An Oracle Data Cartridge for Moving Objects

Nikos Pelekis, Yannis Theodoridis

Laboratory of Information Systems

Department of Informatics

University of Piraeus

Hellas

Technical Report Series

UNIPI-ISL-TR-2007-04

December 2007

An Oracle Data Cartridge for Moving Objects

Nikos Pelekis, Yannis Theodoridis

Dept of Informatics,
University of Piraeus, Hellas
URL: http://isl.cs.unipi.gr/db
E-mail: {npelekis,ytheod}@unipi.gr

Abstract

Composition of time and space in a unified data framework results into spatio-temporal databases. Spatio-temporal Database Management Systems (STDBMS) are able to process, manage and analyze spatio-temporal data. Spatial TAU[1] *(STAU), developed in [Pel02], provides spatio-temporal functionality to Oracle10g Object-Relational Database Management System (ORDBMS). Although STAU is designed in a way that it can also be used as a pure temporal or a pure spatial system, its main functionality is to support modeling and querying of moving objects. Such a collection of data types and their corresponding operations are defined, developed and provided as a data cartridge using the extensibility interface of Oracle10g. This paper presents HERMES Moving Data Cartridge (HERMES-MDC), the core component of STAU system architecture. HERMES-MDC integrates two other data cartridges, namely the TAU temporal data cartridge and Oracle's spatial data cartridge. It introduces time-varying geometries that change their position and/or extent in space and time dimension, either discretely or continuously. HERMES-MDC extends PL/SQL, the data definition and manipulation language of Oracle10g, with spatio-temporal semantics. The resulted query language is applied to a case study related with truck motion analysis formulating a benchmarking framework for the evaluation of next generation fleet management systems.*

Keywords: spatio-temporal databases, data cartridge, Oracle

[1] TAU stands for TA Universe. TA is a particle with a lifespan of 0.000000000000001 (one million billionth) of a second, which was detected by Martin L Perl of Stanford University (Nobel of physics, 1995).

Table of Contents

1 Introduction

Spatial database research has focused on supporting the modeling and querying of geometries associated with objects in a database [Güt94]. On the other hand, temporal databases have focused on extending the knowledge kept in a database about the current state of the real world to include the past, in the two senses of "the past of the real world" (*valid time*) and "the past states of the database" (*transaction time*) [TCG+93]. About a decade efforts attempt to achieve an appropriate kind of interaction between both sub-areas of database research. Spatio-temporal databases are the outcome of the aggregation of time and space into a single framework [Wor94], [Ren97], [AR99], [Peu01], [KS+03] with an up-to-date review of spatio-temporal models proposed in the literature found in [PTK+05].

As delineated in the papers just cited, a serious weakness of existing approaches is that each of them deals with few common characteristics found across a number of specific applications. Thus the applicability of each approach to different cases, fails on spatio-temporal behaviors not anticipated by the application used for the initial model development. The aim of this paper is to describe a robust framework capable of aiding a spatio-temporal database developer in modeling, constructing and querying a database with objects that change location, shape and size, either discretely or continuously in time. Objects that change location or extent continuously are much more difficult to accommodate in a database in contrast to discretely changing objects. Supporting both types of spatio-temporal objects (the so-called *moving objects*) is exactly the challenge adopted by this paper.

In this paper, we present an integrated and comprehensive design of spatio-temporal data types in the form of an Oracle Data Cartridge. *HERMES Moving Data Cartridge* (HERMES-MDC) is the core component of the object-relational part of the *HERMES* system architecture [Pel02]. *HERMES-MDC* provides the functionality to construct a set of moving, expanding and/or shrinking geometries, as well as time-varying base types, which are just variables of simple continuous functions, time periods that obtain hypostasis when projected to the spatial domain at a specific instant associated with of time. Each one of these moving objects is supplied with a set of methods that facilitate the cartridge user to query and analyze spatio-temporal data. Embedding this functionality offered by *HERMES-MDC* in Oracle's object-relational DBMS data manipulation language [FP97], one obtains a query language for moving objects that outperforms related work, in terms of flexibility, expressiveness and ease of use.

In order to implement such a framework in the form of a data cartridge we are based on a set of basic types including standard data types such as integer, real, string and boolean together with the static spatial data types offered by the *Spatial* option of Oracle10g [Ora03b] and the temporal literal types introduced in a temporal data cartridge, called *TAU Temporal Literal Library Data Cartridge* (TAU-TLL) [Pel02]. Based on these temporal and spatial object data types and the ideas behind the *abstract data types* for moving objects that have been introduced in [GBE+00], [FGN+00] and [LFG+03], this paper discusses the design principles and the implementation issues concerning *HERMES-MDC*. The values of such moving types are functions that associate each instant in time, with an Oracle spatial type. Suitable operations are defined on these types to support querying and to make spatio-temporal data management easier and more natural and sensible to users and applications.

Following the object-oriented paradigm as it is provided via the PL/SQL object-relational environment of Oracle, in this paper we first present the data types introduced in *HERMES-MDC* in an abstract way (Section 2) and then, we will illustrate how these conceptual objects are mapped to working constructs and how they are incorporated to Oracle ORDBMS (Section 3), together with an appropriate set of operations (Section 4) that extend PL/SQL. Subsequently, we present the way that *HERMES-MDC* is integrated with the architecture of the *HERMES* system and how it can be applied to a case study related with truck motion analysis, which can be considered as a benchmarking framework for the evaluation of next generation fleet management systems (Section 5). Subsequently, the paper presents related work in the field also in comparison with *HERMES-MDC* functionality (Section 6). Finally, the paper winds up and at the same time points out some interesting future research directions (Section 7).

2 HERMES Moving Data Cartridge

The basic modeling primitives of *HERMES System* [Pel02] are objects and literals. An *object* is a computational entity with a unique object identifier that encapsulates both state and behavior. The *state* of an object is defined by the values it carries for a set of properties. These *properties* can be attributes of the object itself or relationships between the object and one or more other objects. The *behavior* of an object is defined by a set of operations that can be executed on or by the object. On the other hand, a *literal* is a computational entity that has only state. Let *V* be a universe of all possible computational entities, containing objects and literals. A type is a set of elements of *V* that obey some technical properties. Each type is associated with a predicate function defined over the *V*. A value $v \in V$ satisfies a type iff the predicate is true for that value. A value that satisfies a type is called *member* of the type. A *type system* is a collection of types.

Types in *HERMES System* are divided into *Base Types BT*, pure *Temporal Types TT*, pure *Spatial Types ST* and *Moving Types MT*, i.e., $HERMES = BT \cup TT \cup ST \cup MT$. Figure 1 illustrates, in UML notation, all types in *HERMES Type System*.

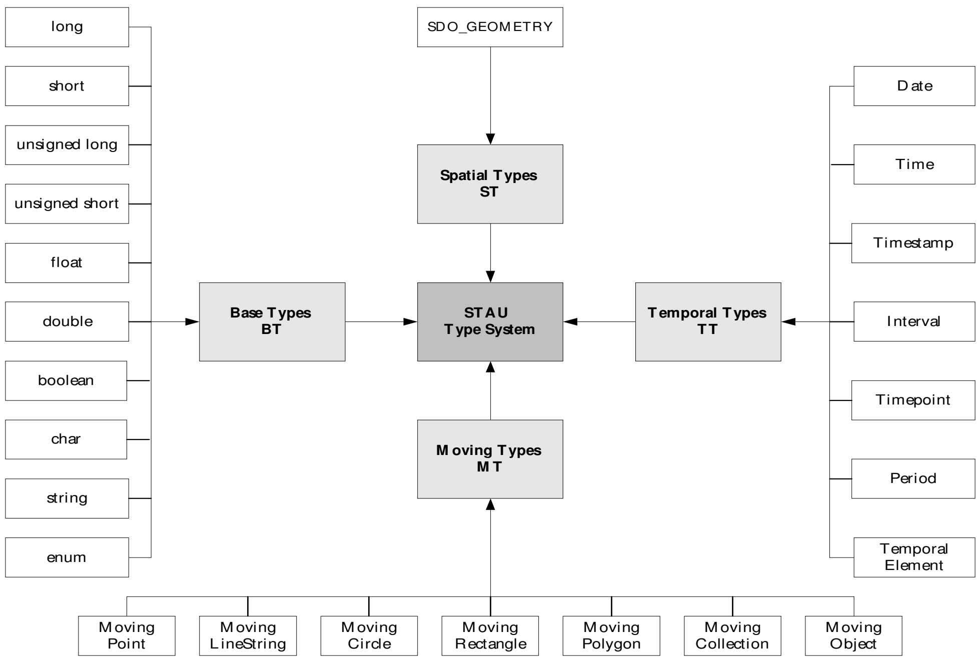


Figure 1 HERMES Type System

In the following sections, we design a type model by introducing the basic types as well as the more complex types upon base types. A fundamental requirement to this design is that types enable us to record the spatio-temporal aspects of the objects in the database. We focus on capturing spatio-temporal processes that change continuously as this is the most challenging and also allow us to capture spatio-temporal phenomena that change in discrete steps as a special case of continuous change (as will be discussed in Section 3). The data types are classified into two main categories. The first category consists of off-the-shelf or already proposed types (base, pure spatial and pure temporal types) and the second category introduces types that describe moving objects.

2.1 Base, Temporal and Spatial Types

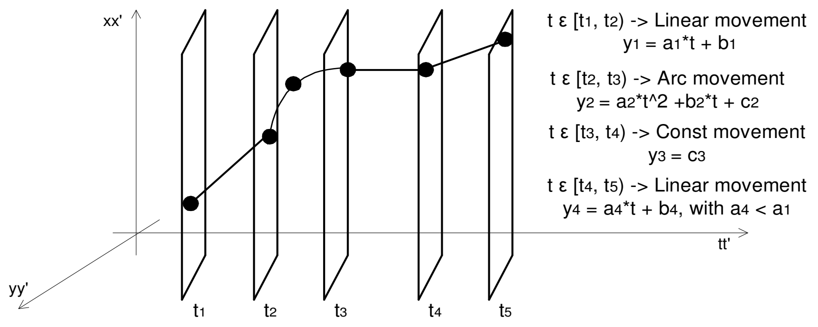
Base types are the standard database types built into most DBMS, such as integer and real (float) numbers, alphanumeric strings and booleans. These types form a subset of the Atomic Literal Types needed to define temporal types (see Appendix A).

Temporal types are introduced in [Pel02] by *TAU-TLL*, which is the component of *HERMES* system responsible for providing *HERMES-MDC* with pure temporal object-relational functionality. Basically, this cartridge implements the *Time Model*, adopted by the *TAU Temporal Object Model* [Kak96], and augments the four temporal literal data types found in *ODMG* object model [CB97] (namely, *Date*, *Time*, *Timestamp* and *Interval*) with three new temporal object data types (namely, *Timepoint*, *Period* and *Temporal Element*). TAU-TLL provides clear semantics about the time boundaries, time order, time reference, temporal granularities, and the supported calendar. More specifically, in this cartridge, time is bounded at both ends and times are stored in fixed-sized data structures. Transaction time is by definition bounded by the database creation time (on the left end) and the current timepoint (on the right end). TAU-TLL implements the widely used *Gregorian calendar* and adopts the *discrete* model of time, where time is isomorphic to the integers because of its better representation and manipulation on databases. Time axis is partitioned into a finite number of discrete segments, called *granules* [WJL91]. The choice of a partitioning scheme is termed as *granularity*. The granularity of the timestamp that a fact is associated with denotes the precision to which the timestamp can be represented. Time order refers to whether the time axis can be always considered as linear or non-linear. In the linear model, time advances from past to future in a totally ordered form. The non-linearity of the time axis deals with aspects of the time such as *periodic time* and *branching time* [TL91]. In TAU-TLL, time axis is always considered as linear and totally ordered and its current implementation supports only absolute time (support for relative time is considered as future work). In Appendix A, we formally define the temporal literal types utilized in the specification of the spatio-temporal moving types.

On the other hand, spatial types are supported by another component of the *HERMES* system architecture, called *Spatial Data Cartridge*, which provide the *HERMES* system with pure spatial object-relational functionality. This data cartridge has been developed by Oracle [Ora03b] and is an integrated set of functions and procedures that enable spatial data to be stored, accessed, and analyzed quickly and efficiently in an Oracle10g database. The geometric operations, which form the behavior of the object types supported by this data cartridge, handle queries statically, meaning that there exists no notion of time associated to the spatial objects. The spatial data model adopted by Oracle10g, which in its turn is adopted by HERMES-MDC, is a hierarchical structure consisting of elements, geometries, and layers, corresponding to representations of spatial data (for a detailed description of Oracle Spatial Data Cartridge, see Appendix B). Oracle stores the geometric description of any type of spatial object in a single row, in a single column of object type *Sdo_Geometry*.

2.2 Moving Types

As already mentioned, the authors in [GBE+00], [FGN+00] and [LFG+03] introduce the concept of *sliced representation*, the basic idea of which is to decompose the temporal development of a moving value into fragments called *"slices"* such that within the slice this development can be described by some kind of *"simple"* function. This is illustrated in Figure 2 for a time-varying point (moving point).



xx'
yy'
tt'
t_1
t_2
t_3
t_4
t_5
t ε [t_1, t_2) -> Linear movement
$y_1 = a_1*t + b_1$
t ε [t_2, t_3) -> Arc movement
$y_2 = a_2*t^2 + b_2*t + c_2$
t ε [t_3, t_4) -> Const movement
$y_3 = c_3$
t ε [t_4, t_5) -> Linear movement
$y_4 = a_4*t + b_4$, with $a_4 < a_1$

Figure 2 Moving Point with various types of movement

The *sliced representation* concept is adopted by our spatio-temporal data model and is utilized in the implementation of *HERMES-MDC*. In order to use sliced representation to define a moving type, one has to decompose the definition of each moving type into several definitions, one for each of the slices that corresponds to a simple function, and then compose these sub-definitions as a collection to define the moving type. Each one of the sub-definitions corresponds to a so-called *unit moving type*.

In order to define a unit moving type, we need to associate a period of time with the description of a simple function that models the behavior of the moving type in that specific time period. Based on this approach, two real world notions are directly mapped to our model as object types, namely *time period* and *simple function*. The first concept has been already introduced as an object type of the model by TAU-TLL (called *D_Period_Sec* in TAU-TLL terminology). The second concept is an object type, named *Unit_Function*, introduced to describe different representations of simple functions that have as domain of definition any real number that corresponds to a timepoint *inside* the previously mentioned time period, and as domain of values any real number.

$$f(t)\ :\ t \in \Re \cap [\mathrm{t}_1, \mathrm{t}_2) \longrightarrow \Re, \quad \text{where } [\mathrm{t}_1, \mathrm{t}_2) \text{ is the time period}$$

Combining these two object types together, the most primitie and simplest unit object type is defined, namely *Unit_Moving_Point*. This is a fundamental type since all the successor unit types are defined based upon it.

Following this, we define two unit moving types directly based on *Unit_Moving_Point*, namely *Unit_Moving_Circle* and *Unit_Moving_Rectangle*. As it is easily inferred, these two object types model circle and rectangle geometry constructs that change their position and/or extent over time.

For modeling the subsequent object types (*Unit_Moving_Polygon* and *Unit_Moving_LineString*) an intermediate object type that represents the simplest built-in constituent of these types is needed. This requirement is met by the *Unit_Moving_Segment* object, which models a simple line or arc segment that changes its shape and size according to its starting and ending unit moving points. This is clarified in Figure 3 where a moving segment is mapped to a line segment at two different time instants t_1 and t_2. During the time period between t_1 and t_2, the starting moving point mp_1 follows a simple linear trajectory, while the ending moving point mp_2 follows an arc trajectory.

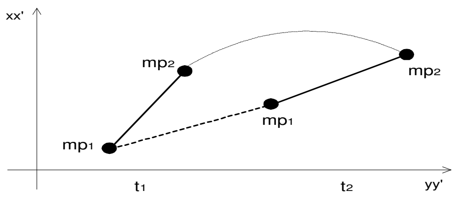


Figure 3 Linear Unit Moving Segment

Having defined the fundamental unit moving types, we now introduce the moving types that play the dominant role in our spatio-temporal data type system. The process that is followed to define the moving types is to introduce a moving type as a collection of the corresponding unit moving type, which means, in terms of object orientation, that there exists a composition relationship between the unit moving type and the moving type. As such, the *Moving_Point*, *Moving_Circle*, *Moving_Rectangle*, *Moving_LineString* and *Moving_Polygon* object types are introduced as a collection of *Unit_Moving_Point*, *Unit_Moving_Circle*, *Unit_Moving_Rectangle*, *Unit_Moving_LineString*, *Unit_Moving_Polygon* object types, respectively.

Similarly, in order to model homogeneous collections of moving types, *multi*-moving types are defined as collections of the corresponding moving types. Consequently, the proposed spatio-temporal model is augmented by the following object types: *Multi_Moving_Point*, *Multi_Moving_Circle*, *Multi_Moving_Rectangle*, *Multi_Moving_LineString* and *Multi_Moving_Polygon*.

An interesting issue here is that the previously mentioned multi-moving types do not carry their own methods interface. All the functionality for these types can be invoked by the methods of another object type, called *Moving_Collection* standing as the supertype and aggregating the interfaces, the object methods and the spatio-temporal semantics of all the multi moving types. Furthermore, the moving-collection type is able to represent heterogeneous collections of moving types, i.e., collections of different time-varying spatial geometries.

