors had proposed an approach to creating a model for moving objects in [EGV97]. The question addressed is whether a suitable approach for handling spatiotemporal data is to add spatial data types to a temporal database, or vice versa. They make the observation that a temporal database is mainly used for storing *facts*, and that by adding *system-maintained* spatial attributes, one would have a system for storing facts of which their truth may vary over space as well as over time.

Adding spatial types as *user*-defined types would result in a system that is more or less the cross product of temporal and spatial databases. It is a reasonable model and suitable for describing stepwise constant values (discretely moving objects), but it is too limited to describe continuously moving entities.

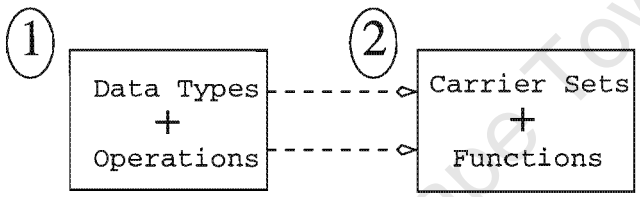

Figure 4.1: Two steps for designing a model

When designing a data model with spatiotemporal types, two steps must be followed:

1. An appropriate group of types and operations must be chosen, which seem to be suitable for querying. This makes up the *signature* of the model.
2. Carrier sets must be defined for the data types and functions for the operators (see Figure 4.1). A carrier set for type D contains all the possible values for D. Functions are the mappings between carrier sets. This forms the *algebra* associated with the model.

There are two different levels of abstraction for making the above decisions: the model can either be designed as *continuous* or as *discrete*.

It is common sense that only discrete models can be implemented because we can only represent a finite number of numbers in computers. It is thus the appropriate level of abstraction to concentrate on.

However continuous models are relatively simple to define, and they can easily be conceptualized. Also, designing at the discrete level may lead the designer to prematurely discard certain design options. For example it may be decided that moving reals are too problematic to represent in a discrete model, but there is still the need to represent the

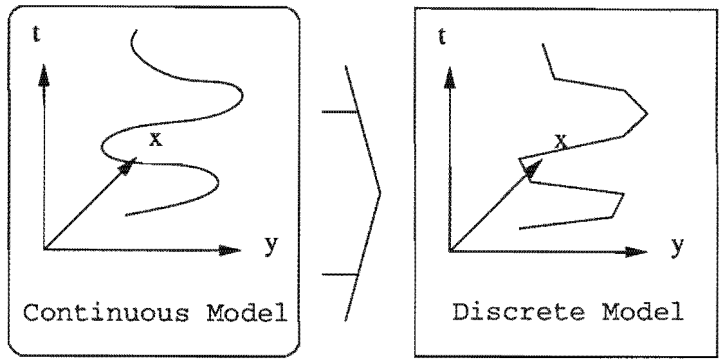

Figure 4.2: First create an abstract model, and use it as a target when creating the discrete model.

distance between moving points or the size of moving regions. This leads to the inclusion of a large number of simpler discrete operations (mindistance, maxdistance, minsize, maxsize, etc..) which is bad for the design of the query language.

For this reason the authors of [EGV97] suggest in their design methodology that an abstract, continuous model be developed as an initial step, and that a discrete model be derived from it, as shown in Figure 4.2. Once the carrier sets and functions for the discrete model are defined, they can be translated into data structures and algorithms for implementation.

The rest of this chapter presents the abstract, continuous model as defined in [GBE+00].

4.2 Requirements, Structure, and Design Goals

The most fundamental requirement for the model is to provide a means of recording and querying spatiotemporal data, in particular moving points and regions. The structure that was chosen for the model is based on traditional data types (int, real, string, boolean) together with some basic spatial types (point, line, region) and a temporal type (instant).

An important design goal was to have orthogonality in the type system. This meant that the same type constructor could be applied uniformly to any of the basic types included in its signature. The other main design criteria were closure, and genericity and consistency among operations.

The idea of *closure* means that any operation performed on any type in the model, will return data which is of some type already in the model. An important operation for example is to measure the distance between two points. It will be returned as a real number, therefore the model has to contain a type that represents a real number. But if either of the two points are moving, the distance between them will start to 'move' as

well, hence a type for 'Moving Real Numbers' also needs to be included. When a point moves for an arbitrary distance, its trajectory forms a line. Therefore a spatial type for a line was also included. Similarly the movement of a line forms a region - another spatial type in the model.

Genericity implies that an operation (e.g. overlap) should be executable on as many types as possible. For example, **overlap** should not only be an operation restricted to spatial regions, but rather apply to all spatial types (e.g. **overlap** between a line and a point) as well as to *range* types (e.g. overlap between two ranges of integers).

Consistency means that whenever an operation does something on one type, it can be expected to do something similar on a different type. For example, the **isempty** operations returns a boolean value signifying whether an object has any elements in it. It should behave the same regardless of whether the object is a *spatial* or *range* or *moving* object.

4.3 The Type System

The basic type system is shown in table 4.1. The signatures on the right describe how some types can be used in the construction of other types. For example the *range* type can be constructed using any of the $BASE$ or $TIME$ types, therefore all the possibilities would be *range*(*int*), *range*(*real*), *range*(*string*), *range*(*bool*), *range*(*instant*).

Type constructor	Signature	
int, *real*, *string*, *bool*		$\rightarrow BASE$
point, *points*, *line*, *region*		$\rightarrow SPATIAL$
instant		$\rightarrow TIME$
moving, *intime*	$BASE \cup SPATIAL$	$\rightarrow TEMPORAL$
range	$BASE \cup TIME$	$\rightarrow RANGE$

Table 4.1: Type signatures for spatiotemporal model

The $BASE$ types are self-explanatory. Although the model does not require the *string* type for closure, it has been added by Güting *et al* for usefulness. All database systems support a string type, and it is often necessary to add comments or other information to data in the model.

Each of the $BASE$ types also have the *undefined* value ($\perp$) included in their carrier sets.

The *range* type represents *sets of intervals* of the $BASE$ and $TIME$ types. It is applicable to one-dimensional types on which a total order exists (*int*, *real*, etc.) The need for range types arises from the requirement that projections be available onto the domain and range of moving types. The total lifetime of a moving object, for example, would be expressed as a range of *instant*.

By definition, a *range* is comprised of a finite set of non-adjacent[1], disjoint intervals, where intervals can be open or closed at either end.

Since ranges over the time domain is of particular interest in this model, it is given a special name: *periods* (synonymous with *range*(*instant*)).

Sets of *spatial* types do not need to be grouped by the *range* type because a single *spatial* object can consist of multiple disjoint elements. For example a *line* and a *region* can have more than one separate part, which means that a region and a set of regions are represented by the same type. Also, a set of *point* objects can be expressed as a *points* type. The *spatial* types are illustrated in Fig. 4.3.

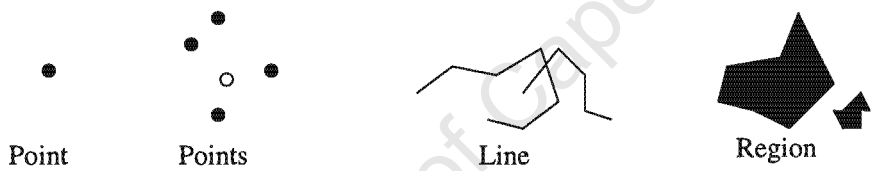

Figure 4.3: Spatial types

The *point* type can either have a value which represents a point in the Euclidean plane, or it can be undefined.

A *points* value is a finite set of points. The reason why there are both a *point* and a *points* type, is that the model requires representation for single values as well as for sets/ranges of values. Even though a single point can be represented with the *points* type, it was decided to include *point* nevertheless since modeling the movements of a point was one of the fundamental aims for developing the model.

The *line* data type consists of a finite set of simple continuous curves in the plane. These curves are defined in such a way that their intersections yields a finite set of points that can be represented by *points*.

A *region* consists of a finite set of *faces*, where a *face* is described as a non-empty, regular closed set. This means that a *face* can not have misplaced lines and points that cut or puncture the region, or that dangle loosely (see Figure 4.4). The *faces* that make up the *region* must be disjoint, but they can have holes in them, and it is allowed that

[1]Two disjoint adjacent intervals are considered as a single interval.

a face is contained inside the hole of another face (islands).

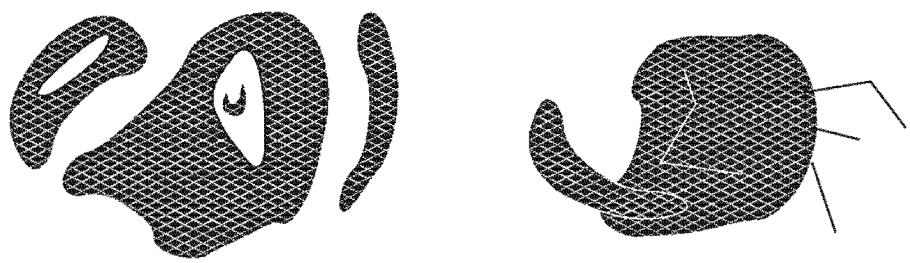

Figure 4.4: Illustration of a valid *region* (left) and an invalid *region* (right)

The *instant* type represents an instant or point in time. Like the other *BASE* types it could either have a value or be undefined. Time is viewed as similar to the real numbers in that it is continuous and linear.

BASE and *SPATIAL* types are used by the *moving* constructor to derive their corresponding temporal types. The resulting temporal type is a mapping from the time domain to the domain of the type being used. Each value in the carrier set of the moving type is a function (continuous component) which describes how a value from the base type developed over time. Thus the *moving* types consist of a finite number of such continuous components. This ensures that the projections of moving objects also only have a finite number of components, which is necessary for the design to be implementable.

The temporal types are infinite sets of (instant, value) pairs. For practical purposes the *intime* constructor was also included to represent a single (instant, value) pair. The *intime* types are useful for representing the results of operations that yield the value of the temporal type at some given instant in time. The types that are of main interest for the designers and users of this model are the spatiotemporal types, which are a subset of the temporal types where the 'values' associated with instants are spatial objects.

4.4 Operations

The operations have been divided up into two sections, namely temporal and non-temporal operations. In their design, operations have been made as generic as possible, and another aim was to achieve consistency between non-temporal and temporal types. The following section discusses operations that form part of the kernel algebra, and section 4.4.2 shows how they have been 'lifted' to the temporal domain.

4.4.1 Operations on Non-Temporal types

Table 4.2 lists the operations on the non-temporal types of the model. The *kernel algebra* is defined as the *BASE* and *SPATIAL* types, together with all the operations in table 4.2. Some of the operations only work on certain types, e.g. **min** and **max** are restricted to 1-dimensional ordered types (*int*, *real*, *time*). However an operation like **isempty** can be performed on *all* types, because all types include the *undefined* value. With an operation such as **isempty** the goal of genericity is successfully attained. Each category in the table is discussed in turn below.

Category	Operations
Predicates	**isempty** **equals,not_equals,intersects,inside** **less_than,greater_than,le,ge,before** **touches,attached,overlaps,on_border,in_interior**
Set Operations	**intersection,union,minus** **crossings,touchpoints,common_border**
Aggregation	**min,max,avg,centre,single**
Numeric	**no_components,size,perimeter,duration,length,area**
Distance and Direction	**distance,direction**
Base Type Specific	**and,or,not**

Table 4.2: Classes of Operations on Non-Temporal types [GBE+00].

Predicates

Both unary and binary predicates have been considered for the model – the only unary predicate being **isempty**.

For binary predicates the possible relationships were considered between two single values, two sets, and combinations of a single value and a set. The predicates were chosen according to set theory, order relationships and topology.

There is some redundancy in that the operators **less_than**, **greater_than**, **less_equals**, **greater_equals**, **not_equal**, **equals**, are not all needed (only **equals** and **less_than** are sufficient), however in order for the language to be more expressive they were all included – there is usually some trade-off between conciseness and ease of use.

Set Operations

Set operations were designed for all set types, even for some single values (for instance **intersection** and **minus** could be applicable to single values as well). When designing the set operations there were some conflicting arguments in terms of the closure of sets. On the one hand, a closure operation is needed for two-dimensional sets (spatial types), since by their definition, they have to be closed (i.e. a region must include its border). For example when the **minus** operation is applied on two regions, the result must include the values along the border where the regions used to overlap (Fig. 4.5). On the other

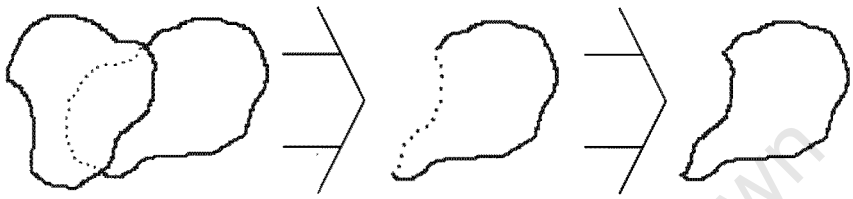

Figure 4.5: Closure needed for 2D set operations, e.g. **minus** in this case.

hand, one-dimensional sets are allowed to contain open-ended intervals (see Fig. 4.6). In the case where a time-changing function followed a stepwise evolution, the value of the function changes at the beginning of the new time value, and thus the previous interval has to be open-ended. Some specialized intersection operations were also defined as

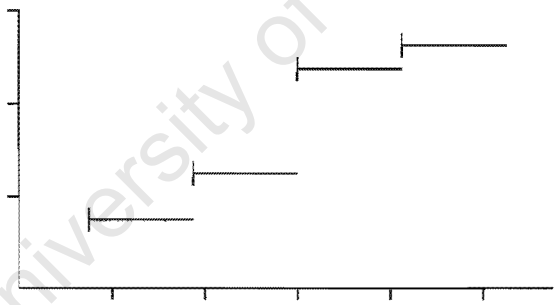

Figure 4.6: Open-ended intervals are needed for stepwise functions.

crossings (between two lines, it returns a *points*), **touch_points** (between two regions or a line and a region), and **common_border** (between two regions, returning a *line*).

Aggregation

The aggregation functions reduces sets of values to single values. The **min** and **max** operations are only applicable to one-dimensional types with a total ordering, and **avg** can only be used on numerical types. The **center** function is for 2D sets, and **single** is used to cast sets that contain only one element to a single value.

Numeric Properties of Sets

For the operations on the numeric properties of sets, there are a number of aliases for the **size** operation. On continuous 1D types, **size** returns the sum of the lengths of all the continuous intervals. The size operation for *periods* was renamed **duration**. For *line* it returns the length of the line (length), and for *region* its area is calculated (area). Additionally there is also a **perimeter** operation for *region*. The number of disjoint components is given by the **no_components** operation, e.g. the number of faces in a *region*, or intervals in a 1D set.

Distance and Direction

An interesting operation is **direction**, which takes two **points** as argument, and returns the direction of the line between them in degrees (0..360). This way, predicates related to direction can be expressed (e.g. 'north' is where direction = 90).

All continuous types have a **distance** operation. In one-dimensional space it would be the absolute difference between two points, and in two-dimensional space the Euclidean distance.

Base Type Specific

Some operations are specific to the *BASE* type *boolean* (**and**, **or**, **not**), but they needed to be included in the kernel algebra.

4.4.2 Operations on Temporal Types

Temporal types are *BASE* and *SPATIAL* types that move over time. All the operations applicable to non-moving types had to be made applicable to moving objects as well – this is discussed under 'lifting' later in this section. In addition, operations were necessary to distill the temporal or the value ranges from the moving types in order to answer queries like 'During which times did the moving object exist?'.

Table 4.3 lists the temporal operations, and each section is discussed below.

Projection to Domain and Range

The times for which a *moving* function is defined is returned by the **deftime** operation.

Category	Operations
Projection to Domain/Range	**deftime,rangevalues,locations,trajectory routes,traversed,inst,val**
Interaction with Domain/Range	**atinstant,atperiods,initial,final,present at,atmin,atmax,passes**
When	**when**
Lifting	(inferred operations)
Rate of Change	**derivative,speed,turn,velocity**

Table 4.3: Classes of Operations on Temporal types

Various operations are available for returning the values that a *moving* had assumed during its lifetime. The range of values assumed over intervals in 1D space is given by **rangevalues**. A *moving(point)* or *moving(points)* projected onto a plane can either yield a *points* or a *line* type, produced by **locations** or **trajectory** respectively. A *line* is obtained by the projection of a *moving(line)* with **routes**, and a *region* by projecting either a *moving(line)* or *moving(region)* with **traversed**. For example if a straight horizontal line moves upward, then the **traversed** operation would return the rectangular region that marks its path.

There are two simple projections for *intime* types, **inst** and **val**, which returns the time and value components respectively.

Interactions with Values in the Domain and Range

atinstant, **initial** and **final** return single (instant, value) pairs relating to an arbitrary instant, or the first or last value in the moving type respectively. The **present** operation checks whether a *moving* existed at a certain instant or time interval, and **atperiods** returns a subset of the *moving* for a given set of time intervals.

Where **atinstant** and **atperiods** selects a subset of the moving history relating to specific times, the **at** operation returns a subset based on a value or a range of values. Also, **atmin** and **atmax** return subsets at the extreme values in the 1D domains. To check whether a *moving* type ever attained a particular value, such a value can be passed to the **passes** operation which would return a *boolean* value.

The when Operation

In [GBE+00] there is speculation about whether this operation can be implemented, but it is mentioned here for the sake of completeness. The **when** operation takes as its argument some *predicate* based on some attribute of the moving type. Using this

operation, one could restrict the *moving* type to when the predicate yields a certain value. For example, a query like "Return all the values of moving line **ml** when its length was no longer than 50" could be expressed as

```
ml.when( [ml.length().less_than(50)] )
```

It is a conceptually simple operation, but certainly in the continuous domain it wouldn't be implementable since the predicate would have to be tested over an infinite set of values.

Lifting Kernel Operations

Lifting the non-temporal operations to being time-dependent is the key to achieving the goals of consistency and closure explained in section 4.2.

It means that each of the operations defined in the kernel algebra (4.4.1) is amended such that any of its arguments can be moving types, which would result in the return type also being a moving type.

For example, where **distance** could be performed with *line* and *point* arguments:

$$line \times point \rightarrow real$$

it is lifted to also include the signatures

$$moving(line) \times point \rightarrow moving(real)$$
$$line \times moving(point) \rightarrow moving(real)$$
$$moving(line) \times moving(point) \rightarrow moving(real)$$

Some of the existing operations that did not seem very useful become far more interesting once they are lifted. For example the intersection of a *point* and a *region* seems less useful that the intersection between a $moving(point)$ and a *region*, or a $moving(region)$ and a $moving(point)$.

It was also found that after lifting operations, there are cases of more than one operation performing the same task, such as **at** and **intersection**. Also, the **when** operation could be expressed differently by using other lifted operations. The example used previously can now be expressed as

```
ml.atperiods(ml.length().less_than(50).deftime())
```

The lifted **length** operation returns a *moving(real)*, which is restricted by the lifted **less_than** and projected to the time domain.

Rate of Change Operations

It is useful to note for values that change over time, the rate at which they change. From the definition of calculating a derivative, the **derivative** operation is applicable to any temporal type that supports a difference operation and is divisible by a *real*. Since *point* has three different possible representations of difference (distance, direction, vector difference), there are three derivative operations defined for it, namely **speed**, **turn** and **velocity**. Derivation is not applicable to discrete types such as *int* since there is no division available.

4.5 Illustrations and Examples Using the Model

The examples shown in this section are expressed using a language format different from that used in [GBE+00]. In that paper the authors describe a query language based on SQL, which incorporates the operations of the model. For illustration, an example from [GBE+00] on a multimedia scenario addresses the query "What is the screen layout at the 5th second of the application?" and is defined by:

```
SELECT name, val(atinstant(actor, 5))
FROM object
WHERE present(actor, 5)
```

where `object` consists of a *string* called `name` and a *moving(region)* called `actor`.

However since the implementation described in chapter 5 is done with an object-oriented language, the examples that follow are illustrated in a similar object-oriented context.

The examples are based on a prototype application for the South African Sea Fisheries Research Institute developed as part of this thesis. The types of objects used in the examples are shown in table 4.4.

