Implementation of Spatio-Temporal Data Types with Java Generics

KREŠIMIR KRIŽANOVIĆ, ZDRAVKO GALIĆ, MIRTA BARANOVIĆ
University of Zagreb
Faculty of Electrical Engineering and Computing
Department of Applied Computing
Unska 3, 10000 Zagreb
CROATIA
{kresimir.krizanovic, zdravko.galic, mirta.baranovic}@fer.hr

Abstract: Spatio-temporal database is a new type of database that manages spatio-temporal objects and supports corresponding query languages. If position and/or extent of an objects changes continuously, we talk about moving object database. A moving objects database supports a number of data types that could be termed moving. Standard and spatial operations on non-moving data types can be applied to moving data types through a process called temporal lifting. In this paper, an implementation of spatio-temporal data types based on Java extended with parametric polymorphism is explored. A framework for implementing moving data types is presented, with emphasis on the concept of temporal lifting and implementation of lifted operations using Java genericity capabilities.

Key–Words: Spatio-temporal databases, spatio-temporal data types, moving objects, Java generics

1 Introduction

For modelling and querying moving objects, there exists a comprehensive framework of abstract data types to describe objects moving freely in 2D plane, providing data types such as moving point and moving polygon. This new database technology should allow one to model in a database the movements of entities and to pose queries about such movements. In some cases only time-dependent locations need to be managed, leading to the moving point abstraction, while in other cases time dependent shape or extent is also of interest and we speak of moving polygons.

Probably the greatest contribution to the area of moving objects databases is the work done by Güting and colleagues presented in numerous papers [2], [1], [5], [7], that resulted in creation of a SECONDO prototype DBMS. SECONDO supports a complete framework of abstract data types and operations for moving objects and can be used as a foundation for representing and querying spatio-temporal data. They set out a goal to support moving point and moving region data types, but through introducing a large number of relevant operations, there appeared a need for additional moving data types such as moving real to represent a distance between two moving points. There are many well known operations available on standard non-moving data types, that should also be applicable to moving data types. For example, a spatial operation which determines whether a point resides inside a polygon is well known. Such an operation should also be available to determine whether a moving point resides inside a moving polygon (producing a moving boolean as a result). Through temporal lifting, operations defined on simple, non-moving data types can be applied to corresponding moving data types. [8] mentions temporal sweeping algorithm as a way to implement lifting in a generic manner.

This paper explores an implementation of spatio-temporal (moving) data types using Java genericity capabilities. A framework of classes and interfaces that can be used as a foundation for representing and querying spatio-temporal data is presented here. Within this framework an implementation of temporal lifting is proposed that enables the user to implement lifted version of an arbitrary unary or binary operation.

2 Related Research

In CHOROCHRONOS project [7] the spatio-temporal database perspective was explored, with the goal of managing time dependent geometries within the database. Earlier work in that area generally focused on geometries changing in discrete steps. This restriction was dropped and continuously moving objects (points, lines and polygons) were considered, which resulted in a data model capturing a complete history of movement.

Research done by Güting and colleagues presented in [5], [2] is a landmark contribution to spatio-temporal database community. They developed an in depth formal theory for moving objects provid-
ing a foundation for future work on spatio-temporal databases. In [3], the importance of both abstract and discrete data models is discussed. [6] summarizes the work on SECONDO prototype DBMS, discussing both concepts and implementation of moving objects data types in a database. Spatial objects can change position and/or extent over time. Point and region are considered the most relevant abstractions, leading to moving point and moving region data types. However, by introducing numerous operations on those two moving data types and standard database supported data types, there appeared a need for additional data types as results of defined operations (e.g. moving real as a distance between two moving points, or moving boolean representing whether a moving point resides within a moving region). All operations defined on non-moving data types can also be applied to moving data types through a concept of lifting making a result of that operation also a moving data type.

An alternative approach by Wolfson and colleagues [12] is complementary to the previous approach, with the focus on capturing the current movements of spatial objects, and their anticipated locations in the near future. Moving objects database are considered from the trajectory location management perspective, i.e., management the locations of a set of mobile objects that are moving around right now - they introduce a concept that focuses on capturing the current motion of moving objects and their anticipated locations in the near future. A data model called MOST\(^1\) supports current and expected future movement through the dynamic attributes. Dynamic attribute implicitly changes its value with time, without explicit updates. Associated with MOST is a FTL query language, based on future temporal logic; it is basically SQL augmented with temporal operators (until, nexttime, eventually, always, etc.). This approach is restricted on moving point objects, and does not address any complex geometries.