The concept of inheritance is also utilized at the level of moving types by introducing an object that encapsulates all semantics and functionality offered by moving types, including *Moving_Collection*. The so-called *Moving_Object* object type is the conjunction of all the other moving object types, which implies that this object can completely substitute any other moving type. Furthermore, the *Moving_Object* models any moving type that can be the result of an operation between moving objects. For example, the intersection of a *Moving_Point* with a polygon geometry is obviously another *Moving_Point* that is the restriction of the first *Moving_Point* inside the polygon. This result can be modelled as a *Moving_Object*. If the result of an operation is not a moving geometry then *Moving_Object* plays the role of a degenerated moving type. In other words, if there is an operation that requests the perimeter of *Moving_Polygon*, then obviously the result of this method is a time-varying real number (*Moving_Real*). Such collapsed moving types like *Moving_Real*, *Moving_String*, and *Moving_Boolean* are also modelled using the *Moving_Object* object type.

Summarizing, Figure 4 illustrates a UML class diagram [FS98] for *HERMES-MDC*. A formal definition of Moving types is found in Appendix C.

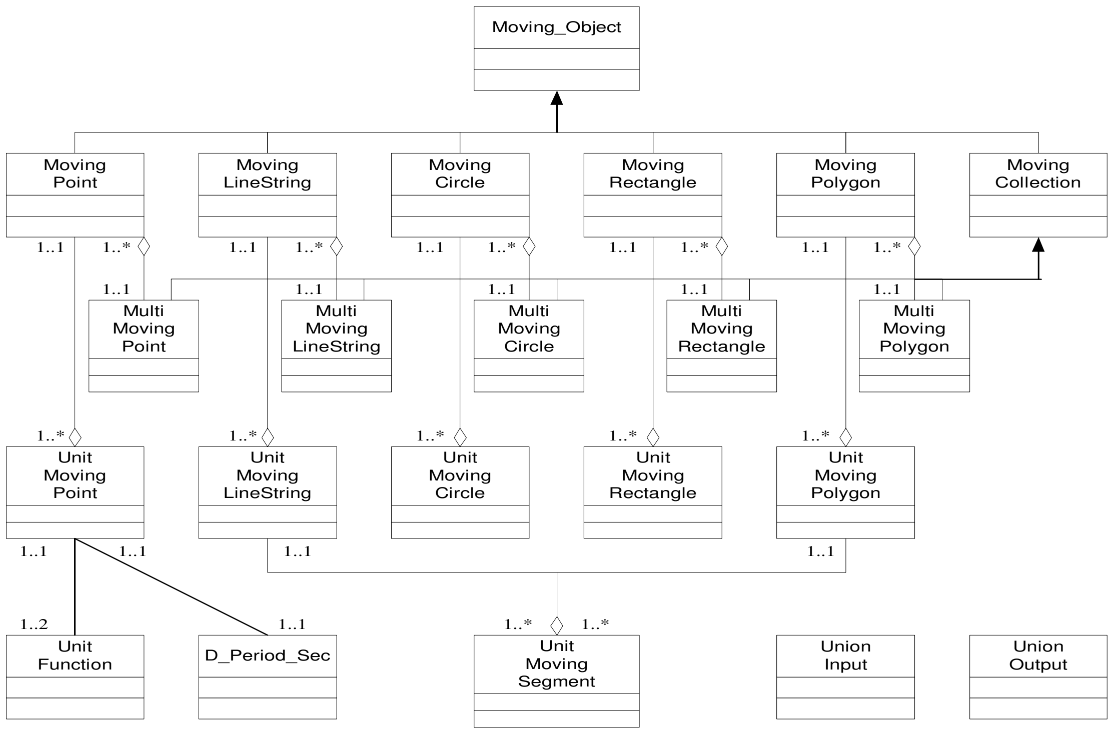


Figure 4 HERMES-MDC Class Diagram

3 Physical Representation of the Moving Object Data Types

The physical representation of the data types reflects the structures that are necessary in order to capture the semantics and implement the methods of these data types. As it has already been stated, the implementation environment, namely the PL/SQL, provides the base types needed and TAU-TLL provides the temporal types required for extending the object-relational interface of Oracle10g with spatio-temporal semantics. In this section, we discuss how moving object data types (described abstractly in Section 2) are mapped to physical structures for storing continuously and discretely time-dependant geometric data to an Oracle10g database. The following subsections propose low-level constructs for the implementation of such objects and illustrate the design decisions and implementation issues considered during development.

3.1 Moving Point

Before starting with the simplest of the moving types, we should first point out the most important technical characteristic of Oracle10g utilized in the implementation of all of the moving types. This is the *nested table* feature that enables us to model the moving types as collections of their corresponding unit moving types. The notion of data cartridges [Ora03a] is based on the idea that the database user is able to define own complex data type and manipulate and store this object in a single column, in a single row in a database table. In this case, the requirement is to store a collection of types with an unknown number of elements, so the solution is to make use of nested tables and, as such, store data out-of-line in a *store table*, which is a system-generated database table.

Having this in mind, we construct *Moving_Point* object type as a collection of *Unit_Moving_Point* objects (pointer to a nested table of *Unit_Moving_Point* objects), which in turn are defined as objects consisting of three attributes. The first attribute is the time period during which the other two attributes are defined. The time period is expressed as a *D_Period_Sec* object implemented in *TAU-TLL*, while the other two attributes are of

Unit_Function object type, whose domain of definition is the set of real numbers inside the open interval *[t_1, t_2)*, where t_1 is the starting point of the period and t_2 is the ending point of the period. There are two such attributes, one for each ordinate *x, y* in the Cartesian plane.

Unit_Function is constructed as a triplet of real numbers (*a*, *b*, *c*) representing variables of functions and a flag indicating the type of the simple function. In the current version, four types of functions are supported, namely polynomial of first and second degree, square root of polynomial of second degree and the constant function. Figure 5 illustrates the type of motion that each one of the mathematical functions implies.

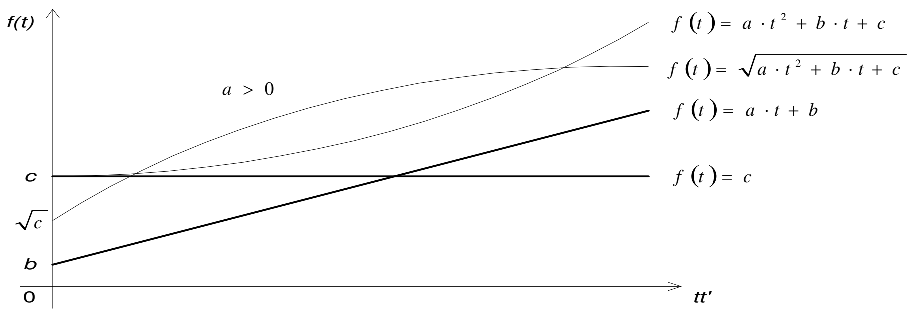


Figure 5 Graphical Illustrations of Unit Functions

The modeling of *Unit_Function* is extensible; for example, if one wishes to add interpolations with spline or polynomials with degree higher than two, then what is only needed to be done is the addition of the necessary variables as attributes of the object and the implementation of such a function.

We model a moving type that changes discretely for a period of time by setting all *Unit_Function* objects of the corresponding unit-moving type to be constant functions. Due to the fact that the ordinates represented by these *Unit_Function* objects do not change for this period of time, it is equivalent to taking a snapshot of the moving geometry, which is valid for the entire period. If at least one of these unit functions is not constant then the moving type change is continuous for this period of time. In case of a moving linestring and in order to model a discrete change for a period, the above assignment should take place for all unit-moving points that compose the corresponding unit-moving linestring. What is more, if this process were continued to all unit-moving types the result would be a completely discretely changing moving geometry.

3.2 Moving Circle and Moving Rectangle

Similarly to the *Moving_Point* object, *Moving_Circle* and *Moving_Rectangle* object types are constructed as pointers to nested tables that are collections of *Unit_Moving_Circle* and *Unit_Moving_Rectangle*, respectively. Even though these two types could be modelled as special instances of *Moving_Polygon* object, we have decided to distinguish these both for simplicity and flexibility reasons as well as for development reasons. The motivation for defining distinct object constructors for these moving types is that both of them need just a small, predefined number of unit types, in contrast to the moving polygon, where the number of its sub-elements is unknown and generally large. What is more, this important distinction facilitates the mapping of these moving types to their corresponding pure spatial geometries and makes the process of finding degenerated moving types at specific time instants easier.

Let us now examine the structure of *Unit_Moving_Circle* and *Unit_Moving_Rectangle* objects. *Unit_Moving_Circle* consists of three *Unit_Moving_Point*, representing the three points needed to define a valid circle. In the same way, *Unit_Moving_Rectangle* is composed of two *Unit_Moving_Point*, modeling the lower-left and upper-right point needed to define a valid rectangle. Figure 6 illustrates a moving circle and a moving rectangle instantiated at four different time points t_1, t_2, t_3, t_4. At time point t_2, it is obvious to see how the different interpolation functions (meaning the variables that define them) affect the position and extent of the mapped geometries, in contrast to time point t_1. At time point t_3, a degenerated moving circle and a degenerated moving rectangle are presented, meaning that the three unit moving points that compose the moving circle become co-linear and the two unit moving points that compose the moving rectangle form a line segment that is parallel to either *xx'* or *yy'* axis. At timepoint t_4, another collapsed state is depicted, where all unit-moving points become equal. *HERMES* system is responsible to deal with such degeneracies and we will discuss in Section 4.1 how it manages to cope with these matters.

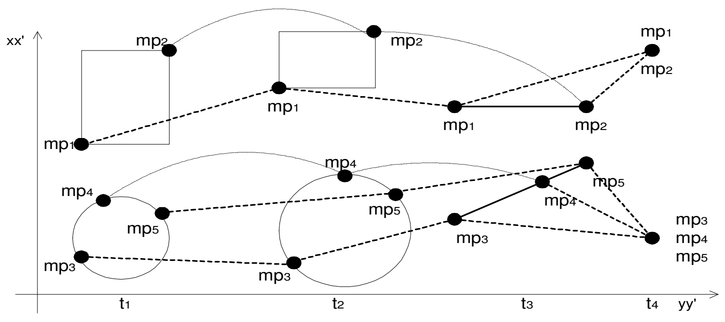


Figure 6 Instances of Moving Circle & Rectangle Objects

3.3 Moving LineString and Moving Polygon

Moving_LineString is a moving type that is also constructed as a pointer to a nested table consisting of *Unit_Moving_LineString* objects. The difference between this moving type and the previously defined is that the *Unit_Moving_LineString* is also defined as a pointer to another nested table comprising of *Unit_Moving_Segment* objects. *Unit_Moving_Segment* in its turn is formed by three *Unit_Moving_Point* objects and a flag indicating the kind of interpolation between the starting and the ending point of the *LineString* geometry. The simplest part of a *LineString* geometry can be either a linear or an arc segment. In other words, this flag exemplifies the usage of the other attributes of the *Unit_Moving_Segment* object. Figure 7 illustrates the structure of the *Moving_LineString* object.

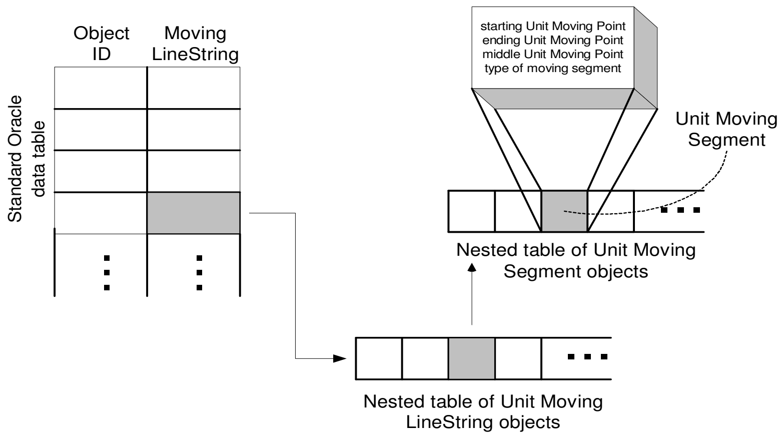

Object ID
Moving LineString
Standard Oracle data table
starting Unit Moving Point
ending Unit Moving Point
middle Unit Moving Point
type of moving segment
Unit Moving Segment
Nested table of Unit Moving Segment objects
Nested table of Unit Moving LineString objects

Figure 7 Structure of the *Moving_LineString* Object

The definition of *Moving_Polygon* is very close to the definition of *Moving_LineString*. The main difference in the two definitions is on the construction of the corresponding unit moving type. More specifically, apart from a pointer to a collection of *Unit_Moving_Segment* objects, the *Unit_Moving_Polygon* object has an additional attribute, a flag that indicates if this set of moving segments forms the exterior ring of a polygon or is an interior (hole) ring. In other words, this extra attribute adds the logic that disjoint moving holes may exist inside a moving polygon, with boundaries not crossing or touching the exterior boundary. Considering the rest aspects of the definition of *Unit_Moving_Polygon*, there is no difference between the two object types.

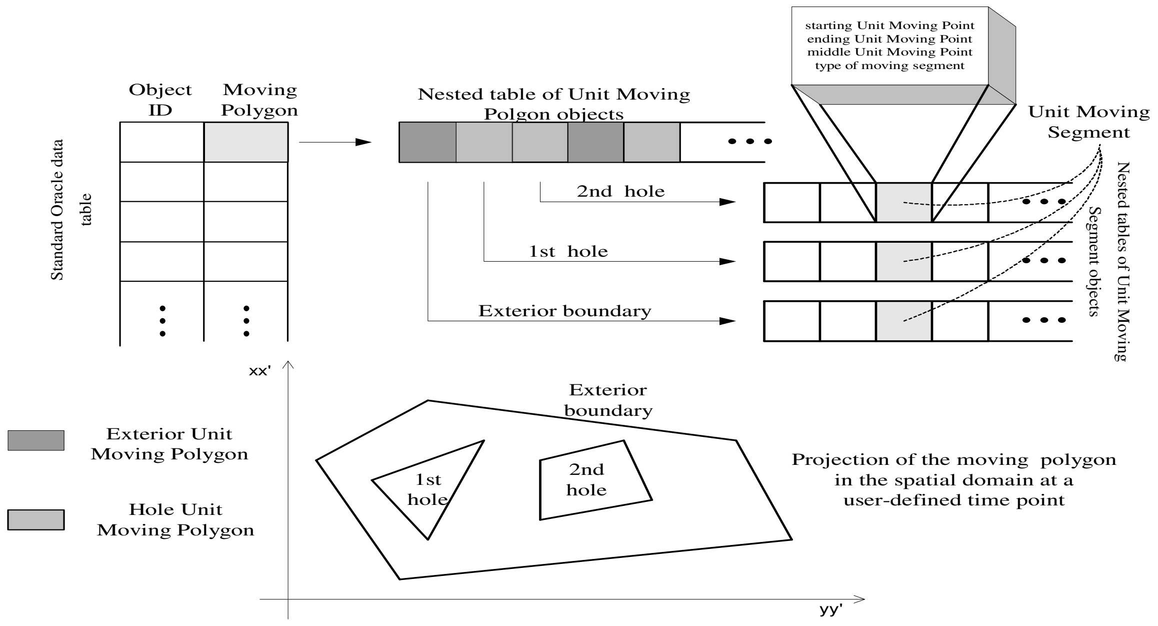

starting Unit Moving Point
ending Unit Moving Point
middle Unit Moving Point
type of moving segment
Object ID
Moving Polygon
Nested table of Unit Moving Polgon objects
Unit Moving Segment
Standard Oracle data table
2nd hole
1st hole
Exterior boundary
Nested tables of Unit Moving Segment objects
xx'
Exterior boundary
Exterior Unit Moving Polygon
Hole Unit Moving Polygon
1st hole
2nd hole
Projection of the moving polygon in the spatial domain at a user-defined time point
yy'

Figure 8 Structure of the *Moving_Polygon* Object

Actually, the differences between these two moving types come from the different utilization of their collections of moving segments by the object methods. For example, an operation that maps a *Moving_LineString* to a *LineString* geometry checks for inequality on the starting and ending points of the line and this is a prerequisite for constructing the geometry. On the contrary, the corresponding method for a moving polygon checks for the opposite, in order to be able to construct a valid polygon. Another discrepancy of *Moving_Polygon*, in contrast to all the other moving types, is that in case it includes interior moving holes, then several *Unit_Moving_Polygon* objects need to be accessed in order to transform it to its corresponding spatial geometry at a specific instant (see Figure 8).

3.4 Moving Collection

Moving_Collection is the object type that models both homogeneous and heterogeneous collections of moving types. This is accomplished by defining this object as a set of five pointers to the following types: *Multi_Moving_Point*, *Multi_Moving_LineString*, *Multi_Moving_Circle*, *Multi_Moving_Rectangle* and *Multi_Moving_Polygon*. Each one of these moving types represents a homogeneous collection of moving points, linestrings, circles, rectangles and polygons, constructed as a pointer to a nested table of *Moving_Point*, *Moving_LineString*, *Moving_Circle*, *Moving_Rectangle* and *Moving_Polygon* object types, respectively.

This object hierarchy is a design practice used in this data cartridge and is basically a technique for simulating *inheritance* in PL/SQL. To simulate inheritance, all of the subtypes for a given supertype are created as object types. The supertype is created as an object type as well. The supertype declares an attribute for each subtype. The supertype also declares the constraints to enforce the one-and-only-one rules for the subtype attributes. All of the methods that can be executed for the subtype must be defined in the supertype.