A SARP region is an area in the ocean specially demarcated for research purposes. Since the boundaries of this area can change, it is classed as a moving region. `Cape Cross` is the name of a beacon on the western coast, and `Red Tide` is a mass of water with depleted oxygen, and high levels of natural toxins.

Object	Data type
textttShip	*moving(point)* (or *mPoint*)
textttRed Tide	*moving(region)* (or *mRegion*)
textttCape Cross	*point*
textttSarp Region	*moving(region)*

Table 4.4: Names and types of the objects used for an application for SFRI.

4.5.1 Example 1

There are different ways for answering the following query:

"During which times was the ship 'Seaflower' inside SARP region S4 ?"

The first makes use of the inside() operator as follows:

```
Periods p = Seaflower.inside(S4).at(true).deftime()
```

An alternative is to use the `overlaps()` method instead of the `inside()` method, but an even simpler solution is to use the `at()` method as follows:

```
Periods p = Seaflower.at(S4).deftime()
```

Each method call has a particular return type. The `inside()` operation has a *boolean* return type, but since it is the 'lifted' version being called on a moving object, the return type should also be a *moving(boolean)*, or *mBool*.

In both cases the `at()` method is a restriction on the range of values of the moving object, in the first case restricting the *mBool* only to `true` values, and in the second case restricting the *mPoint* (*moving(point)*) only to values within the given region.

This query would return the range of times [00:10,00:12,00:14,00:15,00:17].

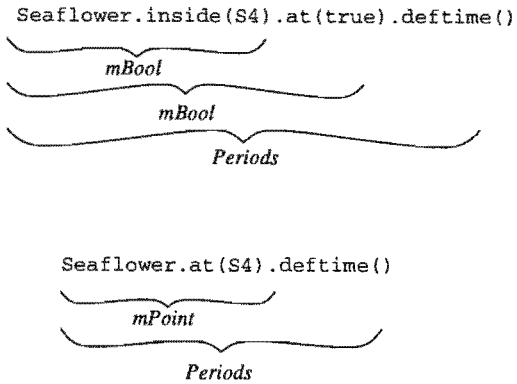

Figure 4.7: Equivalent queries, showing intermediate result types.

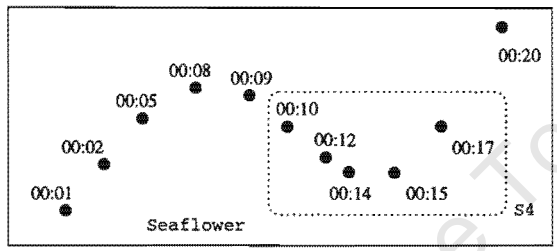

Figure 4.8: The path of the ship "Seaflower" across the ocean.

4.5.2 Example 2

This query is based on the topographical relation between a moving point and a stationary point:

> "When was the ship 'June' within 10 degrees of north from Cape Cross?"

It is answered as follows:

```
mBool left = June.direction(CapeCross).less_than(5)
mBool right = June.direction(CapeCross).greater_than(355)
Periods result = left.and(right).at(true).deftime()
```

The query has been broken up into three separate parts in order to make it fit on this page, but that also helps to make it more understandable. From this it is clear that the `less_than()` and `greater_than()` operations return moving booleans.

In order to see the furthest distance the ship was away from Cape Cross while being within 10 degrees of north, the following line can be added to the query:

```
real distance = June.atPeriods(result).distance(CapeCross).atMax().val()
```

4.5.3 Example 3

"Where did the red tide spread over areas greater than $50km^2$?"

```
Periods time = redtide.area().greater_than(50).at(true).deftime()
Region bigRed = redtide.atPeriods(time).traversed()
```

4.5.4 Example 4

"Could the ships 'Seaflower' and 'June' ever have been in radio contact (i.e. within $200km$ from each other) during week 42?"

```
mPoint w42 = Seaflower.atPeriod(week42)
boolean radio = w42.distance(June).less_than(200).passes(true)
```

4.5.5 Example 5

"Which of the vessels x and y was the first to reach the red tide area, and for how long did it stay on the border of the red tide area?"

```
Period xTouch = x.touches(redtide).at(true).deftime().min()
Period yTouch = y.touches(redtide).at(true).deftime().min()

if (xTouch.before(yTouch))
  return x.onborder(rt).deftime().duration
 else
  return y.onborder(rt).deftime().duration
```

The if-statement used in this query would be language specific. As was mentioned earlier, the examples done in this section are presented in an object-oriented style. To perform a query on multiple objects, some constructs are needed in the language to enable that. The language of the model itself does not extend that far. In [GBE+00] for example, a language based on SQL is used, so that multiple objects can be queried by using a SELECT statement to retrieve them. Or as in the example above, the two objects are queried individually and the results compared using an if-statement.

4.5.6 Conclusions

The spatiotemporal model described in this chapter contains a small and simple set of types, and a number of well-defined operations on each of those types. These two characteristics made the model ideally suitable for this project, since the types could be translated into Java classes, and the operations then implemented as methods in the classes. The types are simple and general and can be applied to any application involving spatiotemporal data (multimedia, vehicle tracking, GIS, etc..). Some of the models introduced in section 2.3 are rather more oriented towards specific application areas such as GIS and include types relevant only to those domains.

The designers of the model [GBE+00] had followed a number of design principles which ensured that the model behaved in a consistent manner. For example there are no operations that could only be applied under certain conditions, and none of the operations would ever return unpredictable or ambiguous results. This makes the model easy to use and easy to understand. The examples in section 4.5 have shown how operations can be applied to solve a wide range of spatiotemporal queries.

Chapter 5

Implementation of the ST Model in Java

This chapter gives an overview of how the ST model described in the previous section was implemented as a class library for Persistent Java. Since the ST Model is an abstract model, some mapping had to be made between the continuous and discrete domains. This mapping will be described in this chapter, as well as the implementation of each of the main types.

5.1 Introduction

Most representations of spatiotemporal data are implemented by storing time and values together at some level (section 2.3). In relational systems tuples are stored with temporal and spatial fields. In object-oriented systems time can be added as attributes for spatial data types or even bound lower down at the object level (section 2.1.3). In any implementation there is only a finite set of (time,value) pairs that can be explicitly represented (Figure 5.1). The more often a moving object is updated (i.e. when it 'moves'),

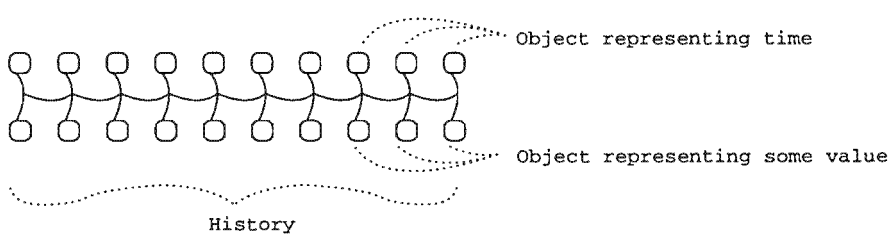

Figure 5.1: A Moving object is implemented as a series of (time,value) pairs.

the longer its history would be. It is nevertheless *atomic* in the sense that there are no

redundant moves. For example, a problem with the snapshot model[1] in GIS systems is that whenever a single feature of some map changes, a new snapshot has to be taken of the entire scene, including the attributes that stayed constant. Therefore it has a lot of redundancy (Figure 5.2).

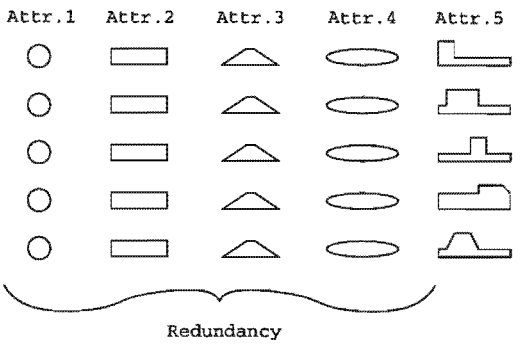

Figure 5.2: A snapshot model could be inefficient for recording changing data.

In the implementation of moving types however, each moving object has its history updated independently of other moving objects, and only when there is a change in its value. The granularity of that value is decided by the application builder.

An alternative would be to model the movements of objects using mathematical formulas. Such functions can yield a much wider range of (time,value) pairs by calculating them from arbitrary arguments, however it also involves a lot more complexity and is seldom feasible since objects rarely move according to patterns that can be expressed mathematically.

Table 5.1 gives the names of classes that were developed as our discrete representation of the spatiotemporal model in Persistent Java, together with the corresponding type in the model. From the outset some of the classes of the standard Java API were used – particularly for implementing the *BASE* classes. There did not seem to be any reason not to use them, and to create a new custom class that represents integers for example, might have only caused compatibility problems with other applications at some future stage. Therefore is was decided to use available classes until such at time that it became evident that a custom class needs to be defined (as was the case with `Point`, see section 5.6.1).

The rest of this chapter is organised into sections corresponding to each of the groups in the type system (table 4.1). In each section the implementation of classes in that group is discussed, as well as any mapping that was made from continuous to discrete domains. The parameters of methods are omitted from most diagrams for the sake of clarity. In section 5.3 the implementation of *TIME* types is outlined, which is very similar to the way in which *RANGE* types were created (section 5.4). The *TEMPORAL* types are discussed in section 5.5 and *SPATIAL* types in section 5.6. The difference between

[1]See section 2.4.3

Java Class Name	Type in ST Model	Group
Integer	*int*	*BASE*
Double	*real*	
Boolean	*bool*	
Period	*instant*	*TIME*
Periods	*range(instant)*	*RANGE*
IntRange	*range(int)*	
RealRange	*range(real)*	
Point	*point*	*SPATIAL*
Points	*points*	
Line	*line*	
Region	*region*	
InTime	*intime*	*TEMPORAL*
mInt	*moving(int)*	
mReal	*moving(real)*	
mBool	*moving(bool)*	
mPoint	*moving(point)*	
mPoints	*moving(points)*	
mLine	*moving(line)*	
mRegion	*moving(region)*	

Table 5.1: Names of implementing classes

TEMPORAL and *TIME* is that *TEMPORAL* types contain both *TIME* and *BASE* or *SPATIAL* components.

5.2 Using Wrapper Classes to Implement *BASE* types

Although Java is described as a purely object-oriented language, it does also include a number of primitive data types that are not objects. As a design decision, this was done to make the language follow C and C++ more closely, which would benefit many programmers [GM96]. Also, there is a wrapper class available for each primitive type, which encapsulates it inside an object, if required (enabling the programmer to write purely object-oriented code).

This left the choice of implementing the *BASE* types in the model either as primitives, or wrapper classes from the Java library, or custom made classes. If primitive types were to be used, then there would have been the problem of representing an *undefined* value for each type. Therefore implementing *BASE* types as objects holds the advantage that an object could have a value, or it can be undefined. There is a special value assigned to uninstantiated objects in Java called `null`.

Figure 5.3: The *undefined* value in the carrier sets of *BASE* types are represented by null references.

Creating custom objects would have been an easy task, but since they would hold no clear advantage, and to keep possible future interactions with other Java applications as simple as possible, it was decided to use the existing Java wrapper classes.

5.3 Mapping *TIME* from Continuous to Discrete

Finding a suitable representation for time is often a problem (the Y2K predicament was a significant result of that). The `Date` class in Java contains a `long` integer that denotes the number of milliseconds since or before midnight, January 1st, 1970.[2] Although it is slightly restrictive in the sense that changes during intervals smaller than a millisecond cannot be recorded for moving objects, it is a conceptually easy and simple approach which makes it attractive.

Since moving objects can stand still at a certain position/value for a length of time, it seems desirable to be able to represent time, not only as a single value, but also as an interval. This would be particularly useful for applications where objects are stationary more often than they are moving (e.g. in GIS). For this reason, the *instant* type was defined as an interval in a class called `Period` (Figure 5.4).

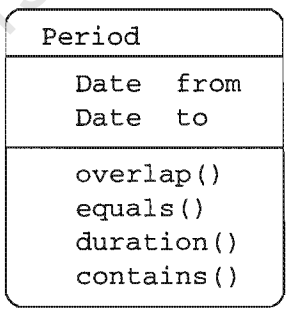

Figure 5.4: The `Period` class - implementation of *instant*.

If a single value representation of time were used instead, then an interval in time would have to be represented by numerous separate values. Should an object stand still in this case, there would have to be d/g (time,value) pairs where d is the duration of the interval and g is the minimum time granularity required. The longer the interval, or the finer the granularity, the more storage space would be required.

[2]This date format can count all milliseconds up to 292 million years from now.

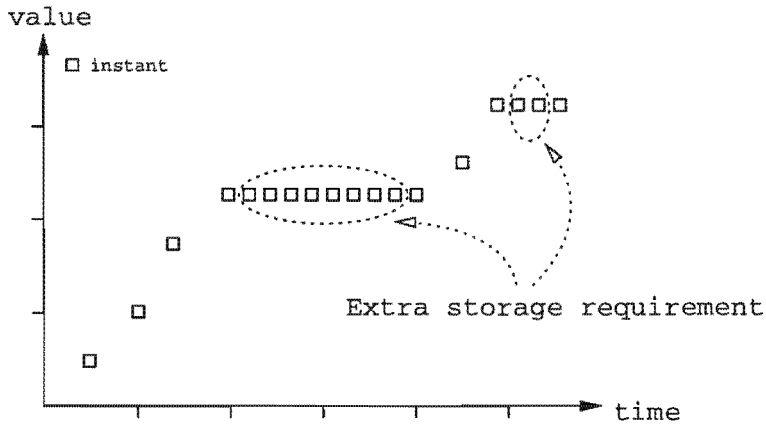

Figure 5.5: Representing time with single values.

With a time interval representation on the other hand, only the initial and final values of the interval needs to be stored, and all intermediate values are inferred. If it is ever necessary to record a single time value then the starting and ending times of the interval can simply be made equal. Moving objects that never stand still for longer than a millisecond would take up a bit more storage space than is really necessary since *instant* is consistently represented as an interval (Figure 5.6). To be precise, for every movement there would be at least 8 bytes extra storage space used.[3] In a worst case scenario the cost would be $n \times (8 + e)$ bytes, where n is the number of movements in the object's history, and e is the storage requirement of additional overhead for handling two `Date` types. It seems that the worst case for using a single value representation would in most

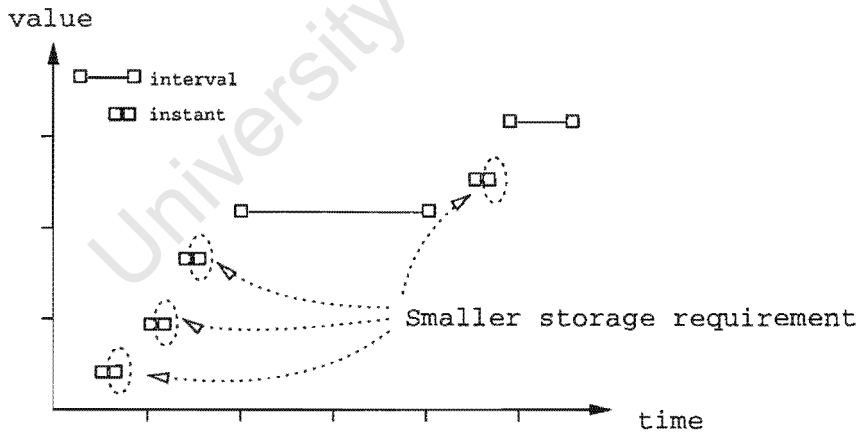

Figure 5.6: Representing time consistently with intervals, for the same situation as in Figure 5.5.

situations outweigh the worst case of using intervals.

For any interval there is a finite number of values that are considered part of the interval.

[3]The `long` data type in Java uses 8 bytes.

This follows from the fact that computer architecture is limited, and one can not represent arbitrarily small numbers on it. Therefore all intervals can be considered *closed* on both ends. Intervals of time, real numbers and integers were all implemented in the same way.

The *periods* type would naturally also be represented as a range of intervals, rather than a range of single values (Fig. 5.7). Depending on the application, this representation of time can be used in different ways, for example to represent separate time intervals such as for sound emitted from a loudspeaker (Fig. 5.7a), or time measured for singular events such as the times when lighting strikes (Fig. 5.7b), or for events that change the state of an object which continues existing, such as a plot of land that has a change in one of its boundaries (Fig. 5.7c).

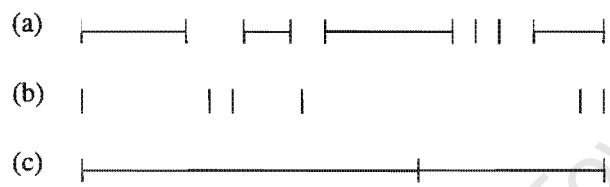

Figure 5.7: A range represented by a group of intervals

This same approach was adopted for representing ranges of real numbers and integers. The following section explains more about `Range` class implementation.

5.4 *RANGE* Types

The existing `Collection` classes in Java were used for the implementation of range types.[4] In particular, all the `Range` classes extend the `ArrayList` class of the Java API. There is an `IntRange` and `RealRange` class for the *range(int)* and *range(real)* types respectively, as well as a `Periods` class for *periods* (equivalent to *range(instant)*).

Each of these classes carry only one modification from the original `Collection` class, which is an overridden `add()` method. Their `add()` methods have been modified to check the class of the object that is to be added to the `Range`, and only if it belongs to the *same* class as the rest of the objects in the `Range`, will the amendment be carried out. Other constraints could also be built into the `add()` method, such as ensuring that all the intervals in the range are disjoint, or that adjacent intervals be joined to form a single interval.

Each `Range` class has a few overloaded versions of the `add()` method, taking either intervals or single values as parameters (Fig. 5.8). The `Collection` will however only contain intervals of a *specific* type (namely `IntInterval`, `RealInterval` or `Period`). The

[4] `Collection` is an abstract superclass with various subclasses for holding objects in arrays, lists, sets, maps, etc.

interval classes are all analogous and behave like the `Period` class that was described in the previous section, i.e. ranges of times and numbers are all sets of intervals.

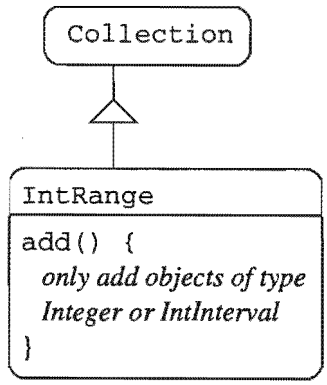

Figure 5.8: Each range type was implemented as an extended `Collection` class.

`Range` types could have been implemented with a single class, making use of the reflection API[5] to apply it to the various types of ranges. A `Range` class could then have an extra attribute which is the `Class` type of whatever object was first added to it. This 'sample' `Class` object can even be set before anything is added to the `Range` (Fig. 5.9). Then every time a new object is added to the `Range`, the object's `Class` can be compared with the `Range`'s sample class, and it would only be added if they are the same.

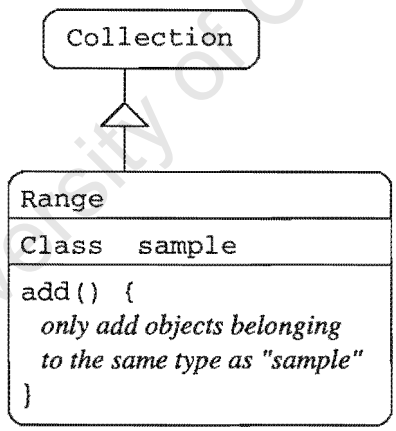

Figure 5.9: An alternative implementation for the range types.

But there may be cases where a specific type of *range* is expected, and where nothing except that type would be appropriate, for example, the `atPeriods()`[6] operation of the *moving* type will require a *periods* as an argument, nothing else. A check would have to be put in place to ensure this. For example the `instanceof` operator of Java could be used to check the name attribute in the sample `Class` contained in the `Range` class.

[5]There are some classes in Java that can be used to determine the type (`Class`) of an object, and its attributes, methods, etc. This is known as the reflection API.