HERMES-MDC [11] integrates two data cartridges: TAU temporal data cartridge with temporal literal types, and commercial Oracle Spatial, which provides static spatial data types. Additionally, Oracle nested tables are extensively used for the physical representation of the moving object data types, and Oracle PL/SQL for implementation of operations on them. HERMES-MDC actually extends Oracle DBMS with spatio-temporal functionality.

Only a few papers address the problem of implementation. Querying moving objects in [1], [2] and [6] relies on C++ implementation of abstract data types as an extension package for SECONDO extensible DBMS. The objective of DOMINO project [14] was to build an envelope on top of object-relational Informix DBMS in order to support moving objects. Moving objects and their plans of motion are stored in the database. DOMINO adds temporal capabilities and capabilities for managing uncertainty that is inherent in expected future motion.

### 3 Data Types

There are three kinds of different temporal predicates in moving object queries. Accordingly, the queries can be classified to three classes: historical, current and future query [9]. A system of data types supporting historical queries defined in [5] is given below. The starting point was moving point and moving region, however by introducing various operations, there appeared a need for additional data types. For example, to model a distance between two moving points, a time variant float type named moving real was needed. Also a time variant boolean (moving boolean)value can be used to represent whether a moving point resides within a moving region.

<table>
<thead>
<tr>
<th>Table 1: Data types for abstract model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow \text{BASE} )</td>
</tr>
<tr>
<td>( \rightarrow \text{SPATIAL} )</td>
</tr>
<tr>
<td>( \rightarrow \text{TIME} )</td>
</tr>
<tr>
<td>( \text{BASE} \cup \text{TIME} )</td>
</tr>
<tr>
<td>( \text{BASE} \cup \text{SPATIAL} )</td>
</tr>
</tbody>
</table>

Data types are defined using many sorted algebra. BASE, SPATIAL and TIME sorts represent simple data types, while RANGE and TEMPORAL sorts represent type constructors. When applied to a compatible simple data type, constructors produce a new data type e.g. range(int) or moving(line). Abstract semantics of data type \( a \) is defined by its carrier set denoted by \( A_a \). Each carrier set contains undefined value denoted by \( \perp \). Abstract semantics for BASE, SPATIAL, TIME, RANGE and TEMPORAL is defined in [5] in detail. BASE and SPATIAL data types are straightforward. Type instant represents a point in time and is isomorphic to real numbers. Constructor range constructor can be applied to BASE and TIME data types, has a starting and ending value and contains all values in between. TEMPORAL constructors can be applied to BASE and SPATIAL data types; intime constructor applied to type \( a \) consists of \( a \) and an instant. Constructor moving represents a mapping from instant to \( a \).
The concepts of temporal types require additional consideration, and will be described briefly. To model values of a spatial type $\sigma$ that change over time, we introduce the notion of temporal function which is an element of type

$$\tau(\sigma) = \text{time} \rightarrow \sigma$$

Temporal functions are the basis of an algebraic model for spatio-temporal data types, where $\sigma$ is assigned one of the spatial data types point, multi-Point, line, or polygon, resulting in $\text{movingPoint}$ as values of type $\tau(\text{point})$, $\text{movingMultiPoint}$ as values of type $\tau(\text{multiPoint})$, $\text{movingLine}$ as values of type $\tau(\text{line})$, and $\text{movingPolygon}$ as values of type $\tau(\text{polygon})$. Consequently, we define moving base types: $\text{movingInteger}$, $\text{movingBoolean}$, $\text{movingFloat}$, and $\text{movingInstant}$ which are relevant in spatio-temporal context.

Temporal lifting is the key concept for achieving consistency of operations on nontemporal and temporal types. The idea is to allow argument sorts and target sort of signature (the arguments and result of the operation) to be of temporal type. Formally, for each operation with signature

$$o : s_1 \times \ldots \times s_n \rightarrow s$$

its corresponding temporally lifted version is defined by:

$$\uparrow o : \tau(s_1) \times \ldots \times \tau(s_n) \rightarrow \tau(s)$$

Consider the binary topological predicate $\text{inside}$ for points and polygons, with signature:

$$\text{inside} : \text{point} \times \text{polygon} \rightarrow \text{boolean}$$

The corresponding temporally lifted topological predicate is:

$$\text{inside} : \text{movingPoint} \times \text{movingPolygon} \rightarrow \text{movingBoolean}$$

The predicate yields $\text{true}$ for each time instant at which the moving point is inside moving polygon, $\text{undefined}$ whenever the point or polygon is undefined, and $\text{false}$ otherwise.