When a *HERMES Moving Cartridge* user requires constructing a homogeneous collection of moving polygons then he/she just constructs a *Multi_Moving_Polygon* and leaves the pointers to other multi moving types unreferenced. In the situation that requires a heterogeneous collection, the user simply references all the different multi moving types and the rest is taken care by the data cartridge. The design decision we made was not to provide operations for all the multi moving types, but to merge all the functionality of these individual moving types under the *Moving_Collection* object type. The methods of *Moving_Collection* treat all the multi moving types uniformly and they do not have the knowledge whether they are dealing with a homogeneous or heterogeneous collection.

3.5 Moving Object

Moving_Object is the outcome of the conjunction of all the previous presented moving objects. *Moving_Object* can be considered as the supertype of these types. It follows the same technique for simulating inheritance as in case of *Moving_Collection* and, although this type can represent the whole type-system of *HERMES-MDC*, is not intended to be directly used or constructed by a data cartridge user. On the contrary, it is intended to be the result type of operations of the other moving types (i.e., system generated). We define this generic object as follows:

```
CREATE TYPE Moving_Object AS OBJECT (
        mobject REF Moving_Object,
        mpoint REF Moving_Point,
        mline REF Moving_LineString,
        mcircle REF Moving_Circle,
        mrectangle REF Moving_Rectangle,
        mpolygon REF Moving_Polygon,
        mcollection REF Moving_Collection,
        geometry REF MDSYS.SDO_GEOMETRY,
        gtype string,
        optype string,
        arg1 integer,
        arg2 integer,
        input Union_Input );
```

Figure 9 *Moving_Object* type definition

As inferred from this structure, the pointers to the moving types presented in the preceding sections model the subtypes of the current (super) type simulating inheritance. The first reference is a recursive reference to the *Moving_Object* itself and its main usefulness is to accommodate object methods that take as parameters the same kind of moving objects (e.g. topological operations between two moving polygons). The pointer to Oracle's object type that constructs any pure spatial geometry is necessary for implementing operations of moving types that have an *Sdo_Geometry* object as an argument (e.g. a topological relationship between a moving and a static linestring geometry).

The subsequent attribute named *gtype* is the property that makes *Moving_Object* behave as if it were a simple moving type. Basically, *gtype* is a flag that identifies one of the possible moving types. When the *gtype* attribute is set, all the other attributes except the moving type that is implied by the value of the attribute are unreferenced.

The rest of the *Moving_Object* attributes are used to model the fact that the *Moving_Object* also represents any result of the moving type methods that is a *time-varying type*. By *time-varying type* we imply any moving type that has been explicitly introduced by *HERMES-MDC* as well as simple types like booleans, reals and strings that change their values on different time periods.

A formal definition of the moving object types introduced in *HERMES-MDC* is provided in Appendix C.

4 Operations on Moving Object Data Types

The design of the operations of the object types introduced by *HERMES-MDC* adheres to four principles:

- *Design operations as generic as possible.*
- *Achieve consistency between operations on pure spatial, pure temporal and spatio-temporal types.*
- *Support database maintenance and consistency.*
- *Capture the interesting phenomena.*

The first principle is crucial, as our type system is quite extended. To avoid a proliferation of operations, it is mandatory to find a unifying view of collections of types, and hence to focus on properties shared by many types.

Secondly, in order to achieve consistency of operations on pure spatial, pure temporal and moving types, we proceed in three steps. In the first step, we study operations on pure spatial types supported by the *Spatial* option of Oracle10g and we select those operations that we would like to associate with temporal semantics (e.g. the length of a linestring geometry). In a second step, we use the functionality of the pure temporal types introduced by TAU-TLL and we systematically extend the operations defined in the first step to the temporal variants of the respective types (see [Pel02]). The interesting feature of this set of operations is that they need an instant in the time line as argument (e.g. the length of a moving linestring at a specific time point). The third step takes the previous time-dependant operations as its outset and removes their time dimension thus not returning pure spatial, temporal or standard data types, but other moving types (e.g. the length of a moving linestring independently of a specific time point, which is a moving object modeling a time-varying real number).

Moreover, in any DBMS that is extended with new functionality, there is a need to form rules and to apply appropriate integrity constraints to ensure database maintenance and data consistency. As such, HERMES-MDC provides suitable methods for supporting such database monitoring procedures, which can be invoked at user request or are automatically applied as functionality of other operations.

Finally, in order to extend PL/SQL with powerful spatio-temporal query constructs, it is necessary to include operations that address the most important concepts from various domains (or branches of mathematics). Whereas simple set theory and first-order logic are certainly the most fundamental and best-understood parts of query languages, we also need to have operations based on order relationships, topology, metric spaces, etc. There is no clear recipe to achieve closure of *"interesting phenomena"*; nevertheless, that should not keep us from having concepts and operations available like distance, size of a region, relationships of boundaries, and the like.

Following, we classify the operations of the moving types introduced by HERMES-MDC into appropriate categories that enable us to describe and analyze the new query capabilities. The initial set of operations is the union of the methods supported by the simple moving types (namely, *Moving_Point*, *Moving_LineString*, *Moving_Circle*, *Moving_Rectangle*, *Moving_Polygon* and *Moving_Collection*). This conjunction of operations is equivalent to the methods provided by the generic *Moving_Object* type as it models all the previous. There are minor differences among the interfaces for the simple moving objects methods, but there is substantial divergence on their implementation. What is more, operations that share the same *signature* (e.g. equal number of parameters, same argument and result types) between *Moving_Object*, *Moving_Collection* and the rest of the moving types follow totally different algorithms. In other words, there are three levels of implementation of the moving types' methods. The first consists of the operations concerning the simple moving types. The second and the third deal with the operations of the *Moving_Collection* and *Moving_Object* types, respectively, sometimes utilizing the operations of the first level. In the sequel, we will distinguish the description of a method at the different levels of implementation.

The identifiable categories of operations that HERMES-MDC supports are:

- *Checking operations*: Operations responsible for keeping the database in a consistent state.
- *Predicates*: These are operations that return boolean values concerning topological and other relationships between moving types.
- *Projection and interaction to temporal and spatial domain*: Operations that restrict and project moving types to temporal and spatial domain.
- *Numeric operations*: These are functions that compute some numeric value (e.g., the perimeter of a moving polygon).
- *Distance and direction operations*: These methods facilitate the computation of, e.g., the minimum distance between moving types or the angle formed between moving points, at user-defined instants.
- *Set operations*: These include basic set operations, such as the set union and set intersection.
- *Rate of change operations*: Operations describing the rate of change (derivative) of motion properties of the time-dependent object types.

The following sections describe the functionality with which Oracle's PL/SQL is enhanced by the use of the methods of the moving object types. The presentation of the operations hides the technical details that would disorient us from expressing the power of the resulted query language. As such, the signatures of the operations are altered to a simplified form, while we abstractly describe the algorithms for only a small motivating set of operations. The interested reader may find signatures, more algorithms and special behavior of the operations in Appendix D.

4.1 Maintaining the Database Consistent

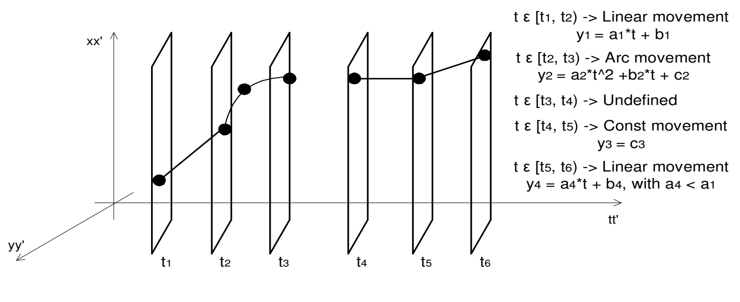
HERMES-MDC provides a set of object methods that enable the user to check the construction data of moving objects and maintain the database in a consistent state. These operations impose some integrity constraints that need to be followed for time-varying spatial data and, as such, protect the user from errors that have to do with the complex internal structure of the moving types. There are six such object methods, which we illustrate below:

- *boolean* ***check_periods_equality*** *(): Check_periods_equality* checks if the periods of the *Unit_Moving_Point* objects of each one of the unit moving types that form a moving geometry are equal. In other words, we do not permit the existence of a moving type that consists of several unit moving types and at least one of them describes the motion of its component *Unit_Moving_Point* objects with different *D_Periods_Sec* objects. Of course, such a method does not have any meaning for *Moving_Point*, as each of its unit moving types consists of only one *Unit_Moving_Point* object.

- *boolean* ***check_sorting*** *(): Check_sorting* does not force any constraint per unit moving level. On the contrary, the rule it entails, is that there should be an ascending sorting of the periods between the unit moving types, each one represented by such a period. Such a constraint is required to model the evolution of the moving types in the time line. The evolution of an object is represented by its consecutive unit moving types and the corresponding time periods should follow the same development.

- *boolean* ***check_disjoint*** *(): Check_disjoint* assures that the *D_Periods_Sec* objects that represent the time period for which the unit moving types are defined, are disjoint and that they do not intersect in any point in the time axis. More specifically, this operation checks if a period *"overlaps"* with the next in sorting-order period, namely the period of the next unit moving type.

- *boolean* ***check_meet*** *(): Check_meet* is an operation that can be invoked only by a user and is not utilized internally by the data cartridge. It checks if a period *"meets"* with the next period in the unit-type-order. This object method has as a precondition the three previous operations, meaning that except the *"meet"* criterion that should stand between periods of sequential unit moving types, all the previous operations should return true. The meaning of this operation is to assure that there is a smooth transformation of the time-changing geometries between sequential unit moving types and there are not *temporal gaps* between them. Figure 2 is an example where the *"meet"* constraint is satisfied in the transition of a moving point, as well as the *"sort"* and *"disjoint"* constraints, while in Figure 10 the *"gap"* in the motion of the moving point is caused because the second period [t_2, t_3) does not *"meet"* the third one [t_4, t_5).



t ε [t_1, t_2) -> Linear movement
$y_1 = a_1*t + b_1$
t ε [t_2, t_3) -> Arc movement
$y_2 = a_2*t\text{^}2 + b_2*t + c_2$
t ε [t_3, t_4) -> Undefined
t ε [t_4, t_5) -> Const movement
$y_3 = c_3$
t ε [t_5, t_6) -> Linear movement
$y_4 = a_4*t + b_4$, with $a_4 < a_1$
xx'
yy'
tt'
t_1
t_2
t_3
t_4
t_5
t_6

Figure 10 The Optional "meet" Constraint Applied in a Moving Point

- *boolean* ***check_degeneracies*** *(D_Timepoint_Sec): Check_degeneracies* is a method that checks if the geometry associated with a moving type at a specific point in the continuous time axis is a non-degenerated geometry. More specifically, this method finds the unit moving type (if there is one), whose period attribute (*D_Period_Sec* object) *"contains"* the time point (*D_Timepoint_Sec* object) passed as argument to the method. Afterwards, it interpolates the internal unit functions for that instant of time, imposing some rules and constraints upon the produced points in the Cartesian system of coordinates.

Depending on the type, *Check_degeneracies* imposes different restrictions on the development of these moving objects at user-defined time points. For *Moving_Point* there is not such an operation as there is no combination of mapped coordinates that could form an invalid geometry. For the rest of the *simple* moving types, the reader can find below some characteristic constraints enforced by HERMES-MDC:

- *Moving_LineString*: (a) Checks if the *Unit_Moving_Point* objects (two for line segments; three for arc segments) that define the *Unit_Moving_Segment* objects become equal at a specific time point, thus degenerating a segment to a point; (b) Checks for overlapping between consequent *Unit_Moving_Segment* objects, meaning that the two time-varying ordinates of a *Unit_Moving_Point "fall"* upon the segment that is defined by the two previous *Unit_Moving_Point* objects; (c) Checks the coordinates of the starting *Unit_Moving_Point* of the first *Unit_Moving_Segment not* to be equal at an instant, with the coordinates of the ending *Unit_Moving_Point* of the last *Unit_Moving_Segment*. In such a situation, the potential LineString is degenerated to a *Polygon* geometry, regardless the fact that this polygon may have other anomalies (e.g. self-intersected segments that are acceptable in a *LineString* geometry); (d) In case of *"arc" Unit_Moving_Segment* the method checks for co-linearity at a specific time point between the three *Unit_Moving_Point* objects that form the *"arc"* moving segment. In this situation the arc segment becomes a degenerated linear segment.

- *Moving_Circle*: (a) Checks if the three *Unit_Moving_Point* objects that define a *Unit_Moving_Circle* object become equal at a specific time point, thus degenerating a circle to a point; (b) Assures that the three *Unit_Moving_Point* objects do not become co-linear.

- *Moving_Rectangle*: (a) Checks if the lower left and upper right *Unit_Moving_Point* objects that define a *Unit_Moving_Rectangle* object become equal at a specific time point, thus degenerating a rectangle to a point; (b) Checks if the *X* or the *Y* ordinates of the projected lower left and upper right *Unit_Moving_Point* objects become equal, meaning that the produced rectangle is collapsed to a linear segment parallel to *xx'* or *yy'* axis, respectively.

- *Moving_Polygon*: (a) Checks for the same rules and constraints as in the case of *Moving_Linestring*, with the difference that, instead of inequality, it imposes equality between the starting and ending *Unit_Moving_Point*; (b) Checks if the *Unit_Moving_Polygon* objects that represent holes of a *Moving_Polygon* are always *"disjoint"* and *"inside"* the exterior boundary.

- *Varchar2* ***validate_geometry*** *(D_Timepoint_Sec, err_msg): Validate_geometry* is a generic method that performs a consistency check for valid moving geometry types. More specifically, this operation utilizes all the previous *"check"* methods by executing them in the order that we presented them, by this way producing a *control pattern* for each moving type. After applying this control pattern, the *validate_geometry* method invokes the *"at_instant"* operation, which maps a moving type to an *Sdo_Geometry* at a specific time point. Subsequently, this pure spatial object is examined under some principles that stand for the geometry model of Oracle10g. For example, polygons should have at least four points, which includes the point that closes the polygon, linestrings should have at least two points and in a multi-polygon, all polygons must be disjoint. Finally, the *validate_geometry* method following these tests returns *'TRUE'* if the moving type is valid, an Oracle error message number based on the specific reason the time-varying geometry is invalid or *'FALSE'* if the moving type fails for some other reason.

In the previous paragraphs, we described the operations concerning the constraints that should hold in a database of *"simple"* moving objects. The corresponding methods of a homogeneous or heterogeneous collection of such *"simple"* moving types, represented by the *Moving_Collection* object follow a different strategy. In other words, these operations traverse one by one all the component objects of the multi moving types that compose a *Moving_Collection* object, and apply the previous discussed operations to them. The first moving type that causes an error or is detected to be invalid or degenerated stops this process and informs the cartridge user with an appropriate message.

In the case of *Moving_Object*, these methods function differently according to the kind of *Moving_Object*. If a *Moving_Object* is just a wrapper of a simple moving type or a homogeneous or heterogeneous collection of them, then these operations just invoke the corresponding method of the wrapped moving type and return the result. If *Moving_Object* represents a time-varying object as the result of an operation between moving types (including *Moving_Collection*), or between a moving type and a static geometry, then HERMES-MDC applies the corresponding method to the moving types that participate on the construction of the *Moving_Object* and combines the separate outcomes to form the concluding result.

4.2 Predicates Modeling Topological and Distance Relationships

HERMES-MDC provides object methods in the form of predicates to describe relationships between moving types. There are two sets of predicates supported by HERMES-MDC, namely *within_distance* and *relate*. Each set of predicates consists of eight operations, each of which models the relationship of the current moving type with a *Moving_Point*, a *Moving_LineString*, a *Moving_Circle*, a *Moving_Rectangle*, a *Moving_Polygon*, a *Moving_Collection*, a *Moving_Object* and a *Sdo_Geometry* object. Each operation comes with two different overloaded signatures, modeling different semantics: the first signature is time-dependent, meaning that the outcome of the operation is related to a user-defined time point, while the second is independent to the time dimension. Below, the reader can find the pair of signatures of only one of the eight operations, and more specifically, those describing relationship with a *Moving_Polygon*. The time-dependent signature of the method is the one without the brackets, while the time-independent version of the operation can be obtained by substituting the return type of the operation with the type in the brackets { } and by removing the *D_Timepoint_Sec* argument from the parameter list. This is a common notation in the remainder of the paper.

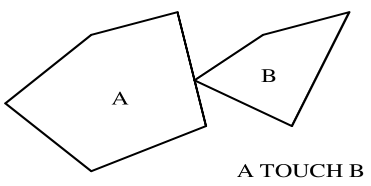
- *boolean {Moving_Object}* ***f_within_distance*** *(distance, Moving_Polygon, tolerance, D_Timepoint_Sec):* The time-dependent predicate determines whether two moving objects are within some specified Euclidean distance from each other at a user-defined time point. After mapping the moving objects to physical spatial geometries at the given instant, the function returns *TRUE* for object pairs that are within the specified distance; returns *FALSE* otherwise. The distance between two non-point objects (such as lines and polygons) is defined as the minimum distance between these two objects. Thus, the distance between two adjacent polygons is zero.

Many object methods in HERMES-MDC accept a *tolerance* parameter. If the distance between two points is less than or equal to the tolerance, the cartridge considers the two points to be a single point. Thus, tolerance is usually a reflection of how accurate or precise users perceive their spatio-temporal data to be. *Within_distance* is a characteristic example for understanding the semantics of the *tolerance* parameter. Also, the time-independent *within_distance* operation differs from the above predicate in that the return value is a *Moving_Object* that represents a time-varying boolean value. This implicitly defined *"moving boolean"* object models the sequence of the time intervals that the two related objects are within or not some specified Euclidean distance.