[6]`atPeriods(Periods p)` returns that part of a moving object that is defined over the given periods. See section 4.4.2

Although this alternative approach is equally viable, it seems that the trouble of creating a class for each range type (all having very small differences) was negligible enough. With the chosen set of classes, the user of the model will be able to see from the declarations of their `Range` objects, what types would be contained in them. The former approach is therefore considered to contribute to more readable code at the cost of hardly any additional complexity.

5.5 *TEMPORAL* Types

In the model there are two *TEMPORAL* type constructors namely *intime* and *moving*. The implementation of the *moving* classes was done using *intime* objects as building blocks. Using the approach of storing (time;value) pairs (in this case, *intime* objects) to represent the history of a moving object is quite intuitive, general and dynamic, as illustrated in the introduction of this chapter.

5.5.1 The Intime Type

The *intime* type combines one of the *BASE* or *SPATIAL* types with an *instant*. In other words, it represents a specific value (possibly spatial) at a certain time. It is not essential for ensuring closure of the type system, but it has been included rather as a helpful type where a specific value, corresponding to a specific time or time period, can be recorded. The `InTime` class which implements this type, combines a reference to a `Period` object with a reference to some other spatial or base object. The references are private members of the class, but accessible through the `inst()` and `val()` methods as in the model.

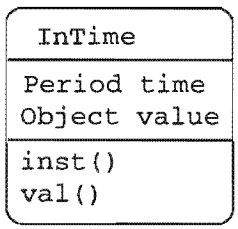

Figure 5.10: The `InTime` object.

A difference between the abstract model and the implementation is that `InTime` can represent a value over an interval, not only at an instant.

Initially there was an `InTime` class provided for each type of *BASE* and *SPATIAL* object (e.g. `IntInTime`, `RegionInTime`, `BoolInTime`, etc.). As in the case of the range types described earlier, this makes code more readable and allows less room for type

casting errors. However, because `InTime` objects form an integral part of the structure of moving objects, the `InTime` classes had to be generalised when the common functionality of the moving classes was generalised (as explained below).

There is nothing preventing the programmer from placing *any* Java object in an `InTime`. This type of anomaly has to be checked by objects or methods that make use of `InTime` objects. It is also possible to leave the value reference as `null`, but this is acceptable according to the model since all types contain the *undefined* value in their carrier sets.

5.5.2 Moving Types

For each moving type a class was created following the same naming convention e.g. `mInt`, `mBool`, `mPoint`, `mRegion`, etc. They contain a `Collection` of `InTime` objects together with the relevant set of operations. The structure is illustrated in Figure 5.11.

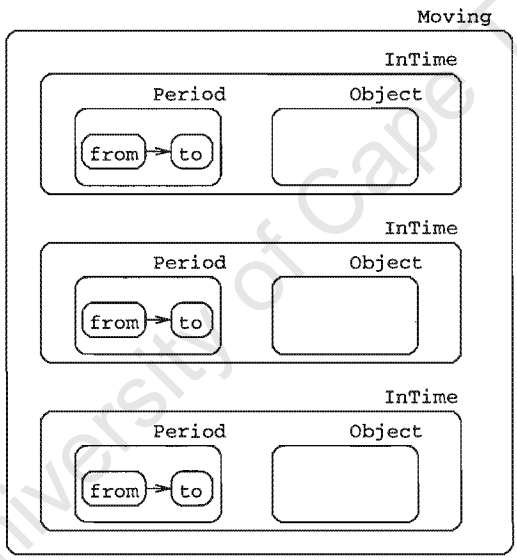

Figure 5.11: The structure of moving objects

The `InTime` objects are not ordered by their `Period` components, but whenever a moving object 'moves', and a new `InTime` is added, a check is performed to make sure that the `Period` of the new `InTime` does not overlap with the `Period` of any `InTime` already in the history of the moving object.

To keep the set of `InTime`-s sorted by `Period` is not very costly, as `InTime` objects would most often be added in chronological order as the object moves. Therefore the overhead of keeping a chronological history would be, in the general case, quite cheap. Certainly for objects with large histories, to search through an unsorted set of `InTime`-s for possible overlapping periods becomes increasingly less efficient. The approach adopted was to provide only the basic structure within the moving classes, and to give users of the class

the option of adding further structures like indices at their own discretion. This aspect of the class library is discussed further in the next chapter.

The moving classes have a method called `move()` which takes an `InTime` as argument and appends it to the object's history. This method contains the necessary checks to ensure data integrity. It could be extended to include a mechanism to add the new `InTime` at its correct position in the history.

An abstract class for moving types was created, which contains the basic projection operations and the `move()` operation that checks that no time intervals in its history overlap. Since `move()` and some of the other projection operations like `atPeriods()` performs the same function for all moving types, the common functionality was generalised and this abstract super-class was created. The class, called `Moving`, now contains all the general methods.

Each specific moving class extends this abstract class, adds its own operations to it, and overrides the `move()` method to make sure that only objects of its particular type is added to its history. For example, a moving integer should only have `InTime` objects containing [`Period`, `Integer`] pairs added to its history. A representative sample of the operations in the model were implemented, but some other methods, which would require very similar implementations to those that were created, were omitted for the time being.

Generalising a method like `present()` was not problematic because it returns a `boolean` irrespective of the actual type of the moving object. On the other hand, to generalise `atPeriods()` was not as simple since it had to return an object of the *same* type as the calling object. For example, if the `Moving` object is a moving integer, the method must return an `mInt`. If it was a moving point, it should return an `mPoint`, etc. Because the method is defined in the parent class, which is abstract, the return type would only be known when an implemented subclass calls the inherited method. The solution to this problem was to use reflection. The return type was declared as `Moving` but the object to be returned was created using the `Class` of the calling object, as follows:

```
public Moving atPeriods(Periods per) {

   Moving result = (Moving)getClass().newInstance();
   ...
   ...
   if (overlapWasFoundInPeriods) {
     result.move(nextInTime);
   }
   ...
   ...
   return result;
}
```

In this way the return type is always the same as the type of the calling object.

The same technique was also used in `MovingSpatial`, which is a subclass of `Moving`, and a parent class of all the moving spatial objects (`mPoint`, `mPoints`, `mLine` and `mRegion`). For example, in `MovingSpatial`, the `at()` method was generalised as well as the lifted version of `distance()`. A representation of the Moving hierarchy is shown in Figure 5.12.

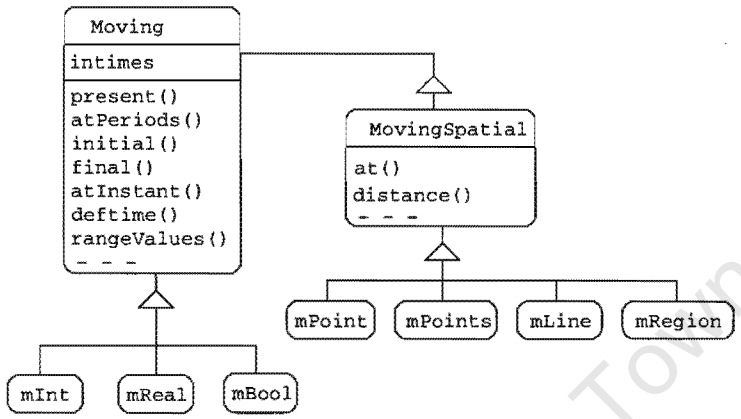

Figure 5.12: The class hierarchy of moving objects.

5.6 *SPATIAL* Data Types

There are four classes for spatial data types, namely `Point`, `Points`, `Line` and `Region`.

Producing a discrete representation for the spatial types is a more involved procedure than for the *BASE* and *TIME* types, because they are two-dimensional. Choosing a discrete representation is only the first step in constructing the spatial types. Most of the work lies in developing the operations for complex two-dimensional types that compute, for example, the intersection between lines and regions.

Spatial types could be discretely represented in a number of ways. `Point` and `Points` types are relatively easy and intuitive, but `Line` and `Region` are more complicated. Regions are especially problematic, because they can consist of multiple disjoint parts, each of which can have any number of holes in it (see Fig. 5.13).

One possible solution would have been to design two classes – one for a simple region and one for a complex region. The simple region would have a boolean flag which indicates whether its coordinates denote a region or a hole inside a region. The complex region would be a class which contains a set of simple regions, each denoting whether it represents a hole or not. A simplified diagram of such a class structure is shown in Figure 5.14.

This approach would have a number of difficulties. In order to add a simple region that

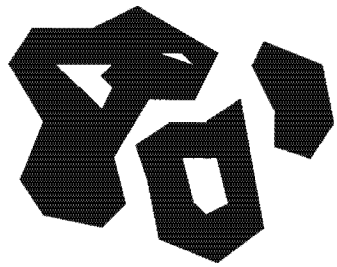

Figure 5.13: A `Region` can have disjoint parts and holes

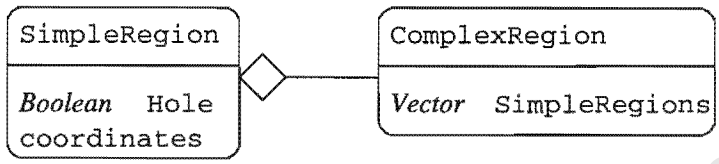

Figure 5.14: Classes for an alternative representation of regions.

constitutes a hole to a complex region, the complex region must already contain a (non-hole) simple region that completely contains it. Also, for operations that return regions as results, determining whether the resulting region is simple or complex and whether it contains holes or not is problematic.

A different approach was chosen for implementing spatial types. It is based on a simpler framework, where all spatial objects are defined by a number of vertices and edges[7], as illustrated in Figure 5.15 This idea has been used extensively in many advanced computer graphics applications [FvDFH90].

The specific shape or structure of a spatial type can be constrained by the way in which edges and vertices are connected. Not all vertices have to be connected, which makes the representation of complex regions with holes a lot easier. This idea is described more fully in section 5.6.4 on the `Region` class.

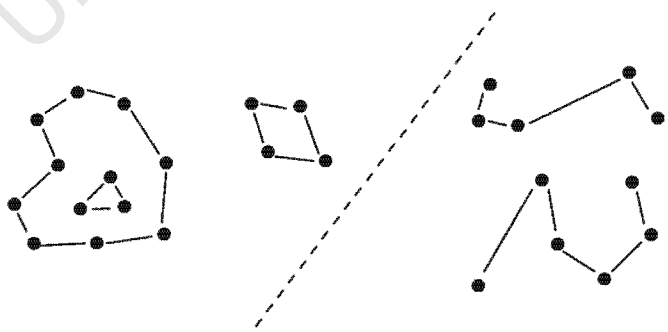

Figure 5.15: A vertex-edge representation of a region (left) and a line (right).

[7]An edge is a straight line that connects two vertices.

Another advantage of this approach is that the similarity between `Points`, `Line` and `Region` classes now makes it an ideal situation for using inheritance. `Points` consists of a number of vertices. `Line` is a `Points` with edges, and `Region` is a `Line` that forms a closed region.

A parent class called `SpatialObject` was introduced for all Spatial classes. This was needed at first to create a simple graphical display used for rendering `SpatialObjects` on the screen. It contained an abstract method called `draw()` that all subclasses were forced to implement, which would contain the code for graphical display. `addVertexClick()` was added to place a new vertex where the user clicks a mouse button on the background, and `waitToBeCompleted()` stops the current thread of a query program while the user draws the spatial object (this was useful for interactively doing spatial queries).

It became clear that `SpatialObject` was not only necessary for rendering objects on the screen, but some of the spatial operations could use it as an argument, thereby applying polymorphism to reduce the total amount of code. Operations like `distance()` and `overlaps()` are required for any spatial object, therefore they were also added as abstract methods to `SpatialObject`. The class hierarchy of spatial objects is illustrated in Fig. 5.16.

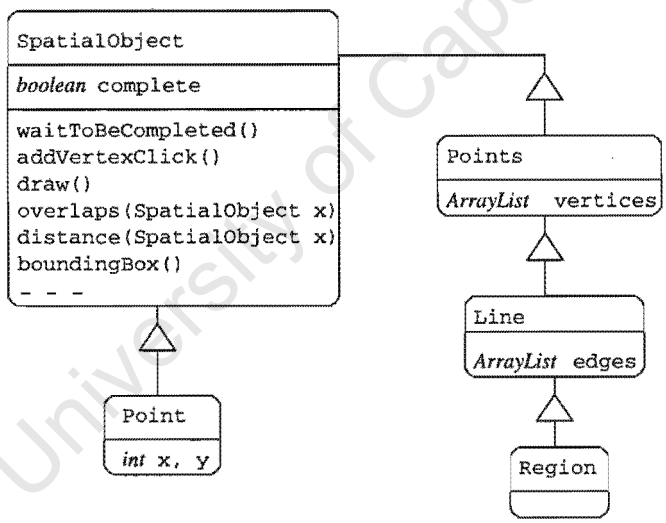

Figure 5.16: Class hierarchy of spatial objects.

5.6.1 Point

Initially a decision had to be made between whether a new `Point` class had to be written or whether the `java.awt.Point` class could be used. The question was not whether it would be too time-consuming to write a new `Point` class, but rather whether the existing AWT class could easily be incorporated into the framework.

Since the `SpatialObject` class was later created as a super-class for all spatial objects, `Point` also had to be a sub-class of it. But because Java does not support multiple inheritance, it was clear that a custom `Point` class had to be written for the package. Some operations of the model would have had to be added to the `java.awt.Point` class in any event, so the fact that its attributes could not be inherited was not a great sacrifice.

The main attributes of `Point` are two integer coordinates, x and y, that specify its location in 2D space. The class diagram is shown in Fig. 5.17. Changing spatial objects to 3D simply requires the addition of another coordinate in the `Point` class, however the spatial operations are significantly more complex in 3D.

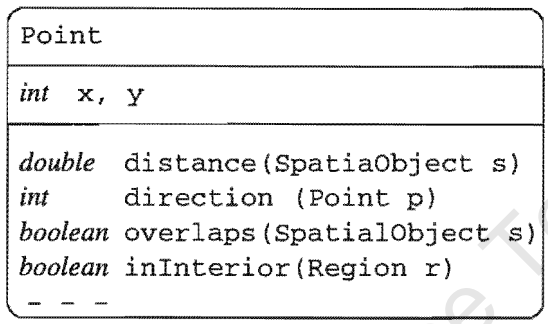

Figure 5.17: The Point class

The `direction()` method takes another `Point` as argument and returns an integer as the relative direction in degrees to that point. For example a point due north would be in direction 0, northwest 315 and south 180.

The `overlaps()` method checks whether there is any overlap between the `Point` object and a given `SpatialObject` argument. The actual type of the argument is checked using reflection, and depending on this the appropriate procedure is followed to determine the outcome of the method. It returns true in any of the following cases:

- The argument is a `Point` or `Points` and this point coincides with at least one of the given points.
- The argument is a `Line` that crosses this point.
- The argument is a `Region` that contains this point.

In the second case the `distance()` method of `Line` is called to determine whether the line crosses the point, and in the last case the `contains()` method of `Region` is called to determine whether this point is inside the region.

The same `contains()` method is called from within `inInterior()`, which returns true if this point is inside the given region (see section 5.6.4).

Calculating the distance between a `Point` and a `SpatialObject` once again depends on the type of spatial object. If it is another `Point`, then a simple Euclidean distance is calculated, if it is a `Points` then the distance to each vertex is calculated and the minimum distance is returned. Distance to a `Line` or `Region` is calculated using the `distance()` method in `Line` (section 5.6.3).

5.6.2 Points

The `Points` class is a superclass of `Line` and `Region`. The vertices of a line or a region is a collection of points, which is why this is such an ideal case for inheritance.

In `Points` there is a `Collection` that contains the set of Point objects. It was called *`vertices`*, for consistency, since it would not be redeclared in subclasses. In the following sections it would be useful to remember that *`vertices`* is a set of `Point`-s, because the term vertex and point may be used interchangeably.

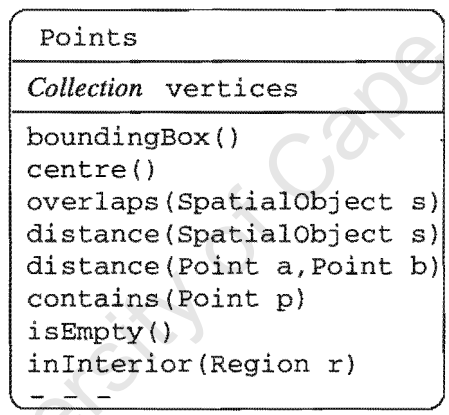

Figure 5.18: The Points class.

Most of the methods are re-used in the subclasses – even if they are overridden, they are still called. For example, the `overlaps()` method in `Line` overrides `overlaps()` in `Points`, but is first calls the superclass method to check whether any of the vertices coincide before it continues with the more computationally intensive check of whether edges intersect.

`Points` also contains the functionality to set and check extreme points (left, right, top and bottom). Every time a new vertex is added, the `addVertex()` method automatically checks and updates the extrema. `Points` can use these values to construct a bounding box and return it as a region (the `boundingBox()` method inherited from `SpatialObject`). This is useful when adding spatial objects to an R-tree index. The centre of the bounding box is also returned by `center()`, which is a rather crude approximation of the centre of a spatial object.

`contains()` returns true if the given `Point` parameter coincides with at least one of the vertices of `Points` (i.e. their coordinates are the same).

`overlaps()` checks whether any of the vertices coincides with a `Point` or `Points` (by calling contains), or whether a `Line` crosses it or a `Region` contains it.

There are two methods called `distance()` in this class. One is a static method that calculates the distance between two given `Point` parameters, using the distance formula

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The other works out the shortest distance between any of this object's vertices and a given `SpatialObject` parameter. Once again, as in `Point`, if the `SpatialObject` is a `Point` or `Points`, the shortest distance is calculated by the above mentioned distance method. However when the distance to a `Line` or `Region` is calculated, a method is required that calculates the distance between a *line segment* and a point. This method is found in the `Line` class, discussed in the next section.

5.6.3 Line

Of the spatial object classes, `Line` contains the most code and it contains the operations that were the most challenging to implement. It is the first class in the spatial object hierarchy that is composed of edges as well as vertices, which is why it contains most of the more difficult methods(Figure 5.19).

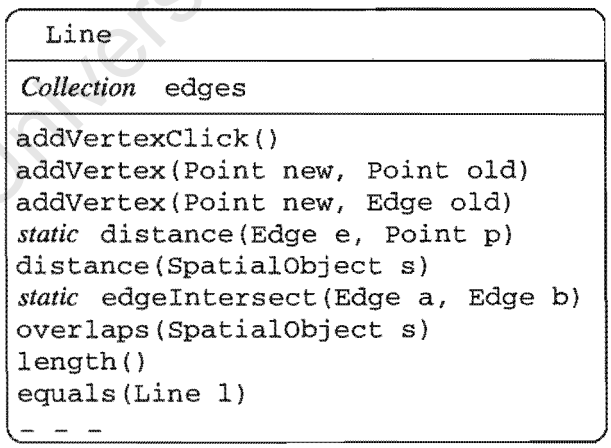

Figure 5.19: The Line class

Although `Region` is perhaps even more complex than `Line`, it was written with less than half the amount of source code since the majority of its operations were inherited from `Line` and `Points` (refer to the hierarchy illustrated in Fig. 5.16).

A small class, Edge, was written to represent an edge that connects two vertices. It contains two references to Point objects, and equals() and distance() methods. These test if two edges are equal, and return the distance between an edge and any given spatial object respectively.

Line contains attributes *vertices*, which it inherits from Points, and *edges*, a new attribute that denotes a Collection of Edges. Any Edge in *edges* can connect any two Points in *vertices*, as shown in Figure 5.20. A maximum of two vertices will have less than two edges that connect them, and those vertices will be the start- and endpoints. No two edges will be equal.

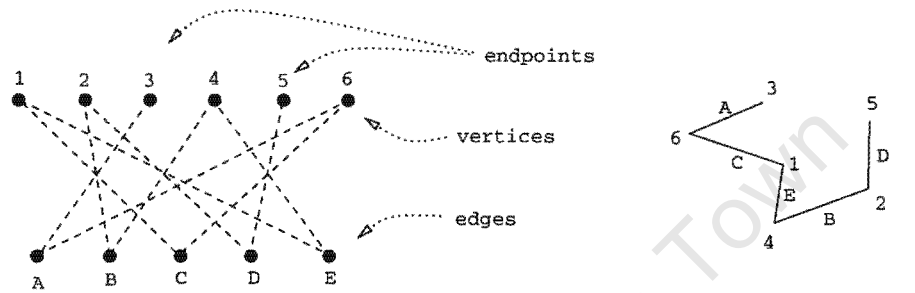

Figure 5.20: A complex relationship between edges and vertices in spatial objects.