Consistency of operation on moving and non-moving data types is achieved by lifting - all operations are first defined on non-moving data types and afterwards are systematically extended to corresponding moving data types changing the resulting data type into a moving data type as well. It is important to note that while lifting provides signatures of new operations, it does not, in itself, provide implementation for those lifted operations.

The system of data types given above is the result of abstract modelling. This is expanded upon in [2], [6] for more straightforward implementation of the moving type constructor. The discrete data type system is given below.

<table>
<thead>
<tr>
<th>$\rightarrow$ BASE</th>
<th>$\rightarrow$ SPATIAL</th>
<th>$\rightarrow$ TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, real, string, boolean</td>
<td>point, points, line, region</td>
<td>instante</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\rightarrow$ TIME</th>
<th>$\rightarrow$ RANGE</th>
<th>$\rightarrow$ SPATIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>instante</td>
<td>range</td>
<td>$\rightarrow$ UNIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\rightarrow$ UNIT</th>
<th>$\rightarrow$ TEMPORAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>urenal, upoint, upoints, uline, uregion</td>
<td>mapping</td>
</tr>
</tbody>
</table>

Representation of temporal types is based on *sliced representation*, which decomposes the temporal development of a value of some temporal type along the time dimension into collection of *temporal units*. Temporal development within each temporal unit is described by a simple function. A temporal unit records the evolution of a value of some temporal type in a given time interval, while ensuring the maintenance of type-specific constraints during such an evolution. Time intervals of a set of temporal units are mutually disjoint, and if they are adjacent, their values are distinct.

Slices are represented by generic $\text{UNIT}$ type constructor, each slice consisting of an interval and a simple function defined within that interval describing the moving value. Data types that change only discretely all use the same simple constant function and are all represented by a const type constructor, while data types that change continuously each have their own unit type constructor ($\text{ural}$, $\text{upoint}$, $\text{upoints}$, $\text{uline}$, $\text{uregion}$). A mapping data structure assembles a set of units and makes sure that their intervals are disjoint.

## 4 Implementation

An approach in [5] starts with moving point and moving region data types, and through introducing a large number of operations ends with a sizeable number of additional data types that could be termed *moving*.

Here we propose a generic Java framework [10] that can be used to implement arbitrary time variant (moving) data. Due to the space limitation we consider only a subset of a large set of operations: predi-
cate and set operation classes. First, we define spatio-
temporal type system as union of base types, spatial
types and temporal types:

\[ ST = B \cup S \cup T \]

B consists of the subset of Java core classes and
Instant time type:

\[ B = \{\text{Integer, Boolean, Float, Instant}\} \]

Type Instant is based on Java core Calendar class. It
represents a point in time; time is considered linear
and discrete, i.e. isomorphic to integers.

S is the set of spatial types:

\[ S = \{\text{Point, MultiPoint, Line, Polygon}\} \]

Spatial types are designed and implemented as Java
classes, according to [4] and [13].

T is the set of temporal types:

\[ T = \{m\text{Integer}, m\text{Boolean}, m\text{Float}, m\text{Point},
m\text{Points}, m\text{Line}, m\text{Polygon}\} \]

To implement a unit object with a non-constant
function, we must first implement a function for a cor-
responding object data type. All functions must im-
plement IFunctor<T> interface. A part of imple-
mentation of a linear function for unit points is given
below.

```java
public class LPointFunction implements IFunctor<Point>{
    private Point initial;
    private float stepx, stepy;
    public LPointFunction(Point initial, float stepx, float stepy);
    public LPointFunction(Point point1, TimeInstant t1,
                           Point point2, TimeInstant t2);
    public Point getVal(TimeInstant Itime){
        float x, y;
        x = initial.x + stepx*Itime.time;
        y = initial.y + stepy*Itime.time;
        return new Point(x, y);
    }
}
```

A new unit point can be given linear function by creat-
ing a new object of the appropriate function class and
assigning it to the member variable.

Moving objects are generally created empty, and
then unit objects are gradually added to a moving ob-
ject. However, MO<T> class contains a special con-
structor that takes a single object of parameter class
T. This special moving object consists of a single unit
object with a constant value on interval \((-\infty, +\infty)\)
and is used in generic implementation of lifting.