- *Varchar2 {Moving_Object}* ***f_relate*** *(mask, Moving_Polygon, tolerance, D_Timepoint_Sec):* This generic predicate examines two moving objects and determines their topological relationship. As previously, the *"relate"* predicate appears with two overloaded versions. The first evaluates the topological relationship upon a specific user-defined time point, while the second version returns a *Moving_Object* modeling a time-varying string (*"moving string"*), which describes the evolution in the topological relationship between the related objects. The user can specify the kind of relationships that he/she requires to check via the *mask* parameter.

  The *"relate"* operator implements a 9-intersection model for categorizing binary topological relations between moving geometries [EF91]. At any time, each object has an interior, a boundary, and an exterior. The boundary consists of points or lines that separate the interior from the exterior. The boundary of a line consists of its end-points. The boundary of a polygon is the line that describes its perimeter. The interior consists of points that are in the object but not on its boundary and the exterior consists of those points that are not in the object.

  Given that an object *A* has three components (a boundary A_b, an interior A_i, and an exterior A_e), any pair of objects has 9 possible interactions between their components. Pairs of components have an empty (0) or a non-empty (1) set intersection. The set of interactions between two projected moving geometries is represented by a 9-intersection matrix that specifies which pairs of components intersect and which do not. Figure 11 shows the 9-intersection matrix for two polygons that are adjacent to one another. This matrix yields the following bit mask, generated in row-major form: "101001111". For more details on topological relationships supported and respective values of *mask* parameter, see Appendix D.



$$\begin{array}{cc|ccc|}
 & & & \mathrm{B} & \\
 & & \mathrm{b} & \mathrm{i} & \mathrm{e} \\
 & \mathrm{b} & 1 & 0 & 1 \\
\mathrm{A} & \mathrm{i} & 0 & 0 & 1 \\
 & \mathrm{e} & 1 & 1 & 1
\end{array}$$

Figure 11 9-Intersection Matrix

4.3 Projection and Interaction to Temporal and Spatial Domain

HERMES-MDC provides object methods of special interest that have been proposed in the literature. Subsequently, we present the operations as these are defined for *Moving_Object* and the semantics behind these methods and we differentiate our presentation in case of change in the semantics of other moving types.

- *Unit_Moving_Point* ***unit_type*** *(D_Timepoint_Sec):* This operation is the single method not defined for a *Moving_Object* type. Generally speaking, this operation is defined only for the *simple* moving objects that their construction is closely related with a collection of unit moving objects. For the rest of the *simple* moving objects the above signature changes the result type to their corresponding unit moving object (see [Pel02]). The simple but very important task that this function performs is that it finds (and returns) the unit-moving object whose attribute time period (*D_Period_Sec* object) *"contains"* the user-defined time point (*D_Timepoint_Sec* object). In other words, it returns that unit-moving type where the time instant represented by the argument *D_Timepoint_Sec* object is *"inside"* the time period that characterizes the unit-moving type. What is more, the *unit_type* method carries out all the necessary checks to maintain the database consistent and to ensure the validity of the moving object.

- *Union_Output* ***at_instant*** *(D_Timepoint_Sec):* The *at_instant* operation is the most important method for the moving types introduced in HERMES-MDC, firstly because it is the operation that maps the abstract variables of mathematical functions to meaningful spatial objects conceivable by end-users and, secondly, because it is the base of implementation for many other object methods. As already mentioned, the above signature concerns the *at_instant* operation for the *Moving_Object* type. The return type (*Union_Output*) is an object that represents the union of all the possible results of the projection of a *Moving_Object* at a user-defined time point. In other words, if *Moving_Object* represents a time-varying geometry then *Union_Output* is basically an *Sdo_Geometry* object. If *Moving_Object* represents a *"moving"* real or string then *Union_Output* is a real number or a character string, respectively.

  In the case of a *Moving_Object* the *at_instant* operation invokes the *at_instant* operations of the moving types that construct the *Moving_Object*. If *Moving_Object* represents a moving geometry then the result of the previous operation is immediately returned. If *Moving_Object* represents a *"moving"* type as the result of an operation between moving objects then the projected geometries of the previous step are applied against this operation and the outcome of this second step is returned.

  In the case of *Moving_Collection*, this operation invokes the *at_instant* operations of all the moving types of the multi moving objects and subsequently applies a special *"union"* operation upon the projected geometries by *"concatenating"* them in a collection object and returns the result of the *"concatenation"*.

- *Moving_Object* ***at_period*** *(D_Period_Sec):* The *at_period* object method is an operation that restricts the moving object to the temporal domain. In other words, by using this function the user can delimit the time period that is meaningful to ask the projection of the moving object to the spatial domain. More specifically, the time period passed as argument to the method is compared with all *D_Period_Sec* objects that characterize the unit moving objects. If the parameter period does not overlap with the compared period then the corresponding unit type is omitted. If it overlaps, then the time period that defines a unit-moving object becomes its *"intersection"* with the given period.

- *D_Temp_Element_Sec* ***f_temp_element*** *():* The *f_temp_element* operation gives HERMES-MDC user the capability to project the time periods that form the unit moving objects that compose a moving type on the time line and subsequently *"concatenate"* all these distinct time periods to construct a temporal element. Figure 12 depicts the result of the *f_temp_element* operation when applied to a *Moving_Point* object.

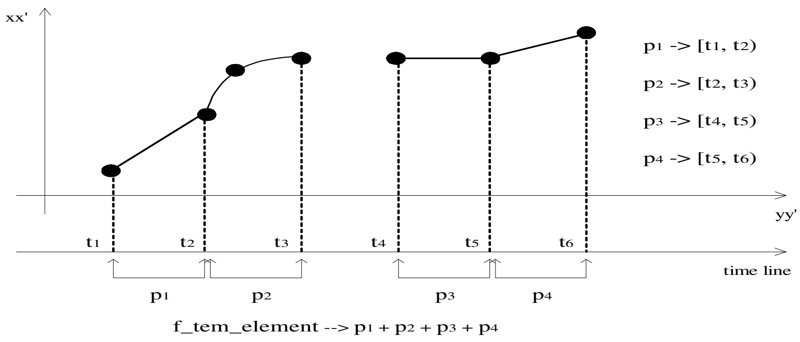


Figure 12 Projection of a Moving Point on the temporal domain

- *Moving_Object* ***at_temp_element*** *(D_Temp_Element_Sec):* Similarly to the *at_period* operation, the *at_temp_element* object method restricts the moving object to the temporal domain, but the process of restricting the periods between which the moving object is valid is driven by a collection of *D_Period_Sec* objects and not just one *D_Period_Sec* object as in the previous case.

- *Sdo_Geometry {Moving_Object}* ***f_buffer*** *(distance, tolerance, D_Timepoint_Sec):* The *f_buffer* operation comes with two overloaded versions. The first generates a buffer polygon around a moving geometry object at a specific user-defined time point, while the second version returns a *Moving_Object* modeling a time-varying polygon, which describes a moving rounded buffer around a moving geometry. Obviously, this method is meaningless for a *Moving_Object* that represents a time-varying real number or string. Calling the *f_buffer* method for such a *Moving_Object* triggers the error handling mechanism of HERMES-MDC, which informs the user with an appropriate message.

  The *f_buffer* operation for a homogeneous collection of moving geometries at a specific timepoint returns a multi-polygon where each polygon represents the buffer of its corresponding element in the collection. An interesting case is the buffer of a heterogeneous collection of moving objects, which is just one polygon that buffers all the different projected geometries together. The above-mentioned issues are visualized in Figure 13, where snapshots of different moving types and their corresponding buffer polygons are presented.

  What is not illustrated in the description of the operation is the specific structure of these buffers for each corresponding moving type. Starting with the *Moving_Point*, someone would expect that the buffer of this type at a specific instant would be a circle geometry with radius the user-specified distance of the buffer. Surprisingly, the geometry returned by *f_buffer* operation is a polygon consisting of two arc segments that circle the point at the specified distance. The same happens in the case of the *Moving_Circle* where the buffer at a specific timepoint is defined as the buffer of its centre but the distance of the buffer now is the initial user-specified distance plus the radius of the moving circle at that instant. The buffer of a *Moving_LineString*, a *Moving_Rectangle* and a *Moving_Polygon* at a specific timepoint is a compound polygon whose number of linear segments is equal to the number of linear segments that exist in the corresponding projected geometries and whose number of arc segments is equal to the number of vertices plus the number of arc segments.

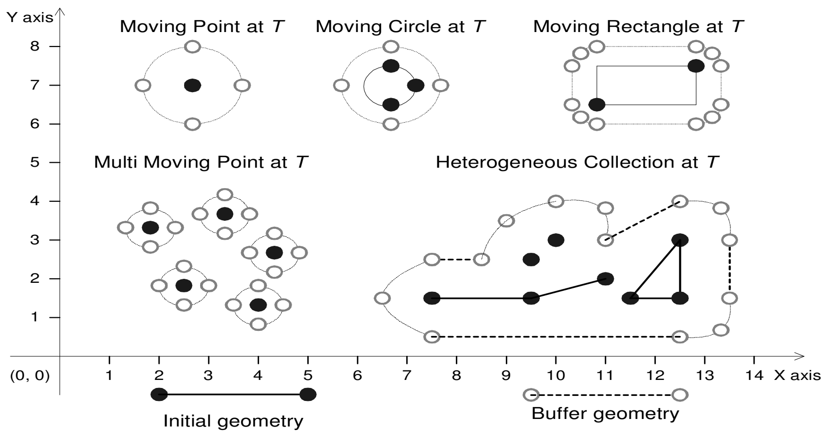


Figure 13 Demonstrating f_buffer operation

- *Sdo_Geometry {Moving_Object}* ***f_centroid*** *(tolerance, D_Timepoint_Sec):* The *f_centroid* operation returns the centre of a moving polygon object at user-defined time points. The centre is also known as the *"centre of gravity"*. The overloaded *f_centroid* function represents a moving point that at any time is the centre of gravity of the moving polygon object. The method is meaningful only for moving types that model single time-varying areas. In all other cases, (collections of moving geometries) an application error is raised informing the cartridge user.

  An interesting case presented when utilizing this operation is once the centre of gravity of the moving region falls out of its area. This could happen when the moving hole inside a moving polygon includes the centre and when a moving polygon becomes too concave at a specific timepoint. Both cases are visualized in Figure 14.

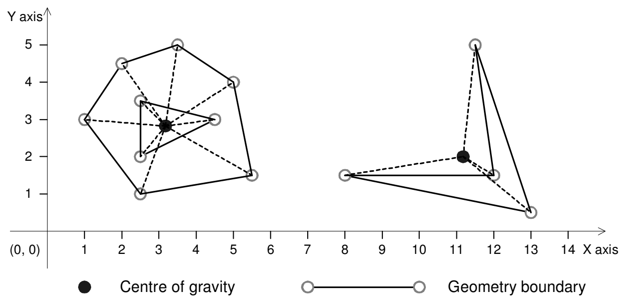


Figure 14 Demonstrating f_centroid operation

- *Sdo_Geometry {Moving_Object}* ***f_convexhull*** *(tolerance, D_Timepoint_Sec):* The *f_convexhull* method returns a simple convex polygon that completely encloses the moving geometry object at a specific instant of time. The *Moving_Object* returned by the second time-independent *f_convexhull* function, models a moving polygon that is the convex hull of a moving object at any time point. HERMES-MDC uses as few straight-line sides as possible to create the smallest polygon that completely encloses an instantiated moving object (see dashed lines in Figure 15). A convex hull is a convenient way to get an approximation of a complex geometry object.

- *Sdo_Geometry {Moving_Object}* ***f_pointonsurface*** *(tolerance, D_Timepoint_Sec):* This function returns a point geometry object representing a point that is guaranteed to be on the surface of a moving polygon when projected to the spatial domain at the time point used as argument. The returned point can be any point on the surface. The user should not make any assumption about where on the surface the returned point is, or whether the point is the same or different when the function is called multiple times with the same input parameter values. The second version of the *f_pointonsurface* operation returns a *Moving_Object*, which models a moving point whose mapping at any instant will be a point that is guaranteed to be on the surface of the corresponding projected polygon at the same time point.

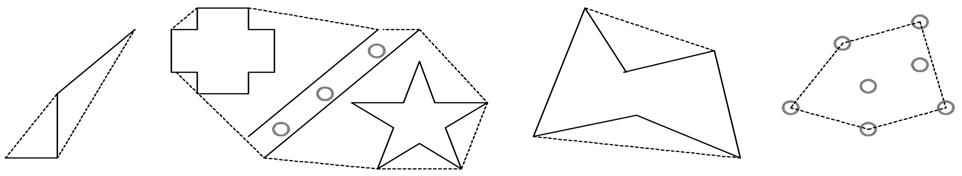


Figure 15 Convex polygons containing snapshots of several moving geometries

- *Union_Output* ***f_initial*** *():* The *f_initial* object method is basically the *at_instant* operation invoked at the first instant of time that the moving object is valid, meaning the first second of the closed-open period that identifies the least recent unit moving object.

- *Union_Output* ***f_final*** *():* Similarly to the *f_initial* object method, the *f_final* operation projects the moving object at the last valid instant of the time period that characterizes the most recent unit moving object.

- *Sdo_Geometry* ***f_traversed*** *():* The geometry returned by this function models all the places that a moving geometry *"traverses"* along its motion during the periods that characterize the unit moving objects. Such a geometry object is of polygon type. In the case of *Moving_Point* objects, the *f_traversed* method is transformed to a special operator (*f_trajectory*) described in the subsequent paragraph. Figure 16 illustrates four examples of traversed areas, one for each of the simple moving types. In the case of the traversed *Moving_LineString*, we notice that the returned geometry is not a single polygon but a multi polygon due to the fact that the periods of the unit moving objects that compose the *Moving_LineString* do not *"meet"* each other or the variables that define the unit functions between subsequent unit moving objects present a substantial difference.

- *Sdo_Geometry* ***f_trajectory*** *():* This function is the *f_traversed* method for the case of a *Moving_Point* object. In other words, this operation simulates the trajectory traversed by a *Moving_Point.* More specifically, this projection of the movement of a *Moving_Point* to the Cartesian plane is done by mapping the time-dependent ordinates of the object at the beginning, ending and a random intermediate time instant of each one of the periods that identify the *Unit_Moving_Point* objects that compose the *Moving_Point.* Subsequently, the algorithm examines whether the intermediate projected co-ordinates "fall" upon the line formed by the other two pairs of co-ordinates. Depending on the result, a linear or arc segment connecting the beginning and ending projected co-ordinates is implied. A process of merging these segments follows, to form the returned *LineString* geometry.

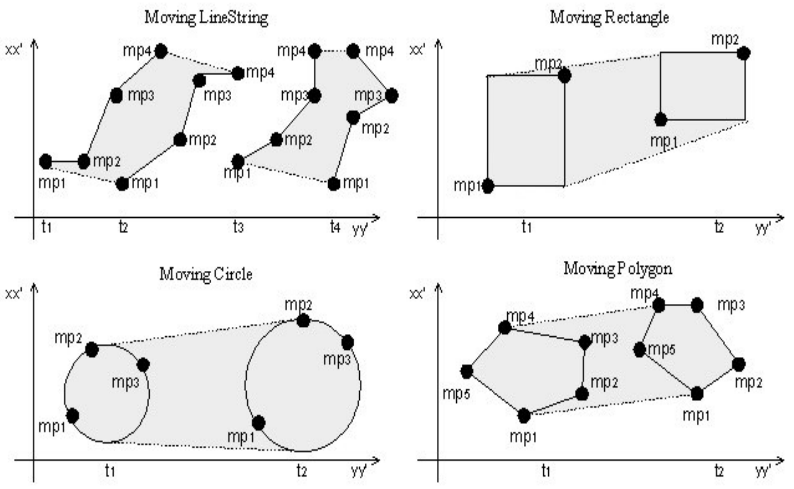


Figure 16 Areas Traversed by Moving Geometries

- *Sdo_Geometry **f_locations** ():* The *f_locations* object method is defined only for a Moving_Point object or a Moving_Object and follows the same algorithm as the *f_trajectory* operation with the difference that the returned type is a multipoint geometry representing the previously discussed projected co-ordinates at the beginning and ending timepoints of the periods that characterize the *Unit_Moving_Point* objects.

4.4 Numeric operations

HERMES-MDC supports a special category of object methods that either compute a numeric value of a moving object at a specific timepoint (e.g., the current perimeter of a moving polygon) or construct a *Moving_Object* representing the same time-varying numeric value. More analytically, we provide the subsequent numeric operations:

- *number {Moving_Object} **f_area** (tolerance, D_Timepoint_Sec):* The *f_area* operation is defined for those moving types that their projection to the Cartesian plane depicts a closed region and computes the area for this region. The second (time-independent) version of the method returns a *Moving_Object* representing the time-varying area of a moving, extending and/or shrinking region. This function works with any moving polygon, including polygons with moving holes.

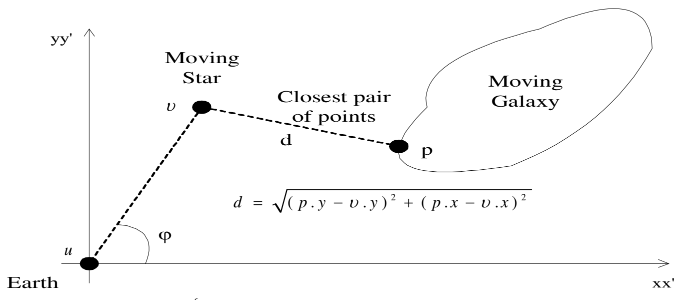
- *number {Moving_Object} **f_length** (tolerance, D_Timepoint_Sec):* The *f_length* object method computes the length of a *Moving_LineString* object or the perimeter of a *Moving_Circle*, *Moving_Rectangle* or *Moving_Polygon* projected at the Cartesian plane at a user-defined time point. For a *Moving_Polygon* that contains one or more holes, this function calculates the perimeters of the exterior boundary and all holes at the given time point, and returns the sum of all the perimeters. The second version of the method returns a Moving_Object representing the time-varying length or perimeter of the moving type that invokes the operation.