This complex relationship makes it difficult to create a clone[8] of a Line or a Region. It is not sufficient to create a copy of *vertices* and of *edges* without retaining the references from edges to vertices, because the cloned edges will reference the original vertices and thus need to be changed to reference the new (cloned) vertices instead. Line does have a copy constructor which builds the object identical to a Line supplied as a parameter. It recreates the references between edges and vertices. For this it is necessary to know the positions of edges and vertices in their collections, to be able to reconstruct the references as in the original object. The difficulty of cloning Line-s and Region-s is an example of the well-known deep copy problem in Java, where only a single level in the object graph is copied when cloning.

New vertices can be added to a Line using one of three methods.

- addVertex(Point new, Point old) adds the new Point to *vertices* and it adds a new Edge that connects the existing (old) vertex with the new one.
- addVertex(Point new, Edge old) adds two new Edges that connect the new Point with the old Edge's Point-s, and then removes the old edge (Fig. 5.21).
- addVertexClick(Point new) assumes that the Line is being manually constructed, therefore the new Point is added as well as a new Edge connecting it to the last

[8] an object that is identical, but completely distinct from the original object.

(previous) vertex if one exists. This method was designed to be called when the user adds a new vertex by clicking a mouse button, while interactively drawing the object in a window.

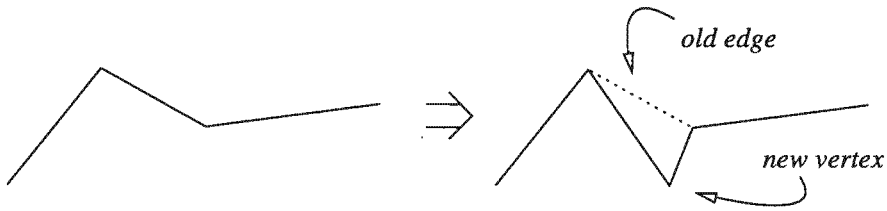

Figure 5.21: Adding a vertex on an existing Edge.

In order to determine whether two lines intersect, the `intersect()` method follows a simple algorithm:

```
for each edge in line A:
   for each edge in line B:
      if edge in A intersects edge in B
        then add to points of intersection
return points of intersection
```

While it is relatively easy to calculate the intersection between two lines that stretch to infinity, determining whether two line segments intersect is more involved. The `edgeIntersect()` method determines whether two line segments (edges) intersect, and returns the point of intersection. The technique described below can be found in [FvDFH90] and it is illustrated in Fig. 5.22.

1. Two parametric equations are set up for the line segments, such that a line is defined by $(x, y) = (a, c) + t(b, d)$ where (a, b) and (c, d) are the coordinates of the endpoints.

2. The equations are normalised such that the parameter t has values ranging from 0 to 1 for points on the line segments.

3. The actual point of intersection, say (m, n) is substituted into the equations, and they are solved for t. If either of the values of t are greater than 1 or less than 0, it implies that the line segments do *not* intersect.

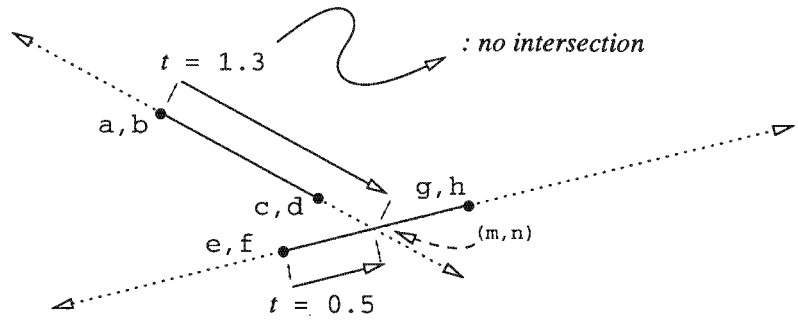

Figure 5.22: Determining the intersection of two line segments using their parametric equations.

Another method that required similar use of parametric equations was for calculating the shortest distance between a point and a line segment. Once again, where it is trivial for an unbounded line, it is more difficult to calculate distance to a line *segment*. The procedure is as follows:

1. An imaginary line is drawn from the point, orthogonal to the line segment.
2. The parametric equation of the line segment is normalised (as above) and solved for the value of t where the line intersects with the orthogonal imaginary line (see Figure 5.23).
3. If the value of t is between 0 and 1, the intersection point (m, n) is the closest point from the line segment to the given point. Otherwise one of the endpoints is closer (which endpoint is closer, depends on whether t is greater than 1 or less than 0).

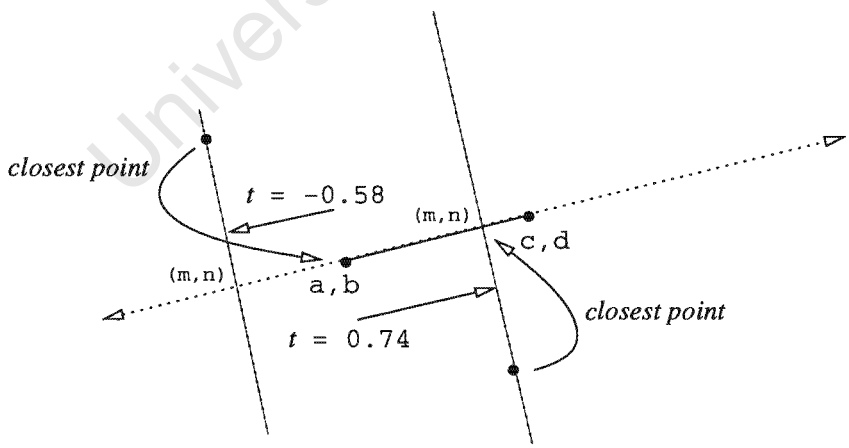

Figure 5.23: Calculating the distance between a point and a line segment.

The preceding procedure was implemented in a static method called `distance()` that accepts a `Point` and an `Edge` as arguments. This method is used in the distance method that overrides the one inherited from `Points`, which accepts a `SpatialObject` as argument. The latter `distance()` method follows the algorithm:

```
if the argument is a point
  then for each edge in this line
       find distance between
       the edge and the point
       if smaller than minimum distance
          update minimum distance
if the argument is a points, line or region
  then for each edge in this line
         for each vertex in argument
           find distance between
           the edge and the vertex
           if smaller than minimum distance
              update minimum distance
return minimum distance
```

It is necessary to check the distance between every edge of a line and vertex of the other spatial object. The shortest distance between any vertex of a line and a vertex of another object is not significant, since there could be an edge that is closer. This fact is illustrated in Figure 5.24.

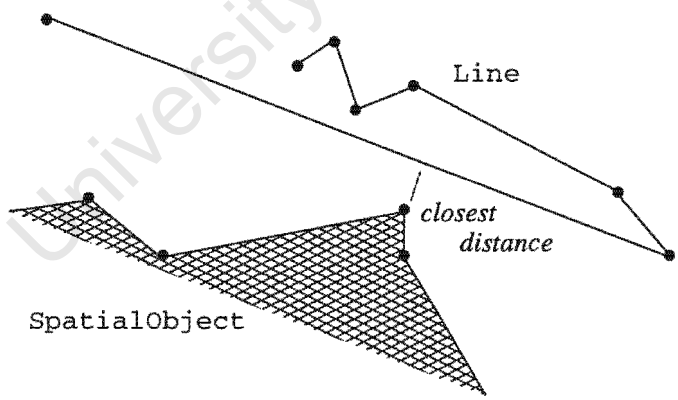

Figure 5.24: An edge connecting the vertices furthest from another spatial object could be the shortest distance from it.

Because of the inheritance hierarchy, methods like `intersect()` and `distance()` need only be declared once. `Region`, which is similar to `Line` in most respects, inherits them so they can be called in the same way. `Region` has a different structure and additional methods such as `contains()` (see 5.6.4), but the intersection points of a `Region` are determined by exactly the same algorithm as for a Line.

Not only do subclasses re-use these methods, but other `SpatialObject` subclasses can also call them. For example, `distance()` in `Point` checks whether the `SpatialObject` argument is a `Line` or `Region`. If it is, the `Point` calls `Line`'s distance method and passes itself as a `SpatialObject` parameter. Because distance is a symmetric binary operation, pt.distance(reg) = reg.distance(pt) and thus the method only has to be defined in one place.

The `equals()` method of `Line` checks whether the argument has the same number of vertices and edges as the calling object. It then checks if each edge in the argument connects the same vertices as in the original. Since the conditions of equality are exactly the same for a `Region`, the method can simply be inherited.

The functionality to get a bounding box for the line is unchanged as it was inherited from `Points`.

5.6.4 Region

The implementation of a `Region` class is small compared to `Points` and `Line`, since it inherits most of its methods from them. A region has structure similar to a line in that it consists of vertices and edges, but it also has additional constraints. A region can be defined as:

> A set of vertices and edges where each edge forms part of a closed loop of edges, and none of the edges intersect.

This ensures that none of the 'simple regions' or holes or islands in the holes overlap, and that a hole can not extend outside the perimeter of a region.

Every time a vertex or an edge is moved, added, or deleted, a check needs to be performed that makes sure that the above definition is satisfied.

Another constraint that can be installed is to ensure that a region contains at least three connected vertices. Such a constraint however, can make it difficult to create a region initially, i.e. after it is instantiated, one vertex is added. Then, for a while before the next vertex and the first edge are added, the region contains only a single vertex – which would breach the constraint. A constructor could be designed to create three vertices at arbitrary points, that can be moved to appropriate locations before more vertices are added. Alternatively the constructor can be given three edges as arguments. However to keep the creation of regions simple, this constraint was not enforced.

Some constraints were implemented however, for example that a vertex can not be added to a region without being the endpoint of at least two edges. Even the second vertex that

is added to a region will be connected to the first vertex by two edges. This constraint was necessary for the `contains()` method to return a correct result in the case where a region has only two vertices.

The `contains()` method determines whether a point is inside a region as follows (also from [FvDFH90]):

1. Construct a line from the point to infinity (in the discrete case, the edge of space that can be represented).
2. Count the number of intersections between the new line and edges of the region.
3. If the number of intersections is even, then the point is not inside the region. It is only contained if there are an odd number of intersections.

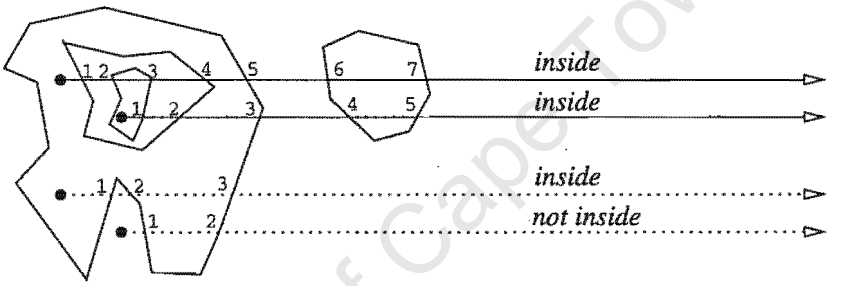

Figure 5.25: Determining whether a region contains a line by counting the number of intersections with a line from the point to infinity.

There are some methods that were not implemented for `Region` due to time constraints and their complexity. One such method is `area()`. Calculating the area of a region can be done by applying a triangulation algorithm, which divides the region up into a mesh of triangles. Each triangle's area can be calculated and summed up to find the total area of the region. Alternatively, a crude approximation would be to simply return the area of the bounding box of the region.

The `overlaps()` method determines whether a region overlaps with another spatial object by testing whether any of the spatial object's vertices are inside the region (the test is done by calling `contains()`). According to the model however, the `overlaps()` method should return the actual section where the objects overlap. Although this is not difficult to determine for lines and points, calculating the area of intersection of two regions is an expensive procedure. It involves determining the points of intersection and clipping one of the regions [FvDFH90]. Determining both the area and overlap of regions are even further complicated when there are holes and islands in the regions.

The `perimeter()` method is merely an alias for `length()`, inherited from `Line`.

5.7 Package Organisation and Usage

The classes listed in table 5.1 were all added to a Java package called ST. Some other packages that were used are `java.util` and `java.io` (for the `Serializable` interface that was used when measuring physical object size). When developing a spatiotemporal application, the ST, `util` and `io` packages must be imported, then moving objects such as `mPoint` can be instantiated. Often however, moving objects also have other changing attributes such as 'temperature' or 'priority', that change along with their spatial locations. In such cases, a subclass of the moving class has to be created that deals with additional changing attributes. Different approaches can be taken as illustrated in the example below.

Assume an application must keep track of ants with unique names, that stop at various locations, each location also having a unique name. Each ant must at all times also have a status of either 'late' or 'on time'. Figure 5.26 shows one way in which this problem can be resolved.

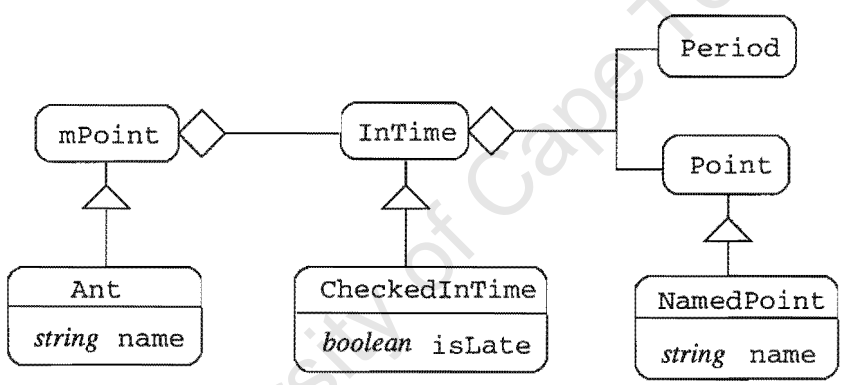

Figure 5.26: Using inherited subclasses of `InTime` and `Point` to store additional changing attributes.

An `Ant` class that inherits from `mPoint` is given an additional attribute called `name` that holds the ant's unique name.

> Data that describes a moving object, but does not change throughout its history should be contained as an attribute of the subclass of `Moving`.

A subclass of `Point`, called `NamedPoint` has an additional `name` attribute that holds the unique name of the location, and `CheckedInTime`, which inherits from `InTime` has an `isLate` attribute. When the `Ant` moves, a `CheckedInTime` object, which contains a `Period` and a `NamedPoint`, must be added to its history, that way the other changing attributes are recorded in the `Ant`'s history. It would be advisable to also add a checking mechanism to ensure that only `NamedPoints` and `CheckedInTimes` can be added to the history. An alternative approach would have been to design the `Ant` as a container class

that holds an `mPoint`, `mString` and an `mBool`, each of which records different changing data (Figure 5.27. This approach would need to have a checking mechanism which ensures

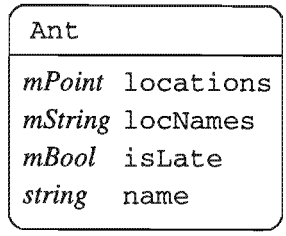

Figure 5.27: Composing different moving objects to store different changing attributes.

that the times in each of the moving attributes are synchronised.

> Attributes that change together with an object's location can either be added as single moving object, or as an attribute that inherits from `InTime` or the `Object` contained in `InTime`.

An intermediate approach can also be followed where all the changing attributes (i.e. including `isLate`) are stored in a subclass of `Point`, and all non-changing variables in a subclass of `mPoint`. It is up to the user of the `ST` package to decide which approach is suitable for the particular application. In general it would make sense to attach changing variables that have to be *synchronised* in time, lower in the structure, i.e. below `InTime` or `Point`. Classes composed from many moving objects may be less efficient in terms of size, but are often easier to use (as was found in the experiment in section 7.2).

5.8 Summary

This chapter outlined how the spatiotemporal model was implemented in PJama. Some of the existing classes in the Java API could be used such as the wrapper classes for *BASE* types and `Collection` for implementing *RANGE* types. Other classes such as Point of the Java AWT could not be used since it had to form part of the Spatial hierarchy and Java does not support multiple inheritance. An interval representation of time was chosen so that numeric and time intervals could be concisely expressed.

As in the model, a single class (`InTime`) represents all types of *intime* objects (such as *intime(point)* or *intime(real)*). Separate `Moving` classes had to be designed since the operations of moving types vary between different types of moving objects. Through the use of the Java reflection API however, it was possible to generalise many common operations in a `Moving` superclass. Not only does such a superclass allow the creation of heterogeneous groups of moving objects, but it also greatly simplifies maintenance.

There are also other areas where the use of inheritance was extremely beneficial. *RANGE* types, for example, were implemented by inheriting from a `Collection` class and overriding some of its methods. *SPATIAL* classes were integrated in a hierarchy where operations in superclasses were used by all descendant classes, and complex operations only had to be defined once. The opportunity for creating such a class hierarchy for spatial objects was a result of the generic way in which spatial objects are represented (with vertices and edges). If there had been support for multiple inheritance in Java, the design may have been slightly different in that `mPoint` may have inherited from both `Moving` and `Point`, for example. This would not have made the implementation of the moving classes any simpler, however it probably would have been advantageous for the user of the library to have an `is-a` relationship between a moving point and a point. The chapter that follows focuses on the techniques used for adding indexing structures to moving objects in an efficient and flexible way.

Chapter 6

Indexing

The histories of moving objects have the potential to grow very large over extended periods of time. Finding the state of an object at a particular time interval may require the use of an efficient access method to facilitate the search.

This chapter is organised as follows: some background on spatiotemporal indexing methods is given in the following section, and section 6.2 gives an overview of how indices were added to the spatiotemporal classes discussed in chapter 5. Section 6.2.4 addresses the question of creating indices over multiple moving objects.

6.1 Background on Spatiotemporal Indexing

The study of spatiotemporal data is a relatively young area of research, and therefore only a small amount of research has been done towards creating efficient indexing mechanisms for spatiotemporal data. There is an increasing amount of interest in this topic though. Many large telecommunications companies are facing problems with distributing bandwidth across their networks [KGT99], [SJLL99]. To know where mobile phone users are and where they are physically moving towards, i.e. where more bandwidth would be required in the immediate future, can be valuable information for such organisations.

Some studies have focused on solving the problem of indexing the current and possible future positions of moving objects. In [KGT99] an indexing method is presented whereby the trajectories of moving objects are mapped to a higher dimensional space. For example a line with the equation $y(t) = vt + a$ is mapped to a point (v, a) in a *dual space*. The mapped point can then be indexed with a point indexing method such as a Kd-tree [Rob81]. This avoids problems associated with indexing lines with spatial access methods like the R-Tree, where the minimum bounding rectangle of a diagonal line contains a lot of dead space (Fig. 6.1). Using the method of [KGT99], spatial queries that return the

future positions of moving objects then also have to be mapped to the dual space.

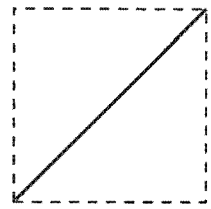

Figure 6.1: The minimum bounding rectangle of a diagonal line, showing the large amount of dead space.

The authors of [SJLL99] adopt a different approach where they propose an index structure that is parameterised using velocity vectors, called a TPR-tree (Time Parameterised R-Tree). In this structure the positions of points (and rectangles that bound groups of points) are represented not only by coordinates, but by coordinates together with the speeds at which they change. The rectangles that enclose points change in size and shape as the points move. The data is thus indexed in its native n-dimensional space (not mapped to a higher dimension), but the index can be "viewed" at some future time.

Neither of the approaches mentioned here employs any data "replication" i.e. the position of a point is only stored once, and positions at other times are inferred. Also, both index structures remain functional indefinitely and do not require periodic rebuilding. They can however be optimized for queries only in some specific time horizon, and due to the uncertainty of future trajectories of objects, their efficiency deteriorates as time progresses.