5 Implementation of Lifting

Operations applicable on stationary data types can be
applied to moving data types through a process of
temporal lifting as shown in chapter 3. Each opera-
tion that needs to be lifted, has to inherit from one of
two abstract classes: MUnaryFunction<R, T> or
MBinaryFunction<R, T1, T2> class, depending on the number of arguments.

Both classes are parametrized taking input and output argument classes as parameters. This means that operations with the same name, but different arguments will be represented by two separate classes. For example:

```java
public class DistancePP extends MBinaryFunction<Float, Point, Point>;
public class DistancePL extends MBinaryFunction<Float, Point, Line>;
```

where class DistancePP represents distance between two points and class DistancePL represents a distance between a point and a line. We believe that this is acceptable since these operations are calculated using different algorithms and require separate implementations.

Figure 2: MUnaryFunction and MBinaryFunction

To actually execute implemented operation, classes MUnaryFunction and MBinaryFunction contains several overloaded execute methods. MUnaryFunction has one method for simple objects, one for unit objects and one for moving objects. The first method represents the operation applied to stationary data type, while the third method is one that is used for corresponding moving data type.

```java
public abstract R execute(T a1);
public UO<R> execute(UO<T> a1);
public MO<R> execute(MO<T> a1);
```

Only the first method is abstract and must be implemented. For example, to implement lifted version of Area(Polygon), the user has to extend MUnaryFunction<Float, Polygon>, and implement execute method for simple data types.

Since the framework assumes constant values for unit objects by default, second execute method will simply apply the method defined above on the value of input unit, and the third execute method will apply execute on each of its units. If, for example, a linear function is used for unit objects, execute method for unit objects will also need to be overridden to support that custom unit object function.

```java
UO<Float> execute(UO<Polygon>)
```

Implementation of MBinaryFunction is somewhat more complex compared to MUnaryFunction. It has 2 additional execute methods where only one of the arguments is of moving data type, while the other is of stationary data type. Also, resulting distribution of unit time intervals will not, in general, correspond to that of either argument, but will be a combination of the two and will have to be calculated.

```java
abstract R execute(T1 a1, T2 a2);
UO<R> execute(UO<T1> a1, UO<T2> a2);
MO<R> execute(MO<T1> a1, T2 a2);
MO<R> execute(T1 a1, MO<T2> a2);
MO<R> execute(MO<T1> a1, MO<T2> a2);
```

User satisfied with constant unit values will only need to implement the first execute method, one that applies to stationary data types. If a different function for slices is required (linear for example), second execute method will need to be overridden. Only the last execute contains actual implementation, while other two methods are mapped to the last one through a special moving object constructor mentioned in the last chapter.

```java
public MO<R> execute(MO<T1> a1, T2 a2) {
    return execute(a1, new MO<T2>(a2));
};
public MO<R> execute(T1 a1, MO<T2> a2) {
    return execute(new MO<T1>(a1), a2);
};
```

Since both input arguments possibly have a different distributions of unit intervals, a refined distribution must be computed as shown in Figure 3.

Figure 3: Refined distribution
The value (function) of the resulting moving object is then calculated for each interval of the new interval distribution. This algorithm given in [8] is called *temporal sweeping*. For the resulting moving object to be defined on one of the new distribution intervals, both input moving objects have to be defined on that interval, otherwise the value of the result is *undefined*. Special care has to be taken while overriding execute method for unit objects to correctly support inexact slice interval distributions.

### 6 Conclusion and Future Work

A simple framework of generic classes for implementing moving objects and temporally lifted operations is presented in this paper. Java parametric polymorphism is exploited to provide user with a simple and effective means to implement arbitrary moving objects. A sliced representation is used to describe moving objects, and by default, each slice has a constant value. Linear and other types of interpolation can be achieved through inheritance and overriding a small number of methods. A generic implementation for lifted unary and binary operations is also proposed. Operations are implemented by extending abstract classes and overriding a simple execute method. A process of temporal sweeping is used for binary operations.

For future work we plan to implement all data types and most operations defined in [2] using framework presented here, and to work on generic implementation of a linear (and other) unit functions for suitable data types. Additionally, we plan to explore the possibility to represent operations of the same name with a single class.

**Acknowledgements:** This work was partially supported by the *Geospatial Sensors and Moving Objects Databases and Semantic Integration of Heterogeneous Data Sources* projects, funded by the Croatian Ministry of Science, Contracts 036-0361983-2020 and 036-0361983-2012.

**References:**