- *Varchar2 {pls_integer} **f_num_of_components** ({mtype Varchar2}):* This operation is introduced only for *Moving_Collection* objects and its functionality is to estimate and return a structured string that describes the number of components that compose the collection of moving types. The second version of this object method takes a string describing a moving geometry as parameter and returns the number of the objects of the same type that participate in the construction of the moving collection.

4.5 Distance and Direction operations

The following two methods assist the cartridge user to calculate the minimum distance between moving objects or the angle formed between moving points.

- *number {Moving_Object}* ***f_distance*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* HERMES-MDC provides a distance measure that exists for all moving types, which either computes the distance between two instantiated moving objects (the time-dependent version) or returns a time-varying real number that represents the minimum distance between these moving types at all time points (the time-independent version). The distance between two objects is the distance between the closest pair of points or segments of the two objects.



$$\phi(u,\upsilon) = \begin{cases} \perp, & if\ u = \upsilon \\ \arctan\left(\frac{\upsilon.y - u.y}{\upsilon.x - u.x}\right), & if\ \upsilon.x > u.x\ and\ \upsilon.y > u.y \\ 90, & if\ \upsilon.x = u.x\ and\ \upsilon.y = u.y \\ 180 + \arctan\left(\frac{\upsilon.y - u.y}{\upsilon.x - u.x}\right), & if\ \upsilon.x < u.x \\ 270, & if\ \upsilon.x = u.x\ and\ \upsilon.y < u.y \\ 360 + \arctan\left(\frac{\upsilon.y - u.y}{\upsilon.x - u.x}\right), & if\ \upsilon.x > u.x\ and\ \upsilon.y < u.y \end{cases}$$

Figure 17 Distance & Direction Operations

- *number {Moving_Object}* ***f_direction*** *(Moving_Point, D_Timepoint_Sec):* The *f_direction* function is defined only for *Moving_Point* objects returning the angle of the line from the first to the second moving point (measured in degrees, $0 \leq angle < 360$), after these have been projected to the Cartesian plane at a specific time point. The time-independent version of the function returns a *Moving_Object* modeling a *"moving real"*, which corresponds to the time-changing angle formed by the conceptual line segment that joins the two moving points and the *xx'* axis. Figure 17 illustrates the distance between a star (*Moving_Point*) and a galaxy (*Moving_Polygon*) projected at the spatial domain in a user-defined timepoint, as well as the angle formed by the moving star and the earth.

4.6 Set Relationships

HERMES-MDC provides four object methods for describing set-relationships between moving types. Each comes with two overloaded versions, one for describing a geometry object as the result of applying the set-relationship at a user-defined time point and one for describing a moving geometry that is defined as the set-relationship at all the time periods that this relationship is meaningful. For example the intersection of a

Moving_Point with a *Moving_Polygon* results in a *Moving_Object* that represents another moving point, which is the restriction of the initial *Moving_Point* inside or on the boundary of the *Moving_Polygon*.

Subsequently, we present the supported set-relationships operations between any moving type and a *Moving_Polygon* object. Similar operations are defined for all the other moving types, as well as operations describing set-relationships of a moving type with a pure spatial object.

- *Sdo_Geometry {Moving_Object}* ***f_intersection*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_intersection* object method returns either a geometry object that is the topological intersection (*AND* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. Invoking *f_intersection* method for the simplest moving object (*Moving_Point*), as one would expect, the result of this operation is the projection of itself on the spatial domain (point geometry) at time instants that intersects with other moving types or static geometries and *null* at time instants where it is not on the boundary or the interior of linestrings and polygons or it coincides with none of the points in a collection of them. Let us now present some motivating cases when invoking *f_intersection* method for moving linestring and polygon objects with other single or multi moving types that have more than one common points, segments or areas. Figure 18 below depicts the instantiation of a *Moving_Object* modeling the intersection of a *Moving_LineString* with a polygon, at three different timepoints t_1, t_2, and t_3. At timepoint t_1 it is obvious the result of such an operation, which is a linestring geometry. At timepoint t_2 this intersection has as result a multi-linestring geometry due to the development of *Moving_LineString*, while at timepoint t_3 the resulted geometry is a heterogeneous collection of lines and points.
- *Sdo_Geometry {Moving_Object}* ***f_union*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_union* object method returns either a geometry object that is the topological union (*OR* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation.

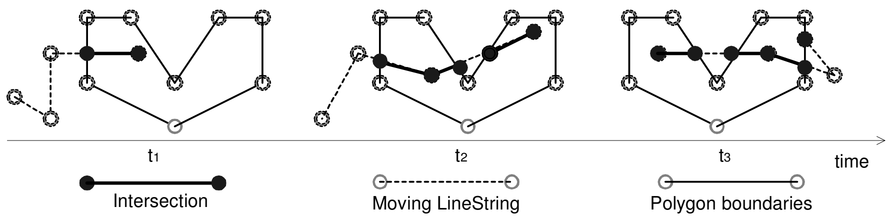


Figure 18 Demonstrating f_intersection Operation

One could extract a series of rules that stand for the outcome of the *f_union* object method, except the common one. More specifically, the union of a single moving geometry or a homogeneous moving collection with a disjoint moving (or static) geometry of the same type at a specific timepoint, results in a multi-geometry of that type. If the argument object is of different type from the caller and do not have common boundaries and/or interior areas, then the result in any case will be a heterogeneous collection. A noteworthy case is the union of a moving point or linestring with linestring or polygon geometries when at the time of the query their projection falls upon the linestring or the boundary of the polygon, respectively. In such case, the points of the moving point or linestring are interleaved as additional points in the sequence of points that defines the linestring or the boundary of the polygon.

- *Sdo_Geometry {Moving_Object}* ***f_difference*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_difference* object method returns either a geometry object that is the topological difference (*MINUS* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. Generally speaking, the *f_difference* operation returns the part of the caller object that does not belong to the argument object. More specifically, applying this method to a moving geometry at a specific timepoint, the result is the projection of this moving type if the argument object is disjoint with this projection. In a different case where the argument object completely encloses the caller's projection the result is the *null* value. For example this happens when a user requires the difference of a moving point or linestring whose instantiation falls on the boundary or the interior of a polygon or upon the segment of a linestring. An interesting case happens when the *f_difference* operation is invoked between two moving polygons at an instant where the argument polygon has been moved wholly inside the caller moving polygon. The result in such case is a polygon with a hole.
- *Sdo_Geometry {Moving_Object}* ***f_xor*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_xor* object method returns either a geometry object that is the topological symmetric difference (*XOR* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. The *f_xor* operation provides the union of the caller with the argument object, "subtracting" their intersection. As such, similarly to the *f_union* case, the *f_xor* for a moving polygon with another one that is totally inside the first returns also a polygon with a hole. If the first moving polygon does not *cover* completely the parameter moving polygon but just overlap, the result of the *f_xor* operation at a specific timepoint is a multi-polygon geometry. What is more, invoking this operation for a moving point with argument another moving point, the outcome at a specific instant is a multi-point if their projections are not the same and *null* if they are.

4.7 Rate of Change

An important property of any time-dependent value is its rate of change, i.e., its *derivative*. To determine which of our data types is applicable to this concept, consider the following definition of the derivative.

$$f'(t) = \frac{\partial f(t)}{\partial \mathrm{t}} = \lim_{\Delta t \longrightarrow 0} \frac{f(t+\Delta \mathrm{t}) - f(t)}{\Delta t}$$

This definition, and thus the notion of derivation, is applicable to the moving types that firstly support a difference operation and secondly support division by a real number. *Moving_Point* type is the single type that clearly qualifies the above prerequisites. At least three operations assume the rule of difference in the definition, namely the Euclidian distance, the direction between two points and the vector difference (viewing points as two-dimensional vectors). This leads to three different derivative operations, called *speed*, *turn* and *velocity*, respectively.

- *number {Moving_Object}* ***f_speed*** *(D_Timepoint_Sec)*: The *speed* operation comes in two overloaded signatures. The time-dependent version returns a number representing the *speed* of a moving point at a specific timepoint, while the time-independent version returns a *Moving_Object* modeling the time-varying *speed* at any time instant.

  The algorithm that implements the *speed* method is based on its formal definition:

  $$speed(t) = \sqrt{\upsilon_x^2(t) + \upsilon_y^2(t)} = \sqrt{\left(\frac{\partial s_x(t)}{\partial t}\right)^2 + \left(\frac{\partial s_y(t)}{\partial t}\right)^2}$$

  where υ_x, υ_y are the corresponding speeds of the moving point along *xx'* and *yy'* axes, which are expressed as the time derivatives of the distance functions, namely $\frac{\partial s_x(t)}{\partial t}, \frac{\partial s_y(t)}{\partial t}$. These functions are not other than the two *Unit_Function* objects needed to define a *Unit_Moving_Point*.

- *number {Moving_Object}* ***f_turn*** *(D_Timepoint_Sec)*: Similarly, *turn* operation is provided by the following two signatures, one representing the rate of change of the angle between the *xx'* axis and the motion vector at a specific timepoint and one expressing the same derivative value at any time instant.

  The above-mentioned time-varying angle $\phi(t)$ can be computed as the tangent between s_x and s_y. Utilizing the derivative of the *arctan* function $\frac{\partial \arctan(\mathrm{x})}{\partial \mathrm{x}} = \frac{1}{1+x^2}$ and the definition of derivatives of composite functions $(g(f(x)))' = g'(f(x)) \cdot f'(x)$, the derivative of $\phi(t)$ can be computed as follows:

  $$\tan(\phi(t)) = \frac{s_y(t)}{s_x(t)} \Leftrightarrow \arctan(\tan(\phi(t))) = \arctan\left(\frac{s_y(t)}{s_x(t)}\right) \Leftrightarrow \phi(t) = \arctan\left(\frac{s_y(t)}{s_x(t)}\right) \Leftrightarrow$$

  $$\phi(t)' = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \left(\frac{s_y(t)}{s_x(t)}\right)' = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \frac{(s_y(t))' \cdot s_x(t) - s_y(t) \cdot (s_x(t))'}{(s_x(t))^2} = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \frac{\upsilon_y(t) \cdot s_x(t) - s_y(t) \cdot \upsilon_x(t)}{(s_x(t))^2}$$

- *Sdo_Geometry {Moving_Object}* ***f_velocity*** *(D_Timepoint_Sec)*: Finally, the *velocity* of a moving point at a specific timepoint or at any instant during its development, is represented as a point geometry or a *Moving_Point* object, respectively.

  Viewing a Moving_Point as a two-dimensional vector $\vec{s}(t) = (s_x(t), s_y(t))$, the derivative of this vector, which implements the *velocity* operation, is given by the following equation $(\vec{s}(t))' = ((s_x(t))', (s_y(t))')$.

5 Architectural Aspects of HERMES-MDC and an Application Example

Figure 19 illustrates the architecture of the HERMES System [Pel02], as this is formed on the top of Oracle's Object-Relational DBMS. The reader can see how HERMES-MDC can be utilized in a real world scenario to assist a database developer in modeling and querying spatio-temporal data. A straightforward utilization scenario is to design and construct a spatio-temporal object-relational database schema and build an application by transacting with this database. In this case, where the applied ORDBMS is Oracle10g, in order to specify the database schema, the database designer writes scripts in the syntax of the *Data Definition Language* (DDL), which is the PL/SQL, extended with the spatio-temporal operations previously introduced.

To build an application on top of such a database for creating objects, querying data and manipulating information; the application developer writes a source program in *Java* wherein he/she can embed *PL/SQL scripts* that invoke object constructors and methods from HERMES-MDC. The *JDBC pre-processor* integrates the power of the programming language with the database functionality offered by the extended PL/SQL and together with the *ORDBMS Runtime Library* generate the application's executable. By writing independent stored procedures that take advantage of HERMES functionality and by compiling them with the *PL/SQL Compiler*, is another way to build a spatio-temporal application.

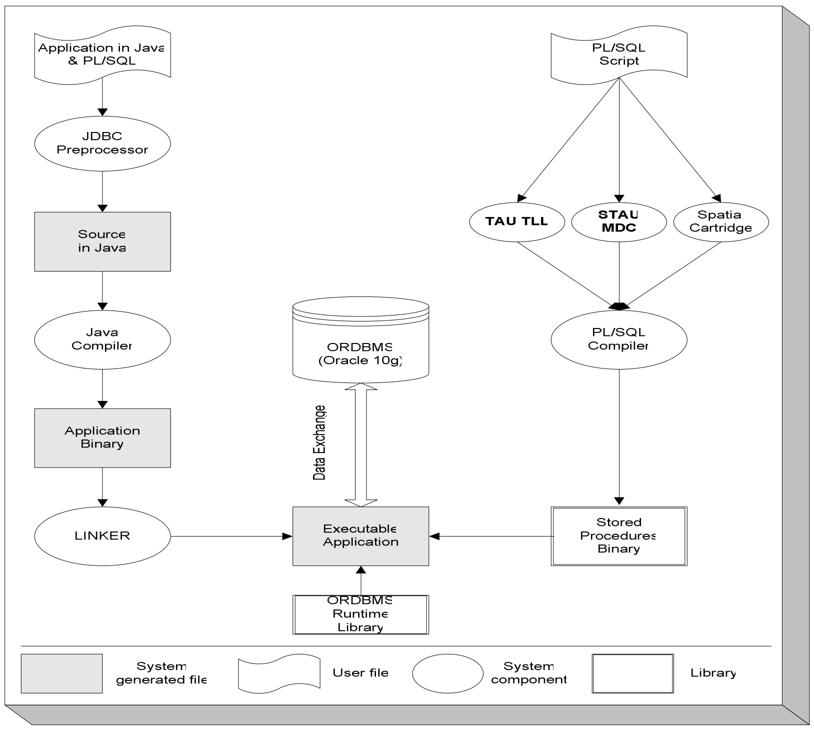


Figure 19 The architecture of the HERMES system

To demonstrate the functionality of the proposed HERMES-MDC, in the following paragraphs we present an application example related to truck motion analysis (**T**ruck **I**nformation **S**ystem TIS). The motivation is that a number of big truck-transportation companies need a flexible way to manage the motion of their lorries that transfer commercial goods. The benefits to be gained from a possible optimization of truck movements are straightforward. First of all, it is crucial for the company's people not only to know where the exact location of all trucks at any time is, but also to plan future routes in order to improve the effectiveness of their transportation network for the benefit of their clients. Furthermore, due to the fact that time is money, they must ensure that the trajectories of their trucks are optimal and that they avoid delays in the transportation of products. In general, the ability to model and query the spatio-temporal configuration and activities of such companies would be an important aid to the decision makers of the companies. By utilizing this application example, the expressive power and the applicability of HERMES-MDC in such a commercial domain are demonstrated.

In order to present the capabilities of the introduced data cartridge, we construct the following set of object-relational database tables.

Countries (name: Varchar2, territory: MD_SYS.Sdo_Geometry)
Highways (name: Varchar2, line: MD_SYS.Sdo_Geometry)
Sites (name: Varchar2, kind: Varchar2, location: MD_SYS.Sdo_Geometry)
Trucks (company: Varchar2, id: Varchar2, type: Varchar2, route: Moving_Point)
Weather (name: Varchar2, kind: Varchar2, extent: Moving_Polygon)

Countries relation is a set of polygon geometries describing the territories of several countries and *Highways* relation is a set of linestring geometries along which the lorries are supposed to be moving. *Sites* relation contains locations of certain landmarks, such as petrol stations, etc. *company* and *id* attributes of *Trucks* relation identify the *route* of a lorry that is modeled as a moving point. *type* attribute stamps each lorry with a characteristic description of each kind (e.g. refrigerator, container, etc.). *Weather* relation records weather events that could influence the route or the schedule of a truck, such as hurricanes, storms, or temperature maps. These events are given a name for identification purposes. *kind* attribute gives the type of the weather event, such as *"snowstorm"* or *"tornado"*, and *extent* attribute provides the time-varying region of each weather phenomenon.

In Figure 20, we illustrate a composite spatio-temporal query in the domain of our application example. The linguistic description of the query is followed by the implementation of the query in the form of a *PL/SQL* block, as well as by an abstract presentation of the way that such a query is resolved. This query illustrates the expressive power and the spatio-temporal query capabilities added to *PL/SQL* by HERMES-MDC. To start with, we exhibit that spatial objects (*greece*) as well as moving types (*truck245* and *stormA*) can be stored and retrieved by traditional database tables following the same syntax as if they were standard data types and, what is more, after retrieved they can be stored in local variables. As such, in order to answer Q1 we invoke a typical SQL statement that counts from the *Truck* relation the lorries that satisfy the *WHERE*-clause, which is the time-dependent version of *f_relate* operation.

To address Q2, we demonstrate how we can restrict a moving point inside a static spatial region and how to temporally and spatially project this restricted moving point in its initial position. The result of such an operation (*f_intersection*) in all cases is a *Moving_Object* that can be handled as any other moving geometry. By temporally projecting it (*f_temp_element*) on the continuous time line and finding the temporal element that consists of the time periods for which are defined the unit moving objects of the moving truck, we can estimate the timepoint when initially entered the territory of Greece. In addition, by applying the *f_initial* object method, we can locate the point that this happened.