The authors of [NST99] claim that at the time of writing there were only five known index structures for accessing spatiotemporal data, specifically for accessing the histories of moving objects (i.e. not taking into account the previously mentioned structures because they are aimed at indexing current and future movements). The MR-tree, Overlapping Quadtree [TVM98], and HR-tree [NST98] are all *incremental* structures, with one tree for every time interval. A younger tree would contain only the spatial data that has changed since an older version, and the unchanged data is still referenced from the previous tree. This idea of overlapping trees is illustrated in Figure 6.2. The HR-tree was found to have a large storage requirement, especially if a large portion of its data is often-changing (this would mean that more of the trees have to be duplicated at every interval). This was somewhat alleviated in an experiment in [NST99] by not creating an index at every time interval, but to batch changes and only create an index at every second or third interval (using 30% and 45% less storage space respectively). The trade-off is that not every time point can be queried. A tree at an adjacent time point may have to be used and the correct answer filtered accordingly. Even so, the HR-tree is still significantly larger when compared to the remaining two methods, viz. 3D and 2+3 R-trees.

In a 3D R-tree time is added as an additional dimension to 2D spatial data. The trajectory

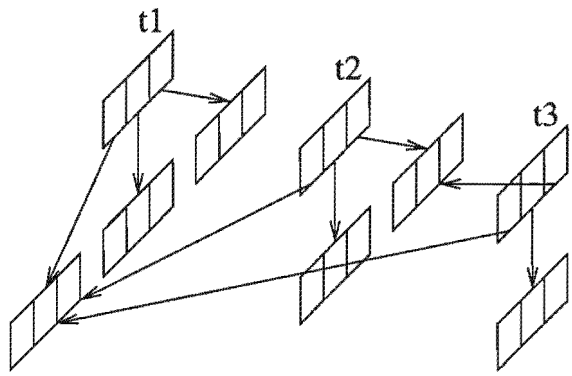

Figure 6.2: An HR-tree structure

of a point that moves over a 2D plane can be viewed as a line in 3D (Figure 6.3). This 3D

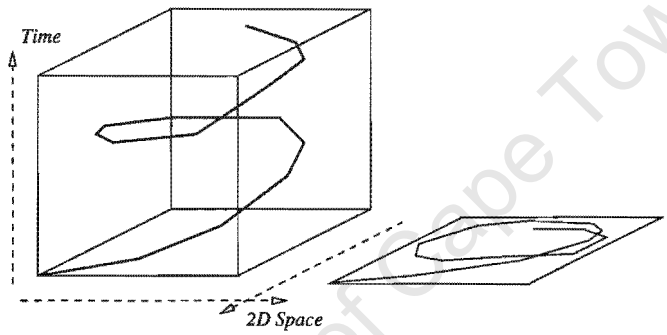

Figure 6.3: An isometric view of how movement on a flat 2D surface (right) can be represented in 3D with time as the vertical axis (left).

information can be stored in a normal R-tree or some derivative of it. One major problem with this approach is that the *current* position of spatial objects cannot be efficiently represented. If an object is currently at some point and it may or may not move again, then its trajectory forms a line to infinity, and R-trees do not handle such open lines well. This problem has previously been mentioned in section 2.1.4 in the context of using R-trees as a temporal access method. A similar solution to that proposed in [KTF98] is evaluated in [NST99], called the 2+3 R-tree. It consists of a 3D R-tree that contains only historical data, together with a 2D R-tree that contains only data at current positions. When an object moves, its previous position is moved from the 2D tree to the 3D tree and its new position is inserted into the 2D tree.

6.2 Indexing Method Applied

6.2.1 Overall Approach

For this project it was decided to use separate spatial and temporal indices to achieve efficient access to moving object data (Figure 6.4), rather than using a single spatiotemporal index. There are a number of reasons why this decision was taken.

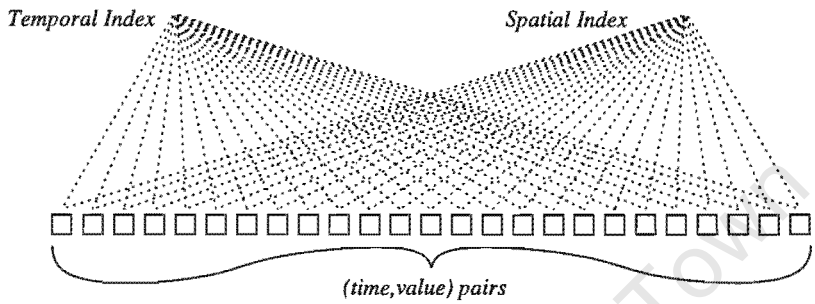

Figure 6.4: Separate specialised indices are used for spatial and temporal indexing

Firstly, the object-oriented paradigm makes this approach possible since an object can have any number of references to it. An `InTime` object can thus be part of a moving object and at the same time be referenced from the leaf nodes of several index structures.

Secondly, existing spatiotemporal indexing methods are not yet in a sufficiently advanced stage to provide an undisputed advantage. Each of the methods discussed above has some or the other drawback – the HR tree is not efficient in terms of storage space, the 3D R-tree cannot be used for online spatiotemporal applications because it cannot handle *current* values effectively, and the 2+3 R-tree is more expensive to build and maintain since every time an object moves there is an insertion and a deletion on the 2D tree plus an insertion on the 3D-tree.

Also, not all the types in the spatiotemporal model are pure spatiotemporal types. There may for example be the need to index changing real numbers or integers apart from spatial and moving spatial objects. Such temporal objects have histories i.e. they are changing, but they are not "moving" since they do not have spatial representation. In this case, the ability to use one of a number of different (including non-spatial) indices may be beneficial.

There are no spatiotemporal operations in the model - spatiotemporal queries are performed by compositions of spatial and temporal operations. Often queries are also purely spatial or temporal, for example the query "Did this object exist at a particular date?" would only need to make use of a temporal index.

Finally, there is no current research on using separate index structures for indexing spatiotemporal data, and this approach could deliver interesting results.

6.2.2 Incorporating Indices

An important consideration for implementing index structures was how the indices would interact with data. The interface between moving objects and indices would have to be carefully designed so that objects do not have to be altered in order to be indexed. In other words an indexed object and a non-indexed object have to be the same in terms of their structure and in the way they are used. In many traditional relational database systems, a column in a table can be indexed by using the statement "*ADD INDEX*". Similar ease-of-use is needed for adding indices to this model. Also the interface of moving objects should not be limited to only a certain set of indices, but it should be flexible enough to allow new and different kinds of indices to be applied. If a spatiotemporal index is developed that has potential benefits, then it should be possible to replace existing indices without having to adapt or re-create any of the moving objects. Whether a temporal, spatial or spatiotemporal index is applied to the data, it should be done in a uniform manner.

With this ideal in mind, initial designs included an `IndexManager` class that would act as an intermediary between the moving object and the index(s). The `IndexManager` would be notified when a moving object needed to be indexed, and instructed which type of index should be used. It would create (or locate) the appropriate index and associate it with that moving object. A moving object would then have a reference to the `IndexManager` and notify it every time it moves, while the `IndexManager` would in turn update all the relevant indices (see Fig. 6.5).

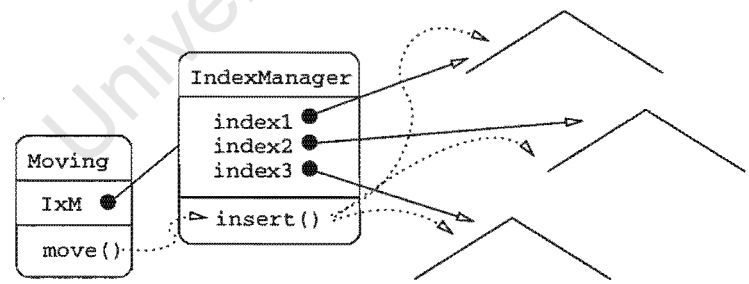

Figure 6.5: An `IndexManager` class proposed as an intermediary between moving objects and indices in an initial design.

It was observed that most of the method calls for the `IndexManager` would originate from the `Moving` class. For each method of `IndexManager` there was a method with a similar name in `Moving` that contained only a call to the related method in the `IndexManager` class. It seemed unnecessary to have such a group of methods that simply relayed messages. In a subsequent design the `IndexManager` class was removed since there was no

obvious reason for the index managing functionality to be in a separate class, and all the index handling methods were moved into the **Moving** class. The **Moving** class now had a number of additional attributes and methods as shown in Figure 6.6.

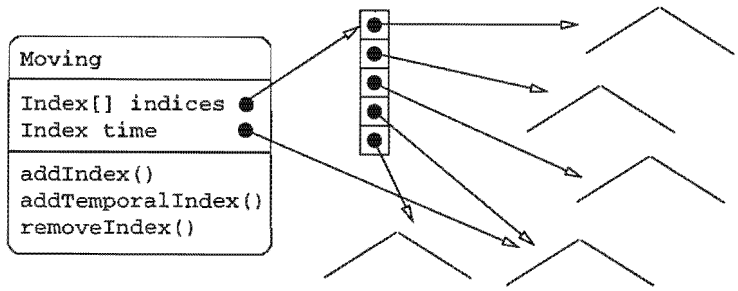

Figure 6.6: The index managing functionality was moved back to **Moving**.

The new **indices** attribute is a dynamic array of references to **Index** objects. **Index** is not a class but rather an *interface* that contains the method stubs that must be implemented by a class in order to be used as an index. If an existing index class is used, then rather than changing it to implement the **Index** interface, a wrapper class can be written that translates the methods required by **Index** to the original methods of the class (Figure 6.7).

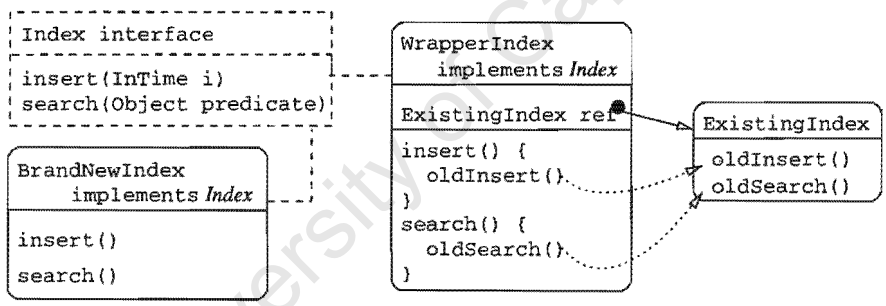

Figure 6.7: New or existing index classes must implement the **Index** interface before they can be used on moving objects.

This approach was used to employ existing B^+-Tree and R*-tree classes based on the GiST structure [HNP95]. For example:

```
public class BTreeIndex implements Index {

  Btree bt;  // existing GiST-based B-tree class

  public BTreeIndex() {     //constructor
      bt = new Btree();
  }

  public void insert(InTime i) {
      BtreeKey bk = new BtreeKey(i.inst().from(), i.inst().to());
      bt.insertNode(bk, i);
  }

}
```

Here the `Btree` is the original index class, and the last line of the example is a call to the insert method of `Btree`, from the wrapper class. Methods required by the `Index` interface take `InTime` objects as parameters and return collections of `InTime` objects. This is because an `InTime` is the smallest general building block of a moving object, which is appropriate to use for a general interface such as `Index` (i.e. the value part of an `InTime` may be spatial or non-spatial). Depending on whether the index will hold temporal or spatial values or both, its methods can unpack what they need from the `InTime` (in the example above, only the `Period` component of the `InTime` is used). It is also useful that collections of `InTime` objects can readily be installed in newly constructed `Moving` objects, since the history of a `Moving` is a `Collection` of `Intime`-s.

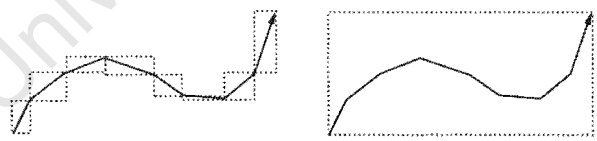

Figure 6.8: Dead space can be reduced by indexing smaller components.

As an example, Figure 6.8 illustrates why the design should allow the smallest possible components of moving objects to be indexed. Using a Minimum Bounding Rectangle (MBR) approximation in a spatiotemporal index, if a moving object was indexed as a whole, its approximation would contain a large amount of dead space. However if its `InTime` components were separately approximated and indexed, the total amount of dead space would be drastically reduced. Although the index structure would be larger as a consequence, its performance would be significantly improved.

6.2.3 Using Indices in a Moving Object

The `addIndex()` and `removeIndex()` methods (Figure 6.6) respectively adds or removes a reference to an index from the `indices` array. This array is a structure that contains references to all the indices over the history of this moving object. A new reference is added to `indices` by the `addIndex()` method. The reference may point to an index that was created to hold only the history of this moving object, or it may reference an existing index that holds the histories of many moving objects. Such global indices are discussed further in the next section. After adding the reference to `indices`, the `addIndex()` method also adds the entire history of the moving object to the index. The `removeIndex()` method simply resets a given reference in `indices` to `null`, and unless the index is elsewhere referenced, it will be garbage collected. Through this mechanism it is possible to add or remove indices from a moving object without affecting its structure (achieving the aim that was set out early in the design). The `move()` method is adapted to update each of the indices in the array whenever the object moves. It contains the additional code fragment:

```
for (int k=0; k<indices.length; k++) {
  indices[k].insert(next);      // update each index
}
```

This ensures that each index in the `indices` array is kept up-to-date when the history of the object changes.

Figure 6.6 also shows another reference to an `Index` called `time`, which serves as a shortcut to one of the elements of `indices`. It is a temporal index that contains the history of this moving object and none other. The `time` reference can only be set by the `addTemporalIndex()` method, which also creates the temporal index and calls the `addIndex()` method to add it just like any normal `Index`. Unless `addTemporalIndex()` has been called, `time` would be set to `null`. The `time` index has the specific purpose of enhancing the performance of temporal operations that search for `InTime` objects based on certain time intervals.

For example the `atPeriods()` method, which returns a subset of a moving object at specific time intervals, has been augmented to include the following code:

```
public Moving atPeriods(Periods per) {
  Moving result = null;
  // result = new Moving() using reflection ...

  if (time != null) {                    //
    result.intimes = times.search(per);  //  new
    return result;                       //  code
  }                                      //

  // ..else, search InTime components,
  // checking if their Period falls within
  // the argument per and adding it to result if it does
}
```

In a similar way, the `MovingSpatial` subclass has an additional attribute, a reference called `space`, which points to a spatial implementation of `Index`, and a method, `addSpatialIndex()`, that places the specialised *spatial* index in `indices` and `space` (just as `time` is used above). If a method like `at()` or `passes()` finds `space` to be non-empty, it can utilise the index to improve searching time.

Additional functionality can be incorporated in `move()` to test whether the history of an object exceeds a certain size, and then automatically call `addSpatialIndex()` or `addTemporalIndex()` in anticipation of deteriorating query times. For example:

```
public void move(InTime in) {
  ...
  if (intimes.size() > TEMP_THRESHOLD) {
    addTemporalIndex();
  }
  ...
}
```

where `TEMP_THRESHOLD` is some known value at which the size of the object's history makes the use of an index viable (see Section 7.3.3 for test results).

The indices added by `addTemporalIndex()` and `addSpatialIndex()` are specialised, e.g. `addTemporalIndex()` must add a *temporal* index that facilitates the search for time intervals. Therefore they are currently defined with no arguments in order to ensure that an appropriate index is added (B^+-tree for temporal and R^*-tree for spatial) and to simplify the process. These methods can be changed to accept any `Index` argument, but

then a check should be put in place to ensure that the given `Index` is appropriate. The `addIndex()` method however, does not have any restrictions on the type of index that it adds to the `indices` array, except that the index class implements the `Index` interface. Therefore it accepts any kind of `Index` as argument and does not need to perform any checks. Even when the index already exists and contains the histories of other objects (as discussed in the next section), this method is applied in the same way.

6.2.4 Indexing Multiple Moving Objects

In most spatiotemporal applications it will be necessary to address queries related to groups of moving objects, e.g. "Find *all* the vehicles that were inside a given region at time T_1" The operations that were discussed thus far pertain only to individual objects, and address the type of query: "Was *this* vehicle inside a given region at time T_1?". This is because operations were implemented as methods that belong to individual objects.

Selection queries, i.e. queries of the form "Find all objects in a set that satisfies some condition", are easily written using `Collection` objects and `Iterators` to return the required subset, for example:

```
Iterator given = givenCollection.iterator();
Collection resultSet = new Collection();
while {given.hasNext()) {
  Moving st = (Moving) given.next();
  if (st.satisfiesCondition()) {
    resultSet.add(st);
  }
}
return resultSet;
```


These language constructs and classes (since Java 2) are also at the disposal of the user of the model, and are sufficient for answering selection queries. For large sets of moving objects however, it would probably not yield acceptable query times. Therefore a more efficient access method is needed.

The mechanism with which an index is added to a moving object also allows for an *existing* index to be added[1]. This means that the history (`InTimes`) of this moving object are to be included in an external index used to access multiple objects. The indices in the previous section were intra-object, but moving items can also be added to inter-object

[1]The word "added" is used here because the reference is included in the `Moving` object. But since the history of the `Moving` object is also going to be "added" to the index, it should be seen as an *association*, rather than a composition.

indices. By calling `addIndex()`, a request is made to *insert* the history of this moving object into the given index, together with any other histories it may already contain.

Having `InTimes` of multiple moving objects in the same index presents a new problem. Although the index can efficiently locate any `InTime` or group of `InTime`-s, it is not possible to tell which `Moving` object(s) the `InTime`(s) belong to. A `Moving` object references the `InTime` objects that belong to it, but an `InTime` does not reference the `Moving` object *it* belongs *to*. An obvious solution would be to add an inverse reference to the `InTime` class, which points to the `Moving` that contains it. This circumstance is illustrated in Figure 6.9. However, adding the additional reference to `InTime` raises the concern that a

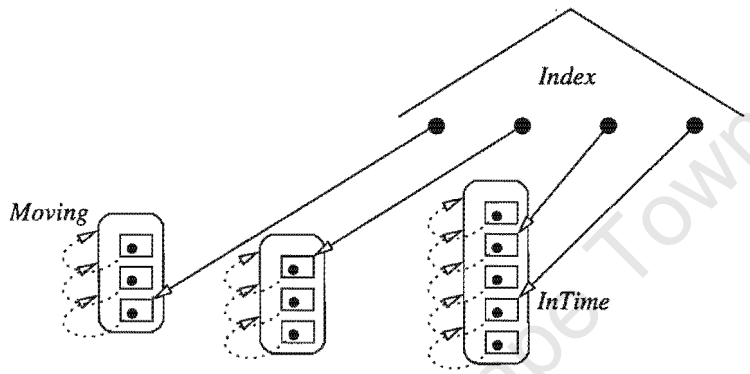

Figure 6.9: Including an inverse reference in `InTime` to identify the `Moving` object that contains it.

moving object would now contain an additional number of references equal to the number `InTimes` in its history. This raises an interesting question: "What is the cost?" Not only will indexed objects be affected by this change, but *all* moving objects will carry the additional cost.

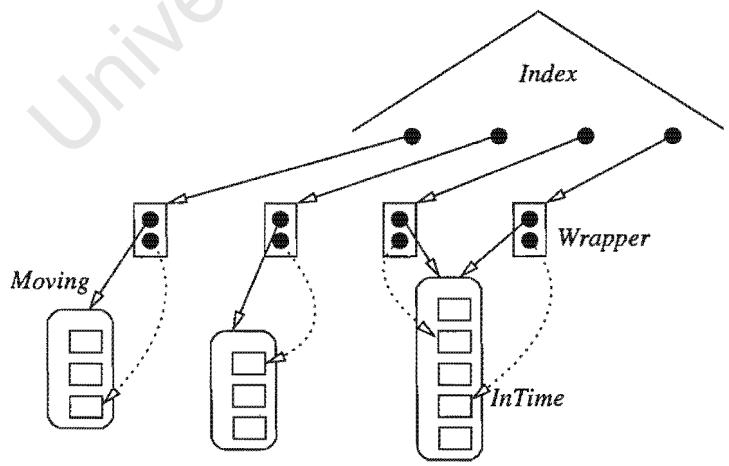

Figure 6.10: Using a wrapper class to determine which `Moving` object an `InTime` belongs to.