The distance traversed over Greece by truck "245" (Q3) is resolved by finding the intersection of Greece with the trajectory followed by the truck (*f_trajectory* operation). This intersection is a *LineString* geometry that restricts the route of the lorry *inside* Greece and by applying *LENGTH* spatial operator upon the resulted *LineString* we compute the required distance. Furthermore, in order to provide the list of petrol stations (Q4), we select those *sites* that are petrol stations and the moving truck is within the specified distance (*f_within_distance* operation) at the time the query is invoked.

Finally, to simplify the resolution of Q5 let us assume that the truck resides at the beginning of a series of highways and that its destination is the ending point of these highways. As such, having a cursor to traverse (*FOR LOOP*) all highways, we choose that highway that is *disjoint* (*RELATE* operator) with the region traversed (*f_traversed* operation) by storm "A" and it has the smallest length (*LENGTH* operator). A complete description of the TIS application example can be found in [Pel02].

Q1: How many trucks of company "A" are traveling inside Greece right now?

Q2: If truck "245" of company "A" is one of them when and where did it enter the territory of Greece?

Q3: What distance does it traverse over Greece?

Q4: Give a list of options to the driver of the truck to refuel his truck in the next 50km.

Q5: Which is the best route, in terms of distance, that this truck can follow in order to avoid storm "A"?

```
DECLARE
    greece MDSYS.Sdo_Geometry;                    now TAU_TLL.TIMEPOINT_SEC;
    count_trucks pls_integer;                     truck245 Moving_Point;
    truck245_IN_Greece Moving_Object;             temp_projection TAU_TLL.TEMP_ELEMENT_SEC;
    when TAU_TLL.TIMEPOINT_SEC;                   where MDSYS.Sdo_Geometry;
    distance double;                              stormA Moving_Polygon;
    stormA_area MDSYS.Sdo_Geometry;               CURSOR highways IS SELECT * FROM Highways;
    highway_length, min_length number := 0;       best_highway MDSYS.Sdo_Geometry;
BEGIN
//Q1
    SELECT territory INTO greece FROM Countries WHERE name = "Greece";
    SELECT route INTO truck245 FROM Truck WHERE company="A"AND id=245;
    SELECT extent INTO stormA FROM Weather WHERE kind="storm" AND name="A";
    SELECT count (*) INTO count_trucks FROM Truck
         WHERE company = "A" AND  route.f_relate('INSIDE', greece, now) = 'INSIDE';
//Q2
    truck245_IN_Greece := truck245.f_intersection(greece);
    temp_projection := truck245_IN_Greece.f_temp_element ();
    when := temp_projection.te(temp_projection.te.FIRST).b;
    where := truck245_IN_Greece.f_initial();
//Q3
    distance:= LENGTH (INTERSECTION (greece, truck245.f_trajectory()));
//Q4
    SELECT name, location FROM Site
         WHERE kind = "petrol station" AND truck245.f_within_distance(50000, location, now) = 'TRUE';
//Q5
    stormA_area := stormA.f_traversed();
    FOR highways_rec IN highways LOOP
       IF RELATE (highways_rec.line, 'DISJOINT', stormA_area) = 'DISJOINT' THEN
            highway_length := LENGTH(highways_rec.line);
            IF highway_length < min_length THEN
                 min_ length:= highway_length; best_highway := highways_rec.line;
            END IF;
       END IF;
    END LOOP;
END;
```

Figure 20 TIS: An Application Example

6 Comparison with Related Work

Several research efforts have tried to model spatio-temporal databases using the *moving object* concept. In [EGS+99] the authors propose a new line of research where moving points and moving regions are viewed as three-dimensional (2D + time) or higher dimensional entities whose structure and behavior is captured by modeling them as abstract data types. Such abstract data types for moving points and moving regions have been introduced in [GBE+00], together with a set of operations on such entities. The authors argue that such a collection of types and operations, together with a number of related auxiliary data types, as pure spatial or temporal types and time-dependant real numbers, can be integrated into any extensible DBMS.

The model presented in [GBE+00] was the first attempt to deal with continuous motion while in [FGN+00] the definition of the discrete representation of the above-discussed abstract data types is presented. The interesting part of the discrete model is how *"moving"* types are represented. The authors describe the *sliced representation* behind which, the basic idea is to decompose the temporal development of a value into fragments called *"slices"* such that within the slice this development can be described by some kind of *"simple"* function. The sliced representation is built by a type constructor *"mapping"* parameterised by the type describing a single slice which we call a *unit* type. A value of a unit type is a pair *(I, u)*, where *I* is a time interval and *u* is a representation of a simple function defined within that time interval. More specifically, the unit types *ureal*, *upoint*, *upoints*, *uline* and *uregion* are defined. For values that can only change discretely, there is a *"const"* type constructor, which produces units, whose second component is just a constant of the argument type. This is in particular needed to represent moving integers, strings and boolean values. The *"mapping"* data structure basically just assembles a set of units and makes sure that their time intervals are disjoint.

The next step in this development was the study of algorithms for the rather large set of operations defined in [GBE+00]. Whereas [FGN+00] just provides a brief look into this issue by presenting two example algorithms at the end, in [LFG+03] the authors present a comprehensive, systematic study of algorithms for a subset of the operations introduced in [GBE+00]. Whereas some algorithms are relatively straightforward and simple, there are still a considerable number of quite involved ones. In all cases the authors analyze the complexity of the algorithms. In [LFG+03] the data structures from [FGN+00] are also refined and extended by auxiliary fields to speed up computations. This paper also offers a blueprint for implementing such a *"moving objects"* extension package for suitable extensible database architectures. More specifically, the details and the current status of a prototypical implementation of the data structures and algorithms described are presented. The prototype is being developed as an algebra module for the experimental database system SECONDO [DG00] and as an Informix Datablade.

As an extension to the abstract model in [GBE+00], the concept of *spatio-temporal predicates* is introduced in [ES02]. The goal is to investigate temporal changes of topological relationships induced by temporal changes of spatial objects. Further work on modeling includes [SXI01] where the authors focus on moving point objects and the inclusion of concepts of differential geometry (speed, acceleration) in a calculus based query language. In [VW01] the authors consider movement in networks and some evaluation strategies.

Another model using moving objects is proposed by Wolfson and colleagues in [SWC+97], [WXC+98] and [WSC+99]. The authors propose the so-called Moving Objects Spatio-Temporal (MOST) data model for databases with *dynamic attributes*, i.e. attributes that change continuously as a function of time, without being explicitly updated. This model enables the DBMS to predict the future location of a moving object by providing a *motion vector*, which consists of its location, speed and direction for a recent period of time. In the model, the answer to a query depends not only on the database contents, but also on the time at which the query is entered. As long as the predicted position based on the motion vector does not deviate from the actual position more than some threshold, no update to the database is necessary. An important issue here is to balance the cost of updates against the cost of imprecise information. The authors also offer a query language (Future Temporal Logic - FTL) based on temporal logic to formulate questions about the near future movement. The approach is restricted to moving points and does not address more complex time-varying geometries such as moving regions.

In the sequel and in order to place the contribution of this paper in the domain of spatio-temporal database research, we briefly present the differences of HERMES-MDC features with the approach described in [GBE+00], [FGN+00] and [LFG+03], which is the one most closely related to this work.

6.1 Comparison in terms of pure temporal or spatial semantics-functionality

To begin with, the temporal functionality needed to implement the spatio-temporal object data types is provided through the implementation of TLL [Kak96] as a data cartridge [Pel02]. *TAU-TLL* provides clear semantics for the time line including the time boundaries, time order, time reference, temporal granularities, and the supported calendar. In addition, it augments the four temporal literal data types found in ODMG object model, *Date*, *Time*, *Timestamp* and *Interval*, with three new temporal object data types: *Timepoint*, *Period* and *Temporal Element*. Furthermore, it provides an extensive set of object methods for these temporal types [Pel02]. The time line in [GBE+00] is continuous, isomorphic to real numbers. The time model adopted by *HERMES-MDC* is also considered to be continuous, but the model adopted by the TAU Temporal Object Model [Kak96] implemented in *TAU-TLL* is a discrete time model, isomorphic to integers. An appropriate kind of interaction between the *moving* and *temporal* data cartridges is adopted in order to support both time models.

Another advantage of our approach compared with the approach presented in [GBE+00] and [FGN+00], is that the temporal semantics in the latter are represented by only two types, namely *instant*, meaning a point in the linear continuous time line and *range (instant)*, meaning an interval between two time points. This limited temporal type system lacks the extensive set of operations defined for the temporal types introduced in TAU-TLL, because it is designed just to facilitate operations for spatio-temporal objects. This fact makes it difficult to be utilized in a stand-alone temporal application that is dependent to standard, non-spatial data. This is exactly the reason why the temporal functionality of *HERMES* system has been packaged as a distinct data cartridge [Pel02].

In HERMES system the spatial functionality is provided by Oracle as a separate data cartridge. The models proposed in the literature, such as the model presented in [GBE+00] and [FGN+00], provide separate objects for constructing different spatial geometries (e.g. points, lines, regions). In our case, we have a uniform representation of all kinds of geometries under the same spatial object (*Sdo_Geometry*), which increases the flexibility and the interoperability between moving types and pure spatial objects.

6.2 Comparison in terms of unified spatio-temporal semantics-functionality

HERMES-MDC is the integration of the above mentioned data cartridges. It introduces time-varying geometries that change location or shape in discrete steps and/or continuously. Our approach for supporting both discretely and continuously changing spatio-temporal objects and which is based on the *Unit_Function* object is more generic and flexible than the tactic adopted in [FGN+00] that asserts the same functionality. In addition to *Moving_Point*, *Moving_LineString*, *Moving_Polygon*, proposed in [FGN+00], *HERMES Type System* also includes types like *Moving_Circle*, *Moving_Rectangle*, *Moving_Collection* and *Moving_Object*. A rich set of object methods is introduced that expresses all the interesting spatio-temporal phenomena and processes. This set of operations is a superset of the operations introduced in [GBE+00] and the temporal variants operations provided by Oracle. The operation set commenced in [GBE+00] at an abstract level, is reduced in [FGN+00] where specific finite representations and data structures are given for all the types of the abstract model, and is further reduced in [LFG+03] where a subset of the algorithms are selected to make the implementation manageable.

Of course, there are more differences between the two operations sets supplied by [GBE+00] and HERMES-MDC. For example, all topological operations introduced in [GBE+00] are combined in HERMES-MDC under a single operator, which distinguishes the different topological relationships via a *"mask"* parameter. Furthermore, HERMES-MDC introduces new operations describing the buffer, the convex hull, the centre of gravity and points on the surface of moving geometries. Additionally, particular attention has been paid to operations that facilitate the user to check the construction of moving objects and to keep such kind of spatio-temporal data in a consistent state. This leads to effective database maintenance and reliable error-handling mechanism.

The *Moving_Collection* object supports not only a homogeneous collection of moving types but also a heterogeneous collection of them. In [GBE+00], heterogeneous collections are not supported and a single moving type corresponds to a homogeneous *Moving_Collection* of *HERMES*. The *Moving_Object* can substitute any of the other moving types, as well as moving geometries that result as operations on other moving geometries and moreover, it can model time-varying objects like the time-changing perimeter of a moving region. In [GBE+00] such degenerated moving types (moving reals, strings and booleans) are constructed as separate objects, which leads to a proliferation of object types that mainly are not spatio-temporal, which additionally delays, makes more difficult and unnatural the utilization of such data types by end users.

Generally speaking, *HERMES Type System* is richer and more flexible than the one presented in [GBE+00]. For example, moving linestrings that intersect themselves during their development are adopted by HERMES type system, while they are forbidden in [GBE+00] due to the fact that the spatial model does not accept self-intersecting linestrings.

Apart from linear interpolations of spatial and spatio-temporal (moving) types utilized in [FGN+00] and [LFG+03], HERMES also utilizes arc interpolations. What is more, the user of *HERMES-MDC* is facilitated with a flexible and extensible interface for additional types of motion for moving types (e.g. splines, polynomials of degree higher than two etc.), which is provided via the *Unit_Function* object type.

Moreover, in *HERMES* there has been an effort to minimize the employment of algorithms from computational geometry like those presented in [FGN+00] and subsequently in [LFG+03]. This design decision as well as the utilization of recursive algorithms for some operations on moving types offers substantial reduction in the complexity of these methods, which leads to better performance.

7 Conclusions and Future Work

In this paper, an Oracle data cartridge for moving objects, called HERMES-MDC, was introduced. This data cartridge is a system extension that provides spatio-temporal functionality to Oracle10g ORDBMS and supports modelling and querying of moving objects changing location either in discrete steps or continuously (moreover, it can be used as a pure temporal or a pure spatial system). A collection of data types and their corresponding operations are defined, developed and provided using the extensibility interface of Oracle10g. Finally, this work has been evaluated through a mobile location application example. This benchmark demonstrates that embedding the functionality offered by HERMES-MDC in Oracle's data manipulation language provides a flexible, expressive and easy to use query language for spatio-temporal data.

We believe that a major contribution of the current work is that it prescribes straightforward future research directions. First of all, due to the fact that our study concerns only two-dimensional spatial objects as well as the change and motion of such geometries in the 2D Cartesian plane, there is need to investigate the way we could model surfaces and three-dimensional spatial objects and the time-changing variants of them. Additionally, a future direction we are planning to follow is to consider the query optimization extensibility interface of Oracle10g in order to enhance the performance of *HERMES* data cartridges. In parallel, we definitely plan to utilize the extensible indexing mechanism of Oracle10g so as to attach an appropriate indexing method to speed up searching for the moving objects we introduce. Finally, an interesting research exploration is to investigate generic mechanisms to map the above discussed object-relational constructs to concepts coming from the object-oriented paradigm.

8 Acknowledgments

Nikos Pelekis and Yannis Theodoridis were partially supported by EPEAEK II / Pythagoras 2.2.3 project "Spatio-temporal Data and Knowledge Management in Expert Virtual Environments" of the Greek Ministry of National Education and Religious Affairs.

9 References

[AR99] T. Abraham, J.F. Roddick. Survey of Spatio-Temporal Databases. *GeoInformatica*, 3:61-99, 1999.

[CB97] R.G.G. Cattel, D.K. Barry (eds.). *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, May 1997.

[DG00] S. Dieker and R. H. Güting. Plug and Play with Query Algebras: Secondo. A Generic DBMS Development Environment. Proc. *Int'l Symp. of Database Engineering and Applications (IDEAS)*, pages 380-390, September 2000.

[EF91] M. Egenhofer and R. Franzosa. Point-Set Topological Spatial Relations. *International Journal of Geographical Information Systems*, 5(2): 161-174, 1991.

[EGS+99] M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3): 265-291, 1999.

[ES02] M. Erwig and M. Schneider. Spatio-Temporal Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 14(4): 881-901, 2002.

[FGN+00] L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider. A Data Model and Data Structures for Moving Objects Databases. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Dallas, Texas, USA, 2000.

[FP97] S. Feuerstein and B. Pribyl. *Oracle PL/SQL Programming.* O'Reilly & Associates, 1997.

[FS98] M. Fowler and K. Scott. *UML Distilled – Applying the standard object modeling language.* Addison-Wesley, 1998.

[GBE+00] R.H. Güting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1): 1-42, 2000.

[Güt94] Güting, R.H. An Introduction to Spatial Database Systems. *VLDB Journal,* 4: 357-399, 1994.

[Kak96] I. Kakoudakis. The TAU Temporal Object Model. MPhil Thesis, UMIST, Department of Computation, 1996.

[KS+03] M. Koubarakis, T. Sellis et al. (eds.). *Spatio-temporal Databases: The Chorochronos Approach.* Springer, 2003.

[LFG+03] J. A. C. Lema, L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider. Algorithms for Moving Objects Databases. *The Computer Journal* 46(6): 680-712, 2003.

[OGIS] Open Geospatial Consortium, http://www.opengeospatial.org (accessed on 16 March 2005).

[Ora03a] Oracle Corp. *Oracle® Data Cartridge Developer's Guide 10g Release 1 (10.1).* Oracle Database Documentation Library, December 2003. Available at http://www.oracle.com/pls/db10g/portal.portal_demo3?selected=7 (accessed on 16 March 2005).

[Ora03b] Oracle Corp. *Oracle® Spatial User's Guide and Reference 10g Release 1 (10.1).* Oracle Database Documentation Library, December 2003. Available at http://www.oracle.com/pls/db10g/portal.portal_demo3?selected=7 (accessed on 16 March 2005).

[Pel02] N. Pelekis. HERMES: A spatio-temporal extension to ORACLE DBMS. PhD Thesis, UMIST, Department of Computation, 2002.

[Peu01] D. Peuquet. Making Space for Time: Issues in Spase-Time Data Representation. *GeoInformatica*, 5: 11-32, 2001.

[PTK+05] N. Pelekis, B. Theodoulidis, I. Kopanakis, Y. Theodoridis. Literature Review of Spatio-Temporal Database Models. *Knowledge Engineering Review*, in press, 2005.

[Ren97] A. Renolen. Temporal Maps and Temporal Geographical Information Systems (Review of Research). Department of Surveying and Mapping, The Norwegian Institute of Technology, February 1997.

[SWC+97] P. Sistla, O. Wolfson, S. Chamberlain, S.Dao. Modeling and Querying Moving Objects. *Proc. 13th Int'l Conf. on Data Engineering (ICDE13)*, Birmingham, UK, 1997.

[SXI01] J. Su, H. Xu and O. Ibarra. Moving Objects: Logical Relationships and Queries. *Proc. 7th Int'l Symp. on Spatial and Temporal Databases (SSTD)*, Redondo Beach, California, USA, 2001.