An alternative approach was considered, where instead of having `InTime` objects at the leaf nodes of an index, a small wrapper class could be used. The wrapper would contain a reference to the `InTime` that was indexed, and a reference to the `Moving` that contains it. This scheme, illustrated in Figure 6.10, would be more costly in terms of the storage requirement of the index, but it would not affect all moving objects in the database. The additional storage requirement would only be necessary for indices that hold references to more that one moving objects' histories.

Both the above approaches were implemented for comparison, and while implementing them, an interesting alternative arose. It was possible to create a subclass of `InTime` called `InTimeWrapper`, which contains an additional attribute that references the holding `Moving` object (Figure 6.11). Then only the `addIndex()` method in `Moving` had to be

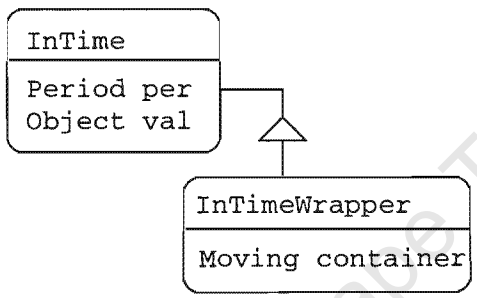

Figure 6.11: `InTimeWrapper` extends `InTime` and includes a reference to the `Moving` object that contains it.

changed[2] to pass the index an `InTimeWrapper` instead of an `InTime`. Even the `RSIndex` class that contained the original method header

```
public void insert(InTime in) { ... }
```

did not have to be changed, since an `InTimeWrapper` *is-a/extends* `InTime`.

If the `InTimeWrapper` did not inherit from `InTime`, it would be smaller, but then the ***Index*** interface would have to be changed so that the reference of the `Moving` object could be passed to the `insert()` method, e.g.

```
public void insert(InTime in, Moving mo) { ... }
```

This change also had to be made to the `move()` method of `Moving` from which `insert()` is called on each index in `indices`.

[2]The methods `addTemporalIndex()` and `addSpatialIndex()` did not have to be changed since they also call `addIndex()`.

It was however less costly in terms of storage space to use the no-inheritance route, since there would be fewer redundant references. This is made clear by the illustration in Figure 6.12, which shows that the inheritance option has to duplicate the original `InTime` objects.

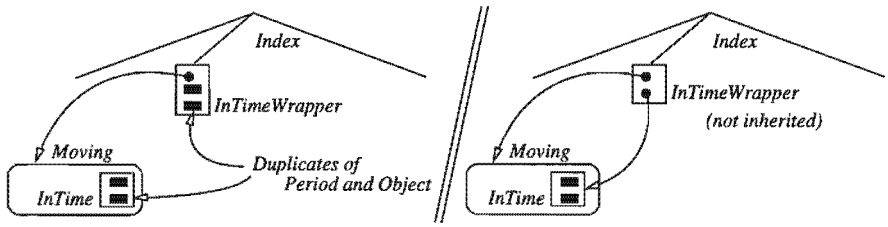

Figure 6.12: A wrapper class inherited from `InTime` (left) vs. a non-inherited wrapper class (right).

The results of the comparison is summarised in Table 6.1

Number of `InTime`-s in history	10	100	1000	
Normal moving region size (%)	100	100	100	A
Moving region with spatial index	155	148	146	B
With inverse ref. in `InTime`-s	102	102	102	C
With inverse ref and spatial index	157	151	148	D
`InTimeWrapper` subclass	162	155	153	E
`InTimeWrapper` not a subclass	160	153	151	F

Table 6.1: Relative storage requirement (using *percentage*) for different implementations of references to Moving classes in index structures.

From this one can see the storage cost of a spatial index is roughly 50% of the original size of the moving object. Using the technique of adding an inverse reference in `InTime`, the additional storage cost is around 2% of the original size of the moving object.

Using the wrapper classes inherited from `InTime`, the moving object is 5% bigger (in E) than in B.

$$\frac{E}{B} \times \frac{100}{1} \approx 1.05 \tag{6.1}$$

This should rather be seen as a 12%-15% increase in the size of its index, since a non-indexed moving object will not be affected in this case:

$$\frac{(E-B)}{(B-A)} \times \frac{100}{1} \approx 12\% \ to \ 15\% \tag{6.2}$$

If the smaller (non-inherited) wrapper classes are used, the increase in index size can be brought down to about 9%-11% (substitute E with F in equation 6.2). That would only be a 3% increase in the entire `Moving` object's size (substitute E with F in equation 6.1). From the result obtained it has been deduced that if more than 30% of all moving objects are indexed, there is a slight advantage to using the wrapper class approach.

6.3 Classification of the ST package

A number of criteria were proposed in [TSPM98] for the classification of spatiotemporal data and the methods used to index it. They are:

- *Data types supported:* Whether regions or only points are supported
- *Database handling of time:* Does it support transaction time, valid time, or both?
- *Data set mobility:* Whether spatial objects move, or the region of interest changes, or both (e.g. a ship at different locations vs. varying seismic activity in a fixed region).
- *Time-stamp update:* Can updates/insertions be made for any object at any time or does data have to be bulk-loaded, or does changes have to be made chronologically?
- *Object approximation:* Used, or not (e.g Minimum bounding rectangles/spheres).
- *Handling of obsolete entries:* Whether very old data can be purged and stored on tertiary media (only applicable to bulk-loading or chronological updates).
- *Specific query processing operations:* Supported, or not (e.g. selection-, join-, or nearest-neighbour queries).

According to this classification system, the library described in chapter 5 and the indexing approach presented here can be summarised as follows:

- Region, points as well as some base types are supported.
- Only valid time is used.
- The data set is fully dynamic.
- Updates are done chronologically (since moving objects are only updated by `move()`).
- Minimum Bounding Rectangles are used for approximation.
- No purging of obsolete entries is supported at present.
- Selection and nearest-neighbour query operations are available.

It is useful to be able to classify a spatiotemporal database or application, because not all kinds of configurations would be suitable for any application. It is also necessary to test a system to see whether its performance meets the requirements of the application. The following chapter discusses some results of experiments carried out to verify the usability of the ST package and its performance.

6.4 Summary

The indexing approach chosen for the spatiotemporal class library involves using separate spatial and temporal indices. Since all operations are either temporal or spatial, the appropriate index can be used as required.

An interface was designed as a generic template that can be implemented by any type of index class. The `Moving` class had some additional facilities included for the management of indices, such as the adding or removing of index structures from a moving object. In particular, a B^+-tree and an R^*-tree were implemented as temporal and spatial indices respectively, for the optimisation of certain operations. An index can be added to a moving object by calling a single method, after which updates take place transparently.

```
movingObject.addIndex(Index next);
```
This adds any index passed as an argument to the moving object.

```
movingObject.addTemporalIndex();
```
This adds a temporal index (B^+-tree) to the moving object.

```
movingSpatialObject.addSpatialIndex();
```
This adds a spatial index (e.g. R^*-tree) to the moving spatial object.

The possibility of adding indices automatically when moving objects reach a certain size was also explored, and some experimentation on the inclusion of inverse references from `InTime` to `Moving` objects revealed the best changes to make to `InTime` objects under different conditions.

Chapter 7

Experimentation

The main focus of this chapter is to give insight into the current usability of the classes designed to implement the spatiotemporal model. This entails both how easy it is to use the library and how the code performs. The first experiment presented here describes the experiences of users of the code and gives qualitative rather than quantitative results (that is beyond the scope of this project). Other experiments outlined in this chapter give straight forward results pertaining to query times and the use of indices. An initial section briefly illustrates some GUI components that were used during development for verifying correctness.

7.1 Tools Developed for Verifying Correctness

A few user-interface components were created for visually displaying the data held in spatial and moving spatial objects. The purpose of these components were mainly to assist with creating test data (e.g. to draw complex regions interactively) and also to check the validity of the output of operations during development.

Figure 7.1 shows the window frame drawn by the `TestArea` class. Spatial objects could be drawn on this window, selected, and turned into moving spatial objects by pressing the *move* button. This environment made development and testing significantly easier. For example when the `contains()` method of `Region` was developed, it was tested by visually checking the results of the method on the screen. A region would be drawn by clicking the button labeled *Region*, and then clicking on the white background to position the vertices (the region is automatically drawn as a closed shape). Then in a similar way a point is placed either inside or outside the region. The *Query* button calls a method named `query()`, which in this case would display the result of `region.contains(point)` on the status bar. Thus when clicking *Query*, the word `true` or `false` should appear depending on whether the region contains the point or not.

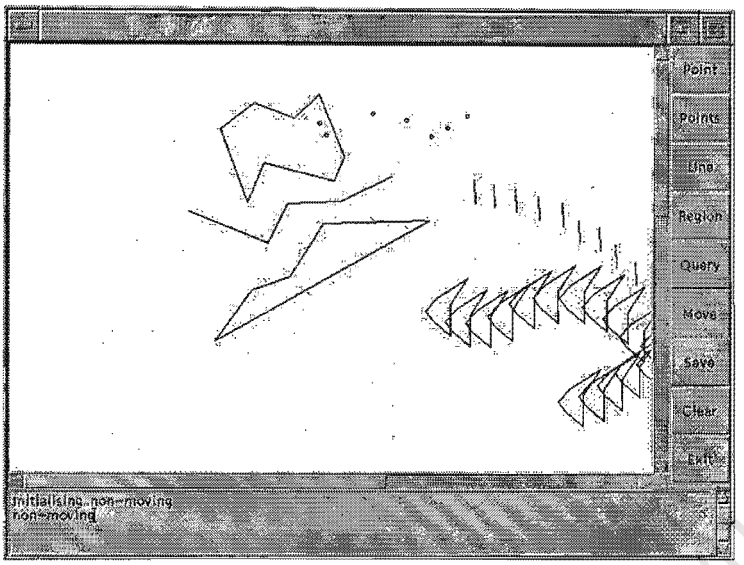

Figure 7.1: A `TestArea` object used for interactively drawing and displaying spatial and moving spatial objects.

Another GUI class that was created is called `Slider`, an instance of which appears in Figure 7.2. This is used to select and display time periods on a time line. Different granularities could be selected (e.g. second, hours, months) and it is linked with the `TestArea` window in such a way that moving objects' `Periods` are displayed when they are selected.

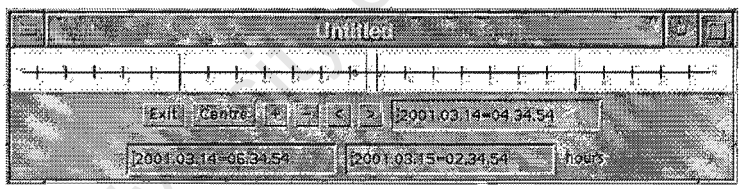

Figure 7.2: A `Slider` window used for selecting and displaying time periods.

For example a region that was drawn would be selected, and then by clicking the *Move* button, the region would be changed into a moving region with a randomly-generated history. The `query()` method, in this case containing a statement such as

```
movingregion = movingregion.atPeriods(myTime);
```

would refresh the display and show only a subset of the original moving object's history. This subset can be verified for correctness by selecting individual components in the result and checking their corresponding intervals in the `Slider` window.

These tools were used throughout the implementation of the model and found to be extremely useful. Although it was not the original intention to extend the model with visual display methods (as these are likely to be application-specific), it should be useful for developers to test applications built using the library.

Once the library had been built and tested for correctness using this GUI, it was employed in a prototype system for the Sea Fisheries Research Institute, so that queries of interest to real users could also be tested. Thereafter the usability of the library was examined by studying how others learnt and applied it in a similar situation. In the following sections a number of experiments are outlined that were focused at testing the overall usability and performance of the library.

7.2 Usability Experiment

It was important to have some measure of the ease of use of the library described in chapter 5. One of the main aims of this thesis was to provide a package or library that would be useful for developing spatiotemporal applications using a persistent object-oriented platform. This section describes an experiment that was aimed at verifying whether the tool is appropriate and easy enough to use for the development of such applications.

7.2.1 Background

Experiments for testing usability usually involve 2-6 participants and can take a number of different forms [Nie93]. There is the "Question-asking protocol" where users are asked to respond to particular questions while they are using a product, or the "Thinking-aloud protocol" where users can freely verbalise their thoughts about good or bad features while using the application. Alternatively such experiments can be structured around interviews and focus groups during which the users discuss their experiences *after* using the experimental application. For this experiment, participants had to develop code, which is a time-consuming process and thus the "Question-asking" and "Thinking-aloud" approaches would have been impractical. Therefore the latter approaches were followed where focus groups and interviews were conducted after the code had been written.

7.2.2 Description of the Experiment

For this experiment, three students performed a number of tasks that were set up in order to measure their productivity with the model. The three students chosen for the task had varying levels of experience in object-oriented programming, and different competencies.

- Student *A* was an undergraduate; an extremely good programmer who had been working with computers since an early age, and had a great deal of experience in object-oriented programming.

- Student B was another undergraduate, not experienced in object-oriented programming, but with a methodical approach to designing programs.
- Student C was an average programmer, but he was a postgraduate and had been exposed to more projects and research work than the other two.

The experiment was divided into three phases. During the first phase the students were required to develop a persistent Java application for keeping track of spatiotemporal oceanographic data. They were given a problem specification which was a simplified version of the specification given by the South African Sea Fisheries Research Institute in [SVZ98] (see appendix A). It entailed the design of an application for keeping track of ocean-going research vessels. These vessels would move across demarcated regions, collecting data about water temperature, salinity, fish larvae, oxygen levels, etc. There were also other "moving" devices for measuring the strength and direction of sea currents. These current meters were picked up by research vessels and moved to other locations. For the initial phase the students were given no hint of the spatiotemporal model or any kind of data structure to be used, and they had complete freedom in terms of the designs they chose for solving the problem. The only requirement was that a number of queries be answered by the programs they developed, most of which were spatiotemporal queries.

A short second phase was scheduled for them to familiarise themselves with the spatiotemporal model and to ensure that they were not confused about how to use it correctly. At the start of this phase they were given some material to read, which included the original paper by Güting *et al* that contains a detailed definition of the abstract model, as well as a preliminary version of chapter 5 of this thesis, and some illustrative examples like those in section 4.5. After allowing them to study the material over three days, they were presented with the exercise sheet in appendix B. This required that they showed their command of the "query language" of the model, i.e. that they could correctly construct queries using the operations supplied in the model. It also required them to create a subclass of a moving object, i.e. to use the model for recording data about some imaginary real-world moving object. The imaginary object also contained a *number* attribute that changed along with its position. They had to provide an implementation of the `trajectory()` method as well, and test their new class with some random data. All of the exercises in phase two were brief, and they were aimed at priming the students for the third and final phase.

Phase three had exactly the same problem description as phase one, except the students were now required to take a different approach. They had to start right from the beginning and re-create the application for the SFRI, however this time they had to do it using the spatiotemporal model. During each of the phases the students were required to record the time they spent on each aspect of each exercise, and to make notes of what they found particularly difficult or interesting.

7.2.3 Results

Phase I

Student A managed to create quite a good design which included a superclass for moving objects and a superclass for (time, position) pairs (very similar to the spatiotemporal model). He did however have trouble to create generic operations for his design. He mentioned that he had to alter his design in order to accommodate certain queries, although no particular details about this were given. The total time spent by him on this phase was 21 hours. His final `Test` class contains one method for each of the 9 required queries, where each method runs that specific query. This is similar to the approach that was used in [SVZ98].

Student B had done by far more design work than the others, but her inexperience with object-oriented programming made her progress much slower. In total she had spent 34 hours on this initial phase, during which also had to do a number of re-designs (in one case, for example, she forgot to incorporate time). Her final design had ships represented as `SeaObjects` on a 2-dimensional grid (raster) which was implemented as a two-dimensional array of which the index positions in the array represents the x and y coordinates of the sea vessel. Each object in the array also had a `Time` object as an attribute.

Student C designed the most general framework of the three. He had a class representing (time, position) pairs called `Data`, and such objects were grouped in `Vector` collections to represent the histories of moving objects (similar, in principle, to the collections of `InTime` objects in the spatiotemporal model) He had a `Reader` class with operations that perform some selection on a given `Vector` argument, which returns a subset of the `Vector`. This is similar to the way in which methods in the Moving class return Moving objects of the same type. However his `Reader` class was only a container for the set of general operations (e.g. `dataConditionRange()`, `dataInArea()`), and had to be instantiated separately, and applied to the moving objects (i.e. a `Reader` does not represent a moving object and its operations do not return `Reader` objects). He failed to record the exact time spent, but estimated it to be between 26 and 29 hours.

Phase II

After the initial three-day reading period, none of the three candidates had any problems with answering the queries on the exercise sheet. All of them completed the queries correctly in under 30 minutes. The only interesting phenomenon was that two of the candidates used binary operators such as < and <= in some of their answers rather than the methods such as `lessThan()` and `lessEqual()` that were used in the implementation of the model. From a focus-group discussion afterwards it was clear that their knowledge

of object-oriented language structure made the exercise easy, once they knew what each method (operation) did (i.e. `object.method()` $\Longleftrightarrow$ `spatiotemporalType.operation()`). The students also felt that the names of operations were not always self-explanatory, and could have been better chosen (e.g. `deftime()` and `at()`).

The greatest difficulty with this phase was experienced with designing the subclass of `mPoint`. Creating the trajectory operation was relatively easy and the students completed this without problems. However students B and C in particular had difficulty with adding the *number* attribute which changed together with the position of the moving point. They were not sure whether to add a number attribute to each `InTime` in the history (i.e. create a subclass of `InTime`), or whether to add an attribute of type `mInt` to the moving object's class (see Figure 7.3).

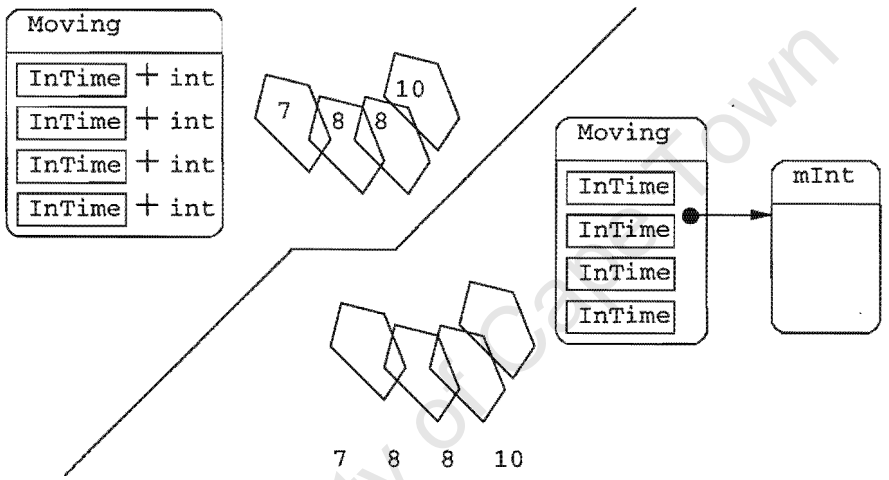

Figure 7.3: Students were uncertain about which approach to adopt for adding changing attributes to moving objects.

When this problem presented itself, the candidates were advised to try more than one approach and see what the advantages/difficulties were with each. They concluded in the discussion afterwards that they had found it easier to add the changing attribute as an additional moving number, rather than adding numbers to the history of the `mPoint` by subclassing `InTime` (i.e. the right side of Figure 7.3 was the preferred approach).

Phase III

The total size of source code produced by each of the students in phase III as a proportion of the code produced in phase I was 70%, 45% and 32% for students A, B and C respectively. The reason for this reduction was that they did not have to implement operations that were already in the model. Furthermore, they noted that they hardly had to spend any time on designing a framework for their code. They still had to write some routines for selecting a particular moving object among a number of moving objects, but

for the rest of the queries they only had to call the operations in the model. The students were also required to use the Persistent Java runtime environment developed at Glasgow University, to make their objects persist.

Student A had finished this phase in just over 5 hours, commenting that some of the classes in the model could do with additional constructors. For example a constructor in the `Period` class that takes a single `long` value as argument would be useful for constructing time intervals where the start and end values are the same (i.e. representing an instant).