[TCG+93] Tansel, A.U., J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass. *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings Publishing Company, 1993.

[TL91] I. Theodoulidis and P. Loucopoulos. The Time Dimension in Conceptual Modeling. *Information Systems,* 16 (3): 273-300, 1991.

[VW01] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. *Proc. 7th Int'l Symp. on Spatial and Temporal Databases (SSTD)*, Redondo Beach, California, USA, 2001.

[WJL91] G. Wiederhold, S. Jajodia and W. Litwin. Dealing with Granularity of Time in Temporal Databases. Proc. *3rd Nordic Conf. on Advanced Information Systems Engineering*, Trondheim, Norway, May 1991.

[Wor94] M.F. Worboys. Unifying the Spatial and Temporal Components of Geographical Information. *Information Symposium on Spatial Data Handling*, 1994.

[WSC+99] O. Wolfson, A. P. Sistla, S. Chamberlain and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7 (3): 257-387, 1999.

[WXC+98] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions. *Proc. 10th Int'l Conf. on Scientific and Statistical Database Management*, Capri, Italy, 1998.

10 Appendix A – the Temporal Data Model Adopted by HERMES-MDC

TAU Model augment the four temporal data types found in ODMG object model, *Date*, *Time*, *Timestamp* and *Interval*, with three new temporal data types: *Timepoint*, *Period* and *Temporal Element*. In the following sections, the semantics and the formal definitions of all the temporal literal types supported by *TAU Time Model* are given, as well as the formal specifications of the atomic literal types that are utilized in the definition of the temporal types, in terms of set theory.

Operations related with each temporal type fall into three categories namely, constructors, access methods and utilities.

- ❑ *Constructors* are operations that create instances of a type and initialize their state.
- ❑ *Access Methods* are operations used to retrieve values of built-in properties.
- ❑ *Utilities* are operations that return general information regarding the instance.

10.1 Atomic Literal Types

The set of *Atomic Literal Types ALT* is defined as

$ALT = \Pi\, boolean T \cup \Pi\, char T \cup \Pi\, short T \cup \Pi\, ushort T \cup \Pi\, long T \cup \Pi\, ulong T \cup \Pi\, float T \cup \Pi\, double T \cup \Pi\, octet T \cup \Pi\, string T \cup \Pi\, enum T$, *where*

$\Pi\, boolean T = \{true, false\}$ $\quad$ $\Pi\, char T = \{x \mid x \in ASCII\}$

$\Pi\, short T = \{x: \wedge \mid s\_lb \leq x \leq s\_ub\}$ $\quad$ $\Pi\, ushort T = \{x: \subseteq \mid x \leq us\_ub\}$

$\Pi\, long = \{x: \wedge \mid l\_lb \leq x \leq l\_ub\}$ $\quad$ $\Pi\, ulong T = \{x: \subseteq \mid x \leq ul\_ub\}$

$\Pi\, float T = \{x: \nabla \mid f\_lb \leq x \leq f\_ub\}$ $\quad$ $\Pi\, double T = \{x: \nabla \mid d\_lb \leq x \leq d\_ub\}$

$\Pi\, bit T = \{0, 1\}$ $\quad$ $\Pi\, octet T = bit^8$

$\Pi\, string T = Char^n,\ n \in \subseteq^*$ $\quad$ $\Pi\ enum T = \{(s, n) \mid s \in string,\ n \in any\ numerical\ type\}$

s_lb, l_lb, f_lb, d_lb are the lower bounds and *s_ub, us_ub, l_ub, ul_ub, f_ub, d_ub* are the upper bounds of the corresponding numerical types. The representation, precision, ranges and operations of numerical types are implementation platform specific.

Further more, in order to formalize the definition of the temporal literal types we should first define the time divisions in the *Gregorian* calendar, which are,

$\Pi\, GrYear T = \{y: long \mid lb \leq y \leq ub \wedge y \neq 0\}$ $\quad$ $\Pi\, GrMonth T = \{m: ushort \mid 1 \leq m \leq 12\}$

$\Pi\, GrDay T = \{d: ushort \mid 1 \leq d \leq 31\}$ $\quad$ $\Pi\, GrHour T = \{h: ushort \mid 0 \leq h \leq 23\}$

$\Pi\, GrMinute T = \{m: ushort \mid 0 \leq m \leq 59\}$ $\quad$ $\Pi\, GrSecond T = \{s: double \mid 0 \leq s \leq 59\}$

as well as the set *granularity* that contains elements that represent time accuracy:

$\Pi\, granularity T = \{YEAR, MONTH, DAY, HOUR, MINUTE, SECOND\}$

As such the set of Temporal Literal Types *TLT* is defined as

TLT = *Π dateT ∪ Π timeT ∪ Π timestampT ∪ Π timepoint⟨ g⟩T ∪ Π intervalT ∪ Π period⟨ g⟩T ∪ Π temporalElement⟨ g⟩T.*

10.2 ODMG Temporal Data Types

The *ODMG* Standard [CB97] defines the following temporal data types:

- *Date*: Instances of the *Date* type represent unique points in time. It supports the fields YEAR, MONTH and DAY.

  $$date =_d \langle\!\langle year: GrYear,\ month: GrMonth,\ day: GrDay \rangle\!\rangle$$

- *Time*: The *Time* data type supports the fields HOUR, MINUTE and SECOND. It either represents a unique point in time (for which the date is implicit) or it represents a recurring point of time. It is possible to specify a precision, i.e. the number of decimal places of accuracy to which the SECOND field will be kept. The default precision is zero (whole seconds only). The maximum precision is implementation defined (at least 6). The *Time* data type has a WITH TIME ZONE option. If the option is not specified the values of the data type are assumed to be always in the current default time zone of the user session. If the option is specified then the values of the data type include the TIMEZONE_HOUR and TIMEZONE_MINUTE fields, which specify the offset of the time zone of the rest of the value from Universal Coordinated Time.

  $$time =_d \langle\!\langle hour: GrHour,\ minute: GrMinute,\ second: GrSecond \rangle\!\rangle$$

- *Timestamp*: The *Timestamp* data type supports the fields YEAR, MONTH, DAY, HOUR, MINUTE and SECOND. It represents unique points in time. With *Timestamp* data type it is possible to specify a precision and WITH TIME ZONE option *(*see *Time* data type*).*

  $$timestamp =_d date \mid\mid time$$

- *Interval*: The *Interval* data type is used to represent an unanchored duration of time. Every interval data type consists of a contiguous subset of the fields: DAY, HOUR, MINUTE and SECOND.

  $$interval =_d \langle\!\langle day: long,\ hour: GrHour,\ minute: GrMinute,\ second: GrSecond \rangle\!\rangle$$

10.3 Advanced Temporal Data Types

We augment the four temporal literal data types found in *ODMG* object model [CB97] with three new temporal object data types presented below:

- *Timepoint*: *TAU Model* extends the *Timestamp* data type to include granularity. The new data type is a subtype of the *Timestamp* data type. It inherits all the properties and the operations that are defined for the *Timestamp* data type. It refines all the operations, which had as argument *Timestamp* to *Timepoint.*

  $$timepoint\langle g\rangle =_d tp\langle g\rangle \cup STV \text{ where}$$

  $$tp\langle year\rangle =_d \langle\!\langle year: GrYear \rangle\!\rangle,\ tp\langle month\rangle =_d tp\langle year\rangle \mid\mid \langle\!\langle month: GrMonth \rangle\!\rangle, \ldots,$$

$tp\langle second\rangle =_d tp\langle minute\rangle \,\|\, \langle\!\langle second{:}GrSecond\rangle\!\rangle$ *and* $STV =_d \{beginning,\ forever,\ now\}$

Beginning and *forever* are defined to be members of *timepoint* such as

$$\forall t \in timepoint\langle g\rangle \cdot beginning \leq t \leq forever$$

- *Period*: The *Period* data type is used to represent an anchored duration of time, that is, duration of time with starting and ending points. A period has an associated granularity. The period is the composition of two timepoints with the constraint that the timepoint that starts the period equals or precedes the timepoint that terminates it. Without loss of generality, it is assumed that both timepoints have the same granularity.

$$period\langle g\rangle =_d \{\langle\!\langle start{:}Timepoint\langle g\rangle,\ end{:}Timepoint\langle g\rangle\rangle\!\rangle \mid start \leq end\},\ g \in granularity$$

There are four categories of periods depending on whether they include their starting and/or their ending timepoints or not: $[T_1, T_2]$ (closed-closed), $[T_1, T_2)$ (closed-open), $(T_1, T_2]$ (open-closed), and (T_1, T_2) (open-open). Without loss of generality, *TAU Model* supports only closed-open periods, with which it is possible to model any other category. For example, the period $[T_1, T_2]$ is equivalent to the period $[T_1, T_2+1$ "granule"). The meaning of "1 granule" depends on the granularity of the period. For instance, if the granularity is day then the period $[T_1, T_2]$ is equivalent to the period $[T_1, T_2+1{*}DAY)$.

- *Temporal Element*: The *Temporal Element* data type is used to represent a finite union of disjoint periods. Temporal elements are closed under the set theoretic operations of union, intersection and complementation.

$$temporalElement\langle g\rangle =_d \{te{:}\ set\langle period\langle g\rangle\rangle \mid \forall i, j \cdot i \neq j \Rightarrow te_i \cap te_j = \emptyset\}$$

11 Appendix B – the Spatial Data Model Adopted by HERMES-MDC

11.1 Description of Spatial Data Types

The spatial data model adopted by Oracle10g is a hierarchical structure consisting of elements, geometries, and layers, which correspond to representations of spatial data. Layers are composed of geometries, which in turn are made up of elements. For example, a point might represent a building location, a line string might represent a road or flight path, and a polygon might represent a state, city, or zoning district.

Element: An *element* is the basic building block of a geometry. The supported spatial element types in the object-relational model are points, simple, arc (circular arcs) and compound linestrings and polygons, as well as circles and rectangles as sub-cases of polygon geometries. Figure 21 illustrates the supported geometric primitive types. *Point* is the simplest geometry, which consists of one coordinate. Each coordinate in an element is stored as a *(x, y)* pair often corresponding to longitude and latitude. *LineStrings* are composed of one or more pairs of points that define line segments. *Polygons* are composed of connected linestrings that form a closed ring and the interior of the polygon is implied.

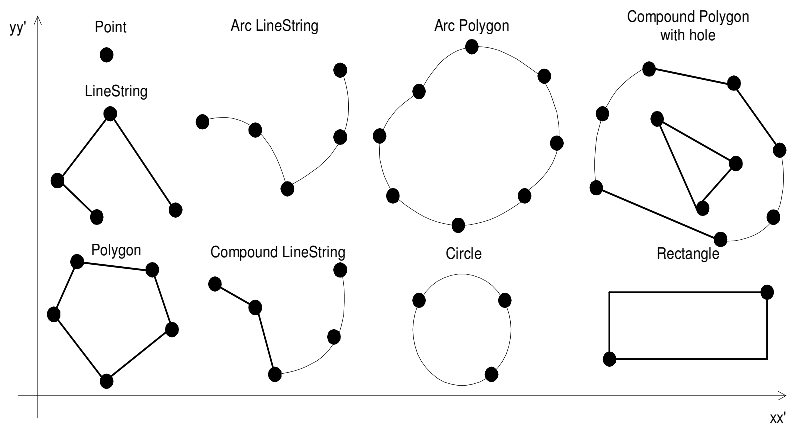


Figure 21 Primitive Geometry Types Supported by Oracle10g

As it is obvious in Figure 21, *arc* and *compound* types generalize the LineString and Polygon types, to represent geometries with arbitrary interpolations but the same topology. Self-crossing polygons are not supported although self-crossing linestrings are (see Figure 22). If a linestring crosses itself, it does not become a polygon. A self-crossing linestring does not have any implied interior. The exterior ring and the interior ring of a polygon with holes are considered as two distinct elements that together make up a complex polygon.

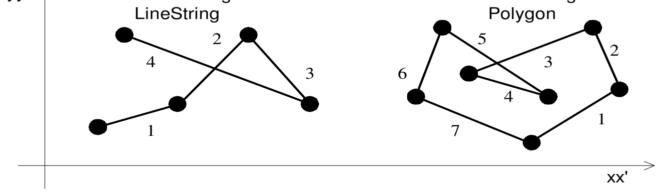


Figure 22 Self-crossing LineString & Polygons

Geometry: A *geometry* (or *geometry object*) is the representation of a spatial feature, modelled as an ordered set of primitive elements. A geometry can consist of a single element, which is an instance of one of the supported primitive types, a homogeneous or heterogeneous collection of elements. A multipolygon, such as one used to represent a set of islands, is a homogeneous collection. A heterogeneous collection is one in which the elements are of different types.

Layer: A *layer* is a heterogeneous collection of geometries having the same attribute set. For example, one layer in a Geographical Information System (GIS) might include topographical features, while another might describe population density, and a third describes the network of roads and bridges in the area (lines and points). Each layer's geometries are stored in the database in standard tables.

11.2 Object Orientation and Geometry Hierarchy

Until now we have clarified all the geometric types that our model supports. In Figure 23, one can see the geometry interface hierarchy adopted by the proposed spatial model and developed as an extension of the Open GIS geometry model [OGIS]. In the proposed model, a geometry object can be either a simple geometry or a geometry collection. A simple geometry is defined as previously, while a geometry collection is a heterogeneous collection of points, linestrings and polygons. More specific types like multipoint, multicurve and multisurface are introduced to represent homogeneous collections of points, linestrings and polygons respectively for easier geospatial analysis.

In Figure 23, the white blocks are helper interfaces i.e., Segment, LinearSegment, CircularArc, Spline and other potential interpolations of a Segment. The dark-shaded blocks are the Open GIS types i.e., Geometry, Point, Curve, Surface, LineString, Polygon, GeometryCollection, MultiPoint, MultiCurve, MultiSurface, MultiLineString and MultiPolygon. The light-shaded blocks are the extended types i.e., CurveString, CurvePolygon, MultiCurveString and MultiCurvePolygon.

The Curve, Surface, Multicurve and Multisurface are intermediate abstract types that make this model more flexible for expansion. A *curve* is an arbitrary topologically one-dimensional geometry object. A *surface* is an arbitrary topologically two-dimensional geometry object that may or may not be plane. A *multicurve* and a *multisurface* represent collections of curves and surfaces, respectively.

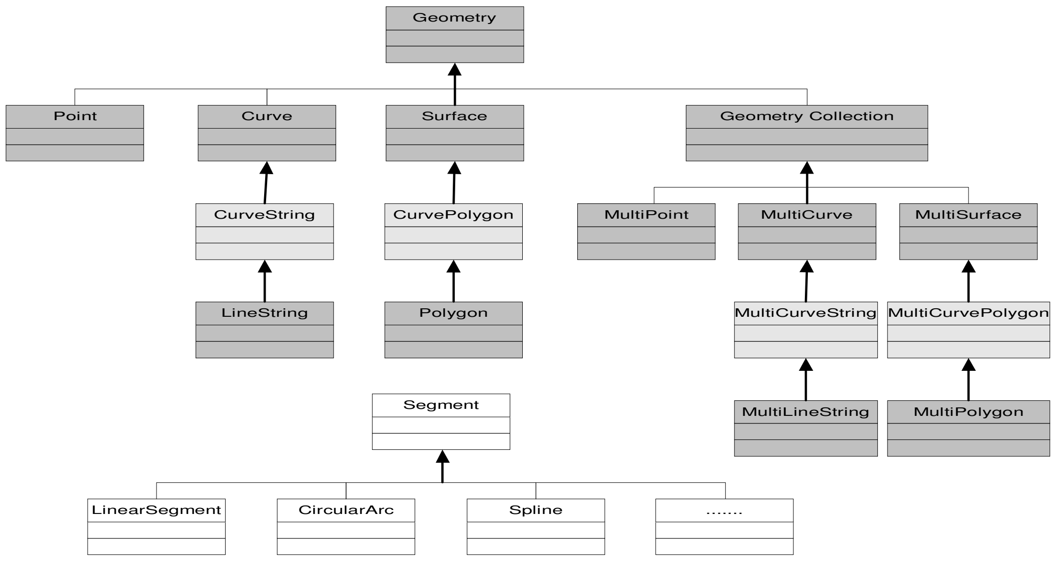


Figure 23 Geometry Interface Hierarchy

The proposed geometry interface hierarchy is fully compatible with the Open GIS model because the existence of the extended types does not affect the inheritance relationships when developers implement linear interfaces only. In fact, the new CurveString and CurvePolygon interfaces generalize the LineString and Polygon interfaces respectively, to represent geometries with arbitrary interpolations but the same topology as traditional Open GIS geometries.

11.3 Structures for Spatial Data Types

In the spatial object-relational model, the geometric description of a spatial object is stored in a single row, in a single column of object type *SDO_GEOMETRY* (defined under the *MDSYS* Oracle user) in a user-defined table. Any table that has a column of type *SDO_GEOMETRY* must have another column, or set of columns, that define a unique primary key for that table. This object type corresponds to the most general type defined in the interface hierarchy of Figure 23. Each subtype is declared and stored in a database table as an *SDO_GEOMETRY* object and the knowledge of which sub-type is or what its special characteristics are, are embodied in the structure of this generic object. Oracle Spatial defines *SDO_GEOMETRY* object type as:

```
CREATE TYPE SDO_GEOMETRY AS OBJECT (
    SDO_GTYPE NUMBER,
    SDO_SRID NUMBER,
    SDO_POINT SDO_POINT_TYPE,
    SDO_ELEM_INFO MDSYS.SDO_ELEM_INFO_ARRAY,
    SDO_ORDINATES MDSYS.SDO_ORDINATE_ARRAY );
```

The sections that follow describe the semantics of each *SDO_GEOMETRY* attribute, and some usage considerations.