Student B had spent in excess of 18 hours trying to complete as many queries as possible. She struggled with debugging her code, and did not finish all the examples in time. Some of the problems she encountered was dealing with `NullPointerExceptions` and incorrect results (all issues arising from inexperience). She also found it difficult to make her code persistent, since persistent roots were implicitly declared as static attributes - a language feature she had difficulty understanding.

Student C re-used his `Reader` approach as he did in the initial phase, however he re-designed it to inherit from the `Moving` class in the spatiotemporal model instead. There is nothing fundamentally wrong with using the `Reader` class as an intermediate class between the `Moving` superclass and the `Ship` subclass, but it does not serve a clear purpose in the design. The time he had spent on this phase was not recorded, however he did complete the exercise.

7.2.4 Conclusions

From the observations made throughout the running of the experiment the following conclusions were drawn:

- Using the spatiotemporal classes does alleviate the effort a programmer has to spend on designing frameworks for spatiotemporal applications; in our experience experimentation time is almost halved as a result. It also leads to significant reductions in the amount of code that needs to be produced.
- The code written using the model was also of better quality because it had improved structure and readability, compared to code written without the model. When examining the code of the previous VIBES project [SVZ98] it was found that objects were very similar to database tables originally used by the SRFI. The VIBES code also contained specialised query methods (much like students A and B had written in phase I) of which large sections were duplicated and that lacked potential to be reused. These methods returned lists of values on which no further operations are available. When using the model, the programmer is led to think of queries in

terms of operations on objects, which have better potential for reuse through the inheritance mechanism.

- Although there is a learning curve involved with using the model, the operations are intuitive and it takes a short time to learn how to use the package.
- Programmers that use the model need to already be proficient in Java (as the model makes use of concepts such as polymorphism) – it does not make Java programming any easier.

Because Student B struggled with mastering the Java language, her timing results do not accurately reflect the effect of using the model, and Student C did not submit complete feedback, therefore the most useful timing results were produced by student A. It indicated that he had spent 75% less time on developing an application using the spatiotemporal model. This does not account for the fact that he had already done an implementation, without the model, and had a certain degree of familiarity with the problem the second time (otherwise it may have taken him up to 50% longer, say 7.5 hours). Also, the time he had spent on familiarising himself with the model (4 hours) was not included. Taking these into account he still spent 40% less time using the model.

The main observations made from focus group discussions were:

- None of the students were able to make their designs generic in phase I. Each student found it necessary to change their design at least once to accommodate more queries.
- There were several similarities between the designs of the classes in the model and the designs of the students' classes. This shows that the model is relatively intuitive and can easily and naturally be used.
- Students felt that it is often preferable to write one's own code rather than to familiarize oneself with an existing API - even though writing your own code takes more time.
- Operations in the spatiotemporal model include some integrity checks (e.g. that periods in the history of an object do not overlap) that were not implemented by the students in their initial implementation – which would have taken them still longer to do.
- Although not evident at first, there are a number of different approaches that can be followed for adding changing attributes to moving objects. The question of which approach is better was resolved by experimenting with the various alternatives.

Even though the sample size of this experiment was very small, it was encouraging in that it demonstrated the usability and ease of learning of the model, and showed that productivity and code quality benefits can be expected. The next issue to study was therefore performance.

7.3 Measuring Query Times for Objects of Varying Sizes

The aim of this experiment was to determine the size of objects that could efficiently be handled by the package of classes described in chapter 5. It also aimed at determining what effect the temporal and spatial indices have on querying times.

There are two main factors that determine the physical size of a spatiotemporal object. One is the size of its history, i.e. how many `InTime` components does it comprise, and the other is its spatial complexity - how many vertices and edges it has.

7.3.1 Design of the Experiment

For this experiment, objects were created with history sizes of 200, 1000, and 5000 `InTime` components, and consisting of 10, 50 and 250 vertices. This gives nine possibilities in total, where the smallest object would have 10 vertices and a history of 200, and the largest would have 250 vertices with a history of 5000 movements (see Table 7.1). The choice of these numbers was motivated by observations made in a number of smaller tests done to verify correctness of certain operations. To put the size of this dataset into perspective, 10 vertices can be used to represent 10 points, 5 lines or 2 pentagons. In GIS systems, buildings may be represented as polygons with 4 to 20 vertices while roads may consist of 100's or even 1000's of vertices. A dataset of 1250000 vertices would therefore represent a medium sized database of *stationary* objects in an average GIS system.

		Number of Vertices		
		10	50	250
History	200	2000	10000	50000
	1000	10000	50000	250000
	5000	50000	250000	1250000

Table 7.1: The total number of vertices in spatial objects used in this experiment.

The objects were randomly created with their n vertices ($n \in \{10, 50, 250\}$) distributed over a square of area $10n$. All objects were made persistent and accessed from a persistent store. The experiment was run on a SUN Ultra 10 with a 440 mhz UltraSPARC IIi CPU, 2MB cache, 640MB EDO RAM and running Solaris 8.

One spatial query and one temporal query were run on each of the nine combinations of history size & vertex count. The spatial and temporal queries were kept separate because spatiotemporal queries are built up as combinations of spatial and temporal operations, for example

```
MovingSpatial ms = myObject.at(someRegion).atPeriod(somePeriod);
```

is a spatiotemporal query which first refines the moving objects in terms of some spatial criteria, then further refines it based on some temporal value. Although the object may contain a spatial or temporal index, it can only be used by the first operation in the query, because that operation builds and returns a new object without any indices. The next operation is called on the new object which returns yet another new object without any indices and so forth. Thus the choice of which type of operation to call first is important and it would depend on the relative speed of operations with and without indices. This experiment recorded execution times of spatial and temporal queries on objects with indices, and on objects without indices.

7.3.2 Results

The graph in Figure 7.4 shows the results of a spatial query on the nine differently composed objects (a log scale was used on the Y-axis for easier viewing). The solid line indicates query times without any index, the short-dotted line shows a temporal index being used and the long-dotted line shows a spatial index being used. The query was

```
MovingSpatial ms = testObject.at(Xregion);
```

where `Xregion` is a region that fills the entire space. This is the worst case scenario for a spatial query, because all `InTime` objects in the history would be returned. Nevertheless objects with spatial indices consistently had faster execution times, because a spatial index uses minimum bounding rectangle approximations, whereas an expensive `contains()` method has to be called in objects with no spatial index. A temporal index is not used at all by the `at()` operation, however its presence does seem to affect query times for objects with small histories. This may be due to the difference in the number of disk blocks that an object with a temporal index spans. Spatial queries on objects with larger histories are less affected by temporal indices.

The following temporal query was used to obtain the results illustrated in Figure 7.5:

```
MovingSpatial ms = testObject.atPeriod(Xtime);
```

where `Xtime` is a random `Period` that covers at least half the total time range of the moving object's history. In this case the spatial index is not used by the `atPeriod()` operation and does therefore not have any positive effect. Results in Fig. 7.5 show that objects with temporal indices do have faster query times. This is more evident with the large objects because the time scale of small objects are so small that external influences like process scheduling by the operating system could have a marked effect (the query time scale is less than 100 milliseconds).

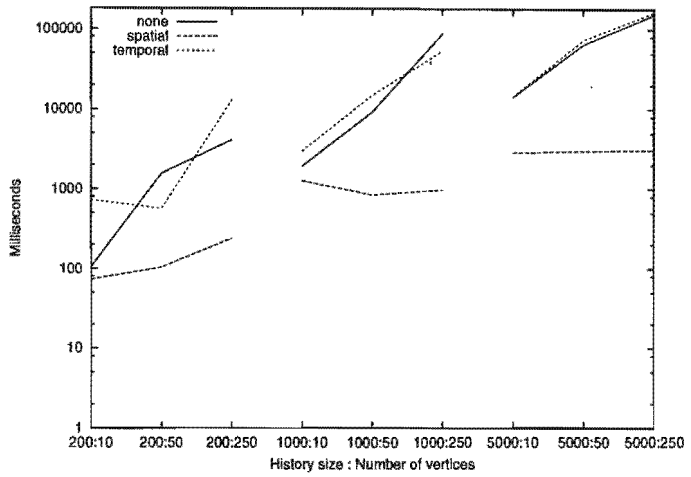

Figure 7.4: Results of a purely spatial query on spatiotemporal objects of varying sizes (number of movements : number of vertices).

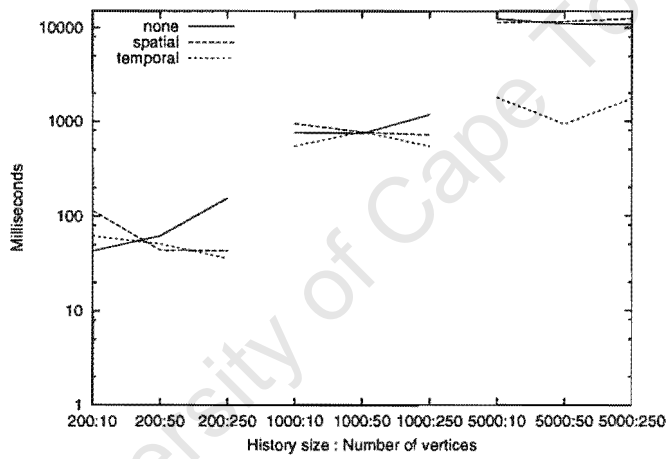

Figure 7.5: Results of a purely temporal query on spatiotemporal objects of varying sizes.

7.3.3 Weighing Up the Cost and Benefit of Indices

Creating and using an index comes with some cost in terms of storage space and insertions times. The spatial and temporal queries in the preceding section were each run 7 times, every time creating a new persistent store containing different random objects. Also, new indices were created on the objects each time, and the creation times have been recorded and averaged out over the 7 runs. These are shown in Figure 7.6 for spatial indices and Figure 7.7 for temporal indices.

In a separate experiment a number of moving spatial objects consisting of 4 vertices were created and moved between 10 and 5000 times (i.e. their histories contained 10, 25, 50, 100, 200, 500, 1000, 2000, and 5000 `InTime` components). Two simple queries similar to the ones shown above, one spatial and one temporal, were run once with an index and

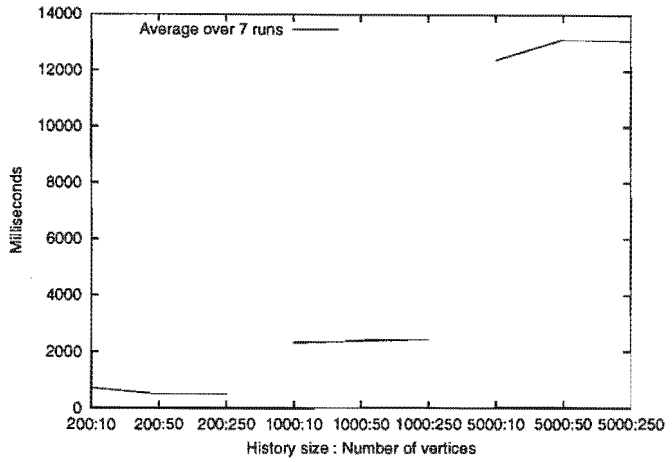

Figure 7.6: Average time for creating spatial indices for objects of different sizes.

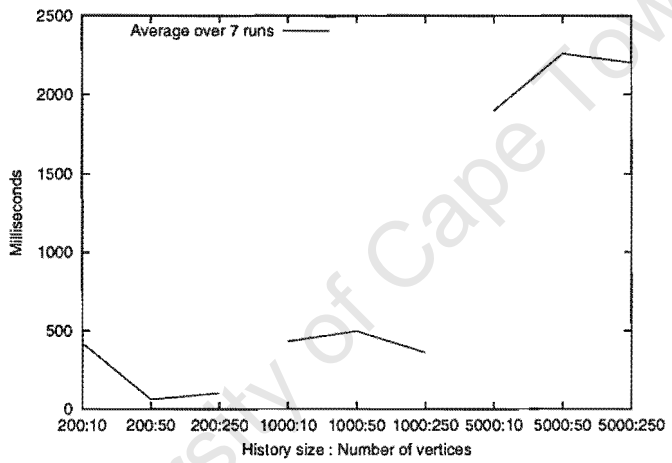

Figure 7.7: Average time for creating temporal indices for objects of different sizes.

once without. The aim was to determine what size an object's history needs to be, for a particular index to become viable. The results shown in Figures 7.8 and 7.9 indicate that objects with as few as 20–25 `InTime`-s in their history, have faster query times resulting from index usage.

7.3.4 Conclusions

Both the size of an object's history and its complexity in terms of the number of vertices it comprises, have marked effects on spatial query times. Objects with a history size of 5000 have acceptable spatial query times only if they contain a spatial index.

Temporal queries are faster than spatial queries, and are not affected by the number

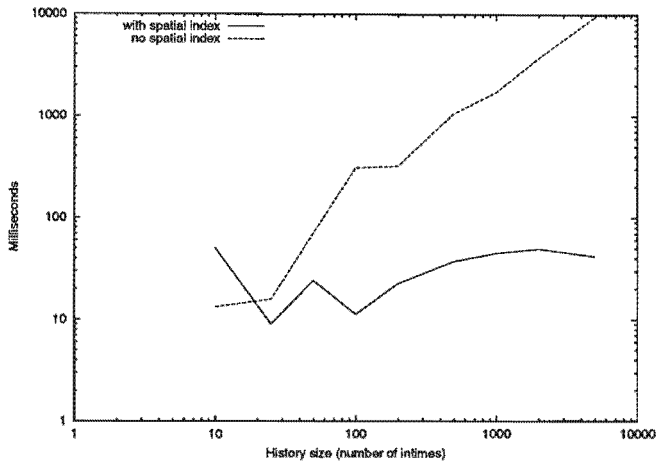

Figure 7.8: Spatial query times of objects with and without a spatial index.

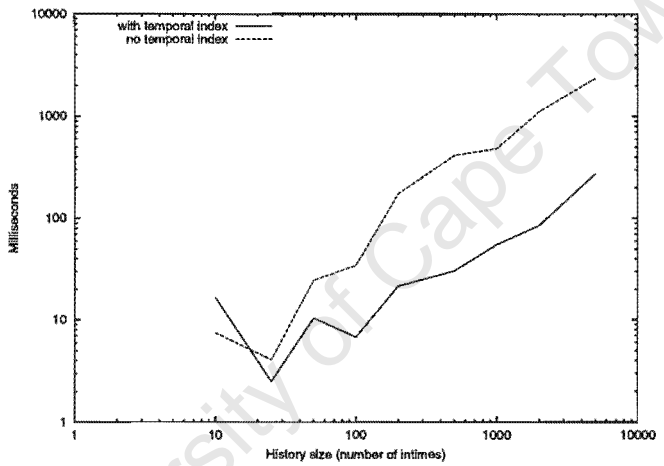

Figure 7.9: Temporal query times of objects with and without a temporal index.

of vertices of an object, rather by its history size. Also, temporal query times become unacceptable if an object's history contains 5000 `InTime` components and a temporal index is not employed.

Index creation times are significantly greater for spatial indices than for temporal indices. Creating a spatial index on a history smaller than 1000, or a temporal index on a history smaller than 5000 takes less than 3 seconds. On an object with history >= 5000 it is still faster to both build a temporal index *and* run a temporal query on it, than running the query with no index. Therefore, because temporal index creation is faster than spatial index creation, a general rule of thumb for spatiotemporal queries is that the spatial index should be used first if both are available (i.e. a query should take the form `obj.at().atPeriod()` rather than `obj.atPeriod.at()`), so that a temporal index can be built on the intermediate result.

7.4 Experiment Comparing a 3D R*-tree Approach and a 2D R*-tree & B+-tree Approach

This experiment was carried out to compare two different indexing approaches. The one approach uses a two-dimensional R*-tree as a spatial index and a B+-tree as a temporal index (as was the situation in section 7.3.3). This approach was suggested in chapter 6 for reasons explained in section 6.2. The other approach uses only a single 3-dimensional R*-tree where time is incorporated as a third dimension. The aim of the experiment was to establish whether the use of separate indices are favourable when purely spatial or temporal queries are made, and whether the 3D index performs better for spatiotemporal queries.

7.4.1 Design of the Experiment

As mentioned in chapter 6, an index could be built over a single moving object or over multiple moving objects. The experiment was divided into two parts, one that measured the effectiveness of indices on a single object with history size 50000, and the other investigating indexing of at least 50 moving objects, each with a history size of 1000.

Four different time periods were used in the queries, namely the whole period for which objects were defined, a **long** period (about 80% of the whole period), a **medium** period (40%) and a short period (10%). Also four different spatial regions were used, namely the entire region in which objects moved, a **high** region that is thin and rectangular and stretches from the top of the area to the bottom, a **wide** region that is flat and stretches from left to right, and a **small** region in the middle of the area. Every combination of region and period was used, except for the combination involving both the entire period and entire area, because that would return the entire test space. The query regions in space and time are illustrated in Figure 7.10.

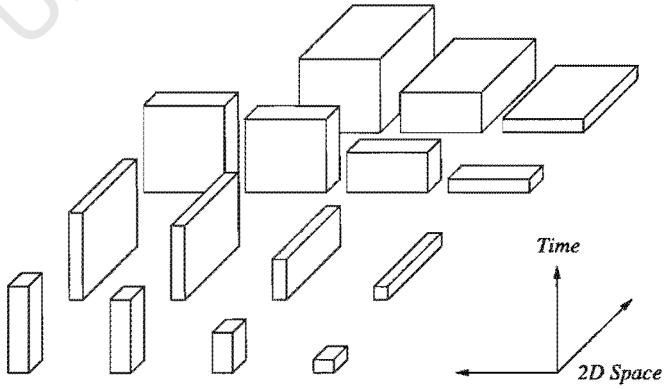

Figure 7.10: Query regions in space-time used in the experiment.

During ten successive runs, objects were created and placed in a persistent store, and indices were added before running queries from a separate program. The query that was answered is

"Find all objects within region X during time T"

although in the code it was expressed differently for the two indexing approaches.

7.4.2 Results

The results of the ten runs were averaged and are shown in the tables below. Table 7.2 shows the query times in milliseconds for the single large object. All query times were acceptable, but the 3D R*-tree most often performed better than the combination of 2D R*- and B^+-trees. The results in the top row of the table show that the separate indices performed better – this is because it is a purely temporal query (i.e. for all locations in space) and could be done using only the B^+-tree.

	*	long	medium	short	
*		7312	5906	3906	3D R*
		4686	3283	176	2D&B^+
wide	845	685	431	295	3D R*
	889	1298	1092	668	2D&B^+
high	665	604	516	245	3D R*
	744	1074	939	574	2D&B^+
small	172	166	141	67	3D R*
	139	153	163	115	2D&B^+

Table 7.2: Comparison of spatiotemporal query times (in *milliseconds*) between a 3D R*-tree and a 2D R*-tree + B^+-tree over a *single moving object.*

Table 7.3 holds the query times of indices over multiple objects (50 in this case). Using an index to query multiple objects involves more computation, because the indices only return `InTime` objects (see section 6.2), and the `Moving` objects containing them must be located in an additional step. This extra sifting process was not done very efficiently (using linear search) and became more costly as queries returned more `InTimes`.

The separate indices once again did better than the 3D-R*-tree for purely temporal queries (top row), but it fared much worse over longer time intervals (see the column headed **long**). The reason is that the separate indices need to sift through all the `InTime` objects returned by both spatial and temporal indices, keeping only the ones that exist in both groups. The more `InTimes` there are, the longer this process takes.

	*	long	medium	short	
*		19289	15453	10485	3D R*
		12350	6080	1408	2D&B+
wide	2730	2645	2503	2221	3D R*
	2412	15113	8874	3629	2D&B+
high	2292	2208	2104	1775	3D R*
	2560	15033	8673	3633	2D&B+
small	367	353	348	315	3D R*
	433	13015	6697	1830	2D&B+

Table 7.3: Comparison of spatiotemporal query times (in *milliseconds*) between a 3D R*-tree and a 2D R*-tree + B+-tree over *multiple moving objects.*

7.4.3 Conclusions

The separate-index approach needs to be made more efficient before it can be used as an alternative to the 3D index. A way has to be found to avoid "losing an index" in queries that involve compositions of several spatial and temporal operations, where the successive operations rely on using indices for efficiency. One solution might be to have operations reconstruct indices on the objects they return – but the viability of such a technique still needs to be explored. Although the workloads used in this experiment were purely synthetic, the objective was to show the relative performance of indices in query regions of different shapes and sizes.

7.5 Summary

This chapter introduced the GUI tools that were used for checking that operations return correct results. Although they were not initially intended to be included in the package, they may be useful for developers that use the spatiotemporal classes.

The usability experiment showed that the model is relatively easy to understand and use. The structure it gives to spatiotemporal applications seem to result in better quality code, however the programmer needs to be proficient in applying advanced programming techniques such as polymorphism.

A spatiotemporal object has two attributes that have a significant effect on its size, namely the number of times it moves (history) and the number of vertices it has. The experiment in section 7.3 showed that spatial and temporal indices have a positive impact on spatial and temporal queries respectively, without which queries on objects with over 250 vertices and 5000 moves are unacceptably slow.

The overhead of maintaining an index was found to be relatively small, which makes

it viable to add an index to objects with histories containing as few as 30 moves (section 7.3.3). Two approaches for indexing spatiotemporal data were compared, one that uses a 3D R*-tree, and another that uses separate spatial and temporal indices. The two-index approach was found to be less efficient in many cases, partly due to the fact that moving objects created in intermediate steps of composite queries are not indexed.

Chapter 8

Conclusions

8.1 Summary

Spatiotemporal databases are a relatively new phenomenon that started gaining attention only during the past decade. Temporal databases and spatial databases have however been around for much longer, and the technologies developed in these two separate fields are extensively applied within spatiotemporal systems. In chapter 2 some background was given on spatial and temporal databases, and a few existing spatiotemporal models that aim to unify the spatial and temporal domains were outlined.

The particular spatiotemporal model that was used in this thesis was designed as part of the Chorochronos research project [Chr96]. The design goals of this model were to achieve closure and consistency, and to make the model expressive, yet simple and powerful. It consists of a small number of data types, which are well-defined, and a number of operations that are defined on those types. Using the model is uncomplicated and there are no special cases where, for example, some operations are used in an inconsistent way, or deliver unexpected results. Because of this, and because the number of types and operations have been kept low, the model is easy to understand and learn. Our own experiments and use of the model confirmed this. The model can be applied to any type of spatiotemporal data (i.e. not only in GIS systems), since it does not contain any domain-specific data types.

All of the data types of the model, as well as a representative set of operations, were implemented and packaged into a library of Persistent Java classes. The package also has an indexing mechanism built in, which operates through a well-defined and general interface. This allows any type of indexing structure to be used to index and look up moving objects efficiently. With many more spatiotemporal indices that are expected to be developed in the near future, such a general interface would be useful for comparing the performance of various indexing approaches.

The library was used to build a prototype application for the Sea Fisheries Research Institute which was then compared with an earlier implementation that used Persistent Java without the library [SVZ98]. In another experiment, students implemented a similar PJama application firstly without and then with the library. These experiences indicate that both productivity and code quality improve when the library is used.

8.2 Thesis Contribution

In order to demonstrate the degree to which this thesis has attained its goals, a number of questions that were posed in the introduction are addressed in the ensuing paragraphs.

> *What is needed for a library that can aid the rapid development of persistent spatiotemporal applications?*

Before considering what set of classes the library might ultimately comprise, we thought that finding a suitable spatiotemporal model would be a logical starting point. A model that has complete, precise definitions of types and operations would provide initial guidelines for dealing with spatiotemporal data. From that a class library could be constructed that contains a comprehensive set of object classes forming the toolset to be used for building complex applications. Having identified the [GBE+00] model as an ideal candidate it was found that its types and operations mapped naturally to object classes and methods. Since it is a continuous model, implementing the types as object classes required the formulation of a discrete representation for each data type. The initial set of classes that were created as a result underwent a series of design changes. This underlines the fact that a strong object-oriented design is hard to get right the first time. As mentioned in [Eck98] it is often only after about the third redesign that a first-rate class structure is distilled.

The class library that emerged from this design process contains a closed set of data types (owing to the spatiotemporal model), which means that any data type generated by the library is also contained in it. In addition, the library has an indexing interface which ensures that an acceptable level of performance of operations can be maintained. Spatial and temporal classes are composed from smaller parts which means that extending types at any level can be done with ease and objects are mutable (e.g. the vertex-edge representation of spatial types enables moving spatial objects to change shape throughout their existence). Through the use of object-oriented features such as inheritance and polymorphism, the overall design could be greatly enhanced and the total size of the spatiotemporal package (in terms of lines of code) is significantly less than it would have been without generalised superclasses. Thus structural optimisations could be made with a minimal amount of effort (such as replacing `ArrayList` with a lighter structure) and the class library is more manageable for future updates.

We conclude that a persistent Java platform is suitable for spatiotemporal systems as long as an appropriate class library is available. Such a library needs to be based on a closed, consistent spatiotemporal model, to provide an interface for optional indexing of moving objects, and to offer a selection of indexing mechanisms that implement this interface. We note also that object-oriented languages offer some distinct advantages to both developers and users of such a library.

> *How does using persistent Java contribute to productivity and are there any trade-offs in code quality or reusability?*

The Java language lends itself to reduced development time since the programmer does not have to deal with issues such as memory management, thus more time can be devoted to creating better program designs. Java also has strong type checking rules that are enforced through an integrated exception handling mechanism, which guides the programmer well towards creating robust and versatile code. Everything is declared as objects in Java, which makes it less confusing than a language such as C++ where objects and individual functions can be used together. Java does not support multiple inheritance (which proved to be only slightly inconvenient when we couldn't inherit from the AWT's `Point` class), but this is compensated to some degree by the use of interfaces, with which specific structures can be imposed on classes. Java APIs such as the reflection library which provides Run-time Type Identification made it possible for important methods to be generalised in superclasses.

Persistent Java does not exclude any of the benefits of Java, but it does hold the additional feature of providing objects with the ability to persist past the execution time of a program. This comes at practically no additional effort for the programmer. There is no distinction between persistent and non-persistent objects, and the code written for persistent Java can also run on normal virtual machines. When the PJama virtual machine [AJDS96] is used, all static attributes are rendered as persistent roots, and all objects reachable from these roots also maintain their state in the persistent store. Data in the store does not have to be decomposed into tables or files and objects do not have to be translated by the program in any way when being moved between storage and memory. Most of the development of the spatiotemporal library was done with the normal Java Runtime Environment, and classes required no alteration when they were finally made persistent. Although persistence makes the programmer's workload lighter, it was found to have somewhat of a learning curve. Later versions of PJama require the use of configuration files which are not always clear for a first-time user. Also, the way in which objects are organised in the persistent store is not transparent to the programmer, and thus it is not always possible to investigate the effect that changes to an object's structure have on efficiency (for example when adding inverse references to moving objects). When optimising class structures, information such as object size, the number of disk blocks used, and the clustering of objects on disk would be useful for the designer.

How does a library facilitate the process of building a spatiotemporal application?

The ST package guides the design of spatiotemporal applications so that the application programmer only needs to consider *what* data will be stored as moving types, and not *how* they should be stored. We also found that using the spatiotemporal classes improves the structure and quality of code in a spatiotemporal application. The need for such a framework was clear in the VIBES project [SVZ98] where a great deal of time was spent on designing structures that hold spatiotemporal data. The class package produced by the VIBES project lacked generality it its design, and has little potential to be used in similar future projects.

Having a library that serves as a framework for building applications is especially valuable when programming with a persistent language. There is no particular way in which persistent data is organised – any object reachable from a persistent root will persist, therefore the structure of data on the store can take any form. Each object is stored with its own unique structure, unlike in a relational database for example, where data is organised into normalised relations. It is therefore desirable to have some guiding structure for organising data because if that can be facilitated, then using persistence holds a considerable advantage over other secondary storage methods.

Since the library forms a super-structure of classes that application-specific classes inherit from, it is more likely that the application classes can be reused in future. Separate applications that have been built using the ST library can therefore easily be made to interface with each other.

8.3 Future Work

As the model is designed to encapsulate all spatial and temporal operations, building and optimising the complete set is a mammoth task. In this thesis all the types as well as a representative sample of the operations were implemented on order to examine the usefulness of the library. The remaining operations that have not been implemented are mostly the 'lifted' operations on temporal types. Adding those methods would be the next step in continuing work on the ST package. Some operations currently also have very simplistic implementations, such as `area()` in `Region`, which only calculates the area of the bounding box of a region. Such methods can be implemented using more sophisticated algorithms, for example triangulation can be used in `area()`. Other operations such as `intersect()` in `Line` may be optimised to perform fewer iterations when checking whether two lines intersect, by checking line boundaries before doing the comparison.

It would also make sense to modify the spatial classes to handle 3-dimensional data,

since there are many applications that deal with 3-D spatiotemporal information. The difficulty here is not to add the extra dimension, because that would only require adding another (z) coordinate to the `Point` class. The main difficulty with handling 3 dimensions it that spatial operations become significantly more complex.

Because not many commercial applications have been written using persistent languages, valuable experience may be gained from writing applications that use real spatiotemporal data (as was the intention of [SVZ98]). It would be particularly interesting to measure the degree by which the use of the ST package expedites the development process. Such projects would also provide suitable testing platforms for future spatiotemporal indexing techniques.

The ST package will be available at http://www.cs.uct.ac.za/ evoges/ST/ for a limited time after the submission of this thesis.

Appendix A

Phase I: SFRI problem description

The South African Department of Sea Fisheries receive large quantities of data from a number of research vessels. This data needs to be stored in a database in such a way that some complex queries can be answered efficiently. Your task is to design such a database, based on the detailed explanation of their data below.

A.1 Research Vessels

Every research vessel in the fleet has the following information:

- a name
- a unique callsign (e.g. "VL224")
- number of crew on board
- top cruising speed

Depending on the equipment on board a vessel for a specific cruise, the vessel can take on one of two "profiles".

1. In the first profile, a the ship would contain instrumentation to record
   - water temperature,
   - salinity,
   - the number of fish larvae for a given region,

- and the speed and direction of sea current.

2. In the second profile, the instrumentation on board will measure
   - oxygen,
   - phosphorous,
   - incident light,
   - as well as depth of the ocean floor.

It may be of some importance to know what profile a ship has at any given time.

A.2 SARP Regions

Certain regions have been demarcated in the ocean as "SARP" regions. These are areas where measurements have been taken consistently over a period in time. A SARP region is defined by minimum and maximum longitude and lattitude values which form a rectangle. These coordinates can also change at any given time. Any vessel which enters a SARP region should have its measurements associated with that region for the duration of time its spends inside the region.

A.3 Current Meters

A device has been designed for measuring the strength and direction of sea currents at some point in the ocean. This is not the same as the equipment used on ships to measure currents, but it records the same type of data. These devices can be picked up by any ship at any time, and dropped of at some different location. Of course, whenever a current meter is moved, such information must be recorded.

A.4 Satellite Images

Satellite images of the entire region are captured by orbiting satellites at different, random times every day. Each image together with its timestamp must be stored in the database. These images are often analysed by researchers who try to find correlation between visual patterns and measured data.

A.5 Queries

The following are typical examples of queries that need to be answered by the database:

- When was the ship "Winterberg" inside SARP region X4?
- What was the average temperature recorded by the vessel "Valdez" while it was inside SARP region X2?
- Has the vessel "Treasure" every been inside SARP X6 during the summer?
- During which periods was the current meter C4 in transit?
- Find the average direction of sea currents in SARP area X1 for August 1998.
- What is the minimum time it will take for a vessel called "SeaFlower" to reach the currentmeter at a given posistion?
- Find all the satellite images that were available during January on days when larvae counts of more than 100 were measured.
- How many times did the "SeaFlower" change profiles in 1997?
- When where there at least two profile "A" ships in SARP region X3 ?

Appendix B

Phase II: ST Model Exercise Sheet

B.1 ST Model Queries

Assume there is a moving point (mp), an moving line (ml) and a moving region (mr) all moving around randomly, somewhere in space. Using their names (mp,ml,mr), Write down quries that would answer the following questions.

- "Where was mr at time X?"
- "When was mr exactly 5 units away from mp?"
- "Did mp intersect ml before it intersected mr?"
- "Was mp inside mr while its area was greater than 30 units?"
- "Was mp inside mr while its area was decreasing?"

What would the return type of the following query be?

```
ml.at(mr)
```

B.2 Using the ST Model

Create a new subclass of the `mPoint` class, and add a method called `trajectory()` to it. The method should return a `Line` which represents the route that the moving point followed.

Also add an additional attribute called `number` to your class, of type int.

Use the `TestArea` class to generate some test data, and add some randomly changing values to an instance of your new class.

Test your trajectory method, and try to find different ways of answering the query "Where was the moving 'thing' when its number attribute increased by more than 5 units?"

Bibliography

[AJDS96] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. An Orthogonally Persistent Java. In *ACM SIGMOD Record*, December 1996.

[AK00] Masatoshi Arikawa and Koichi Kubota. A Standard XML Based Protocol for Spatial Data Exchange. In *Proceedings of the International Workshop on Emerging Technologies for Geo-Based Applications*, pages 37–45, 2000.

[All83] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), August 1995.

[BBK98] Stefan Berchtold, Christian Bőhm, and Hans-Peter Kriegel. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. Technical report, University of Munich, 1998.

[BKK96] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High Dimensional Data. In *22nd Conf. on Very Large Databases*, pages 28–39, 1996.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 325–331, 1990.

[CFvO93] E. Clementini, P. De Felice, and P. van Oosterom. A Small Set of Formal Topological Relations Suitable for End User Interaction. In *Proc. 3rd Intl. Symposium on Large Spatial Databases*, pages 277–295, 1993.

[Chr96] The Chorochronos Spatiotemporal Research Project, 1996. http://www.dbnet.ece.ntua.gr/čhoros/.

[CP00] Christophe Claramunt and Christine Parent. Modelling Concepts for the Representation of Evolution Constraints. In *Proceedings of the International Workshop on Emerging Technologies for Geo-Based Applications*, pages 169–184, 2000.

[Eck98] Bruce Eckel. *Thinking in Java.* Prentice Hall, 1998.

[EGV97] M. Erwig, R. H. Güting, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modelling and Querying Moving Objects in Databases. Technical report, Chorochronos, 1997.

[Fra98] Andrew U. Frank. Different Types of "Times" in GIS. In Max J.Egenhofer and Reginald G. Gollegde, editors, *Spatial and temporal reasoning in Geographic Information Systems*, chapter 3. Oxford University Press, 1998.

[FvDFH90] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, 1990.

[GBE+00] R. H. Güting, M. H. Boehlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *TODS*, 25(1):1–42, 2000.

[Gő94] R. H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3(4), October 1994.

[GLOT96] Cheng Hian Goh, Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. Indexing Temporal Data using Existing B^+-trees. *Data & Knowledge Engineering*, 18:147–165, 1996.

[GM96] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems (java.sun.com/docs/white/langenv/), 1996.

[Gre99] Brian Greene. *The Elegant Universe.* Vintage, 1999.

[GS93] R.H. Güting and M. Schneider. Realms: A Foundation for Spatial Data Types in Database Systems. In *Proc. 3rd Intl. Symposium on Large Spatial Databases*, pages 14–35, 1993.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD International Conf. on the Management of Data, Boston, MA*, pages 47–52, 1984.

[Haw88] Stephen Hawking. *A Brief History Of Time.* Bantam Press, 1988.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalised Search Trees for Database Systems. Technical report, University of Berkeley, California, June 1995.

[Hub00] Martin Huber. Geography to the Server: Geographic Databases, Internet Mapping, Geographic Application Services. In *Proceedings of the International Workshop on Emerging Technologies for Geo-Based Applications*, pages 329–333, 2000.

[Kel94] Stephen H. Kellert. *In the Wake of Chaos: Unpredictable Order in Dynamical Systems.* University of Chicago Press, 1994.

[KGT99] George Kolllios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On Indexing Mobile Objects. In *Proc. of the PODS Conf.*, pages 261–272, 1999.

[KIPS95] Stavros Kokkotos, Efstathios V. Ioannidis, Themis Panayiotopoulos, and Constantine D. Spyropoulos. On the Issue of Valid Time(s) in Temporal Databases. *SIGMOD Record*, 24(3), September 1995.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbour Queries. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 1997.

[KTF98] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bi-temporal Databases. *IEEE Trans. on Knowledge and Data Engineering*, 10(1), 1998.

[Lan92] Gail Langran. *Time in Geographic Information Systems.* Taylor & Francis, 1992.

[MIA96] Stewart D. Macneill, Tony J. Ibbs, and Malcolm P. Atkinson. The JUGGLE Project Report. Technical report, University of Glasgow, http://www.dcs.glasgow.ac.uk/juggle/report/report3.html, 1996.

[MWH] John W. McKenzie, Ian P. Williamson, and N. W. J. Hazelton. 4-D Adaptive GIS: Justification and Methodologies.

[Nie93] J. Nielsen. *Usability Engineering.* Academic Press, 1993.

[NST98] Mario Nascimento, Jefferson Silva, and Yannis Theodoridis. Towards Historical R-Trees. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pages 235 – 240, February 1998.

[NST99] Mario Nascimento, Jefferson Silva, and Yannis Theodoris. Evaluation of Access Structures for Discretely Moving Points. In *International Workshop on Spatio-Temporal Database Management*, 1999.

[PSZ99] Christine Parent, Stefano Spaccapietra, and Esteban Zimanyi. Spatio-Temporal Conceptual Models: Data Structures + Space + Time, 1999.

[Rob81] J. T. Robinson. The kdb-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Conf.*, pages 10–18, 1981.

[Ruc77] R. Rucker. *Geometry, Relativity and the Fourth Dimension.* Dover Books, 1977.

[SJ91] Richard T. Snodgrass and L. Edwin McKenzie Jr. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, 1991.

[SJLL99] Simonas Saltenis, Christian Jensen, Scott Leutenegger, and Mario Lopez. Indexing the Positions of Continuously Moving Objects. Technical report, Chorochronos, 1999.

[SN97] Andreas Steiner and Moira C. Norrie. Implementing Temporal Databases in Object Oriented Systems. Technical report, TimeCenter, 1997.

[SO95] Richard T. Snodgrass and Gultekin Ozsoyoglu. Temporal and Real-time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), August 1995.

[SRF87] T. Sellis, N. Rossopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th Intl. VLDB Conference*, pages 507–518, 1987.

[ST97] Betty Salzberg and Vassilis Tsotras. A Comparison of Access Methods for Temporal Data. Technical report, TimeCenter, 1997.

[SVZ98] Richard Southern, Aisling Vasey, and Daniel Ziskind. Comparison of Relational and Object Oriented Database Paradigms with respect to Geographic Information Systems, 1998.

[TML99] T. Tzouramanis, Y. Manolopoulos, and N. Lorentzos. Overlapping B^+-trees: an Implementation of a Transaction Time Access Method. Technical report, Chorochronos, 1999.

[TSPM98] Yannis Theodoris, Timos Sellis, Apostolos Papadopoulos, and Yannis Manopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *10th International Conference on Scientific and Statistical Database Management, Proceedings.* IEEE Computer Society, July 1998.

[TVM98] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping Linear Quadtrees: a Spatiotemporal Access Method. In *Proceedings of the 6th ACM International Workshop on Geographical Information Systems*, pages 1–7, 1998.

[VO97] Erik Voges and James Owen. A Generic Indexing Mechanism for Persistent Java, 1997.

[Wac98] Monica Wachowicz. *Object Oriented Design for Temporal GIS.* Taylor and Francis, 1998.

[Wor98] Michael F. Worboys. A Generic Model for Spatio-Bitemporal Geographic Information. In Max J. Egenhofer and Reginald G. Gollegde, editors, *Spatial and Temporal Reasoning in Geographic Information Systems*, chapter 2. Oxford University Press, 1998.

[ZKM00] Evangelos Zirintsis, Graham Kirby, and Ron Morrison. Hyper-Code Revisited: Unifying Program Source, Executable and Data. In *Proc. 9th Intl. Workshop on Persistent Object Systems*, pages 202–217, 2000.