SDO_GTYPE indicates the type of the geometry. Valid geometry types correspond to those specified in the *geometry interface hierarchy*. The following table shows the valid *SDO_GTYPE* values and the correspondence between the names and semantics.

Value	Geometry Type	Description
*d*000	UNKNOWN_GEOMETRY	Spatial ignores this geometry
*d*001	POINT	Geometry contains one point
*d*002	LINESTRING	Geometry contains one line string
*d*003	POLYGON	Geometry contains one polygon with or without holes
*d*004	COLLECTION	Geometry is a heterogeneous collection of elements
*d*005	MULTIPOINT	Geometry has multiple points
*d*006	MULTILINESTRING	Geometry has multiple linestrings
*d*007	MULTIPOLYGON	Geometry has multiple, disjoint polygons (more than one exterior boundary)

Table 1 Valid SDO_GTYPE Values

For a polygon with holes, the user should enter the exterior boundary first, followed by any interior boundaries. In a multi-polygon all polygons in the collection must be disjoint. The *d* in the *Value* column of the previous table is the number of dimensions: 2, 3, or 4. For example, a value of 2003 indicates a 2-dimensional polygon. For the time only 2-dimensional geometries are supported. The number of dimensions reflects the number of coordinates used to represent each vertex (for example, *(x,y)* for 2-dimensional objects or *(x,y,z)* for 3-dimensional objects). Points and lines are considered to be 2-dimensional objects. In any given *layer* (column), all geometries must have the same number of dimensions. For example, we cannot mix 2-dimensional and 3-dimensional data in the same layer.

SDO_SRID is intended to be a foreign key in a spatial reference system definition table, in order to integrate support into Oracle10g for storing and manipulating *SDO_GEOMETRY* objects in a variety of coordinate systems.

SDO_POINT is defined using an object type with attributes *x*, *y* and *z* of type *NUMBER*. If the *SDO_ELEM_INFO* and *SDO_ORDINATES* arrays are both null, and the *SDO_POINT* attribute is non-null, then the *x* and *y* values are considered to be the coordinates for a point geometry. Otherwise the *SDO_POINT* attribute is ignored.

SDO_ELEM_INFO is defined using a varying length array of numbers. This attribute helps to interpret the ordinates stored in the *SDO_ORDINATES* attribute (see section 3.2.1.5). Each triplet set of numbers is interpreted as follows:

SDO_STARTING_OFFSET indicates the offset within the *SDO_ORDINATES* array where the first ordinate for this element is stored.

SDO_ETYPE indicates the type of the element. Valid values are 0 through 5, as well as the following: 1003 and 2003 (variants of 3), and 1005 and 2005 (variants of 5). *SDO_ETYPE* values 1, 2, and 3 concern *simple elements*. They are defined by a single triplet entry in the *SDO_ELEM_INFO* array. Moreover, the following are considered variants of type 3, with the first digit indicating *exterior* (1) or *interior* (2):

- 1003: exterior polygon ring (must be specified in counter-clockwise order)
- 2003: interior polygon ring (must be specified in clockwise order)

SDO_ETYPE values 4 and 5 concern *compound elements*. They contain at least one header triplet with a series of triplet values that belong to the compound element. The elements of a compound element are contiguous. The last point of a subelement in a compound element is the first point of the next subelement. The point is not repeated.

SDO_ETYPE	SDO_INTERPRETATION	Meaning
0	0	Unsupported element type. Ignored by the Spatial functions and procedures.
1	1	Point type.
1	*n* >1	Point cluster with *n* points.
2	1	Line string whose vertices are connected by straight-line segments.
2	2	Line string made up of a connected sequence of circular arcs.
3	1	Simple polygon whose vertices are connected by straight-line segments.
3	2	Polygon made up of a connected sequence of circular arcs that closes on itself.
3	3	Rectangle type. A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it.
3	4	Circle type. Described by three points, all on the circumference of the circle.
4	*n* > 1	Line string with some vertices connected by straight-line segments and some by circular arcs.
5	*n* > 1	Compound polygon with some vertices connected by straight-line segments and some by circular arcs.

Table 2 Values and Semantics of SDO_ELEM_INFO

SDO_INTERPRETATION can mean one of two things, depending on whether or not *SDO_ETYPE* is a compound element. If the *SDO_ETYPE* is a compound element (4 or 5), this field specifies how many subsequent triplet values are parts of the element. If the *SDO_ETYPE* is not a compound element (1, 2, or 3), the interpretation attribute determines how the sequence of ordinates for this element is interpreted. For example, a line string or polygon boundary may be made up of a sequence of connected straight-line segments or circular arcs. If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the *SDO_ORDINATES* varying length array. The semantics of each *SDO_ETYPE* element and the relationship between the *SDO_ELEM_INFO* and *SDO_ORDINATES* varying length arrays for each of these *SDO_ETYPE* elements are given in the following table.

Each circular arc in the geometries is described using three coordinates: the arc's starting point, any point on the arc, and the arc's end point. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a linestring made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc, where point 3 is only stored once.

For polygon geometries the user needs to specify a point for each vertex, and the last point specified must be identical to the first (to "close" the polygon). For example, for a 4-sided polygon, we need to specify 5 points, with point 5 the same as point 1.

For compound elements the value, *n*, in the interpretation column specifies the number of contiguous subelements that make up the geometry. The next *n* triplets in the *SDO_ELEM_INFO* array describe each of these subelements. The subelements can only be of *SDO_ETYPE* 2. The end point of a subelement is the start point of the next subelement, and it must not be repeated.

SDO_ORDINATES is defined using a varying length array of *NUMBER* type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the *SDO_ELEM_INFO* varying length array. The values in the array are ordered by dimension. For example, a polygon whose boundary has four 2-dimensional points is stored as $\{x_1,y_1, x_2,y_2, x_3,y_3, x_4,y_4, x_1,y_1\}$.

The values in the *SDO_ORDINATES* array must all be valid and non-null. There are no special values used to delimit elements in a multi-element geometry. The start and end points for the sequence describing a specific element are determined by the *STARTING_OFFSET* values for that element and the next element in the *SDO_ELEM_INFO* array as explained previously.

Usage considerations: The *Spatial Data Cartridge* user should use the *SDO_GTYPE* values as shown in table 1. The *Spatial* component enforces some geometry consistency constraints and more specifically, the following:

- For *SDO_GTYPE* values *d*001 and *d*005, any subelement not of *SDO_ETYPE* 1 is ignored.
- For *SDO_GTYPE* values *d*002 and *d*006, any subelement not of *SDO_ETYPE* 2 or 4 is ignored.

For *SDO_GTYPE* values *d*003 and *d*007, any subelement not of *SDO_ETYPE* 3 or 5 is ignored. (This includes *SDO_ETYPE* variants 1003, 2003, 1005, and 2005).

11.4 Formal Definition of Pure Spatial Types

This section describes formally in terms of set theory the unique object type that represents all the different geometric constructs adopted in our spatial model.

$ST = \Pi SDO\_GEOMETRYT$

Before introducing the constraints and the interdependencies between the element types that compose the *SDO_GEOMETRY* object, let us first define these components, in order to associate conceptually their formal description with their linguistic one in Section 12.3. The reader should have in mind that even though the spatial model supports geometries of higher dimension than two, we are interested in 2-Dimensional spatial objects only.

- $\Pi SDO\_GTYPE\_TYPET = \{gt: ushort \mid 2000 \leq gt \leq 2007\}$
- $\Pi SDO\_POINT\_TYPET = \{(x, y) \mid x, y \in double\}$
- $\Pi SDO\_ELEM\_INFO\_ARRAYT = \{set\langle (so, et, ip) \rangle \mid so \in SDO\_STARTING\_OFFSET \wedge et \in SDO\_ETYPE \wedge ip \in SDO\_INTERPRETATION \wedge \forall i, j \cdot i < j \Rightarrow so_i < so_j \}$
  where

$\Pi SDO\_STARTING\_OFFSET\,T = \{so{:}\,ulong \mid 1 \le so < LAST(ORD)\}$

$\Pi SDO\_ETYPE\,T = \{et{:}\ ushort \mid 0 \le et \le 5\ \vee et \in \{1003, 2003, 1005, 2005\}\}$

$\Pi SDO\_INTERPRETATION\,T = \{ip{:}\ ushort\}$

- $\Pi SDO\_ORDINATES\,T = \{set\langle x \rangle \mid x \in double,\ |set\langle x \rangle| = 2k,\ k \ge 0,\ k \in ulong\}$

As such *SDO_GEOMETRY* is defined as follows:

$SDO\_GEOMETRY =_d \{\langle\!\langle\ SDO\_GTYPE{:}\ SDO\_GTYPE\_TYPE,$

$SDO\_SRID{:}\ ushort,$

$SDO\_POINT{:}\ SDO\_POINT\_TYPE,$

$SDO\_ELEM\_INFO{:}\ SDO\_ELEM\_INFO\_ARRAY,$

$SDO\_ORDINATES{:}\ SDO\_ORDINATES\_ARRAY \rangle\!\rangle \mid$

/ Due to space limitations we use the following abbreviations:*

SDO_GTYPE_TYPE:=GTYPE

SDO_POINT_TYPE:=PTYPE

SDO_ELEM_INFO_ARRAY:=ELEM

*SDO_ORDINATES_ARRAY:=ORD */*

(i) $(\forall gt \in GTYPE \cdot gt \in \{2001, 2005\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM{:}\ et=1) \wedge$

(ii) $(\forall gt \in GTYPE \cdot gt \in \{2002, 2006\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM{:}\ et=2\ \vee et=4) \wedge$

(iii) $(\forall gt \in GTYPE \cdot gt \in \{2003, 2007\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM{:}\ et=3\ \vee et=5) \wedge$

(iv) $(\forall gt \in GTYPE \cdot gt=2001 \Rightarrow (PTYPE \ne \varnothing \wedge (ELEM = \varnothing \wedge ORD = \varnothing)) \vee (PTYPE = \varnothing \wedge (ELEM = (1, 1, 1) \wedge |ORD| = 2))) \wedge$

(v) $(\forall gt \in GTYPE \cdot gt=2002 \Rightarrow |ORD| \ge 4 \wedge (ORD_1 \ne ORD_{LAST(ORD)-1} \wedge ORD_2 \ne ORD_{LAST(ORD)}) \wedge (\forall j=2k+1,\ k \ge 0,\ k \in ulong{:}\ point\ (ORD_j, ORD_{j+1}) \ne point\ (ORD_{j+2}, ORD_{j+3})) \wedge (\forall j=2k+1,\ k \ge 0,\ k \in ulong{:}\ collinear\ (segment\ (point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3})), segment\ (point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})) \Rightarrow \neg overlap(segment\ (point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3})), segment\ (point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})))) \wedge (\forall j=2k+1,\ k \ge 0,\ k \in ulong{:}\ arcline\ ((point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})) \Rightarrow \neg collinear\ ((point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})))) \wedge$

(vi) $(\forall gt \in GTYPE \cdot gt=2003 \Rightarrow (\exists (so, et, ip) \in ELEM{:}\ ip=3 \Rightarrow \neg parallel\ (segment\ (point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3})), xx') \wedge \neg parallel\ (segment\ (point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3})), yy')) \wedge (\exists (so, et, ip) \in ELEM{:}\ ip=4 \Rightarrow \neg collinear\ ((point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3}), point\ (ORD_{so+4}, ORD_{so+5}))) \wedge (\forall j,\ 1 \le j \le |ELEM| DIV3,\ j \in ulong \cdot (so_j, et_j, ip_j) \in ELEM{:}\ et_j=3\ \vee et_j=5 \Rightarrow Polygon_j =_d \{ORDso_j, \ldots, ORDso_{j+1}-1\} \subset ORD \cdot linestringsOfPolygon_j =_d \{set\langle\ linestring{:}\ \langle ORD_k, \ldots, ORD_l \rangle\rangle \mid so_j \le k < l \le so_{j+1}-1\ \wedge\ \forall l, m \in linestring{:}\ l=next(m),\ l \ne m \Rightarrow meet\ (l, m) \wedge \neg intersect\ (l, m) \wedge \neg touch\ (l, m) \wedge (ORDso_j = ORDso_{j+1}-2\ \wedge\ ORDso_j+1 = ORDso_{j+1}-1)\ \wedge\ \forall\ linestring{:}\ \text{(v)}\ rules\ applied\ to\ \langle ORD_k, \ldots, ORD_l \rangle\ instead\ to\ all\ ORD\} \cdot \forall\ m,\ 1 < m{:}\ inside(Polygon_m, Polygon_1) \wedge counter\text{-}$

$clockwise(Polygon_1) \wedge clockwise(Polygon_m) \wedge \forall m_1, m_2\ 1<m_1 \wedge 1<m_2: m_1 \neq m_2 \Rightarrow disjoint(m_1, m_2)))$

$\wedge$

(vii) $(\forall\ gt \in GTYPE \cdot gt=2004 \Rightarrow \forall\ j,\ 1\leq j\leq |ELEM|DIV3,\ j\in ulong: (et_j=2 \vee et_j=4 \Rightarrow geometry_j(ORDso_j,\ldots, ORDso_{j+1}\text{-}1)$ *follows* (v) *rules*$) \wedge (et_j=3 \vee et_j=5 \Rightarrow geometry_j(ORDso_j,\ldots,ORDso_{j+1}\text{-}1)$ *follows* (vi) *rules* $\wedge unique(geometry_j(ORDso_j,\ldots,ORDso_{j+1}\text{-}1)))$

$\wedge$

(viii) $(\forall\ gt \in GTYPE \cdot gt=2005 \Rightarrow \forall\ j,\ 1\leq j\leq |ELEM|DIV3,\ j\in ulong: et_j=1 \wedge ip_j=1 \wedge so_j=2*j\text{-}1 \wedge |ORD|=2*|ELEM|DIV3 \wedge unique(point(ORDso_j, ORD\ so_{j+1}\text{-}1))) \wedge$

(ix) $(\forall gt \in GTYPE \cdot gt=2006 \Rightarrow \forall j,\ 1\leq j\leq |ELEM|DIV3,\ j\in ulong: linestring_j(ORDso_j,\ldots, ORDso_{j+1}\text{-}1)$ *follows* (v) *rules* $\wedge unique(linestring_j\ (ORDso_j,\ldots, ORDso_{j+1}\text{-}1))) \wedge$

(x) $(\forall gt \in GTYPE \cdot gt=2007 \Rightarrow \forall\ i, j,\ 1\leq i, j\leq |ELEM|DIV3,\ i, j\in ulong: i\neq j \Rightarrow disjoint(polygon_i, polygon_j) \wedge polygon_j(ORDso_j,\ldots, ORDso_{j+1}\text{-}1)$ *follows* (vi) *rules* $\wedge unique(polygon_j(ORDso_j,\ldots, ORDso_{j+1}\text{-}1)))$

}

The constraints *(i)* to *(iii)* describe formally some usage considerations, while the rules from *(iv)* to *(x)* illustrate possible interrelations between the constituent types of the *SDO_GEOMETRY* object for each one of the geometries that can be represented by this object. More specifically, the *(iv)* rule depicts the two possible ways to define a point geometry and *(v)* exemplifies that in order to construct a valid linestring the size of the ordinates array should be at least four (namely two points) and the first point must not coincide with the last point, as this is the case that differentiates a simple polygon from a linestring. What is more, each pair of sequential points must be different and for each triplet of points, if these points are intended to describe two sequential linear segments then we require that there are no such co-linear overlapping segments, otherwise if these points describe just an arc-segment then we require that these points are not co-linear.

The *(vi)* set theory proposition describes the constraints that should stand in case the *SDO_GEOMETRY* object models a polygon geometry. Firstly ensures the validity of rectangular and circle geometries, which are special cases of a polygon. This is accomplished by not permitting parallelism between the segment that is formed by the lower left and upper right point that define a rectangular and the *xx'* or the *yy'* axis; and by forbidding co-linearity between the three points needed to define a circle.

For simple polygons the model requires that the first polygon-element described in the elements-info array must be the exterior boundary that will include one or more possible disjoint hole-polygons. The points that form the exterior boundary in the ordinates array must be specified in counter-clockwise order, while points composing hole-polygons must be specified in clockwise order. Furthermore, the linestring subelements that describe complex interpolations of the boundary of a polygon must meet (the end point of a linestring is the same with the starting point of the next linestring), must not intersect in their interior (a point other than an end point), must not touch (the end point lies in the interior of the other linestring) and the starting point of the first linestring must be equal with the end point of the last linestring. Finally, each of these linestring sub-elements must fulfil the constraints imposed for linestring geometries in *(v)*.

The rules described in *(vii)* impose that, for each distinct geometry object that is integrated in a heterogeneous collection, the corresponding constraints must stand depending on the kind of geometry. For example, if the collection has a linestring, then the *(v)* constraints must stand for this linestring. Similarly, propositions from *(viii)* to *(x)* require unique representation and existence of a geometry object inside a homogeneous collection and validity of each of them as this is implied by the rules that conform to its type. An additional rule that is enforced in the case of a multi-polygon is that either the exterior boundaries of the polygons composing the collection are disjoint or the exterior boundary of a polygon is inside a hole of another polygon.

12 Appendix C – Formal Definition of Spatio-Temporal Moving Types

In order to formally define in terms of set theory the moving types introduced in this paper we follow a down-top approach, meaning that we first describe the simpler data types and subsequently we define the more complex data types. In this section we just present the formal definitions while their linguistic explanations have been given in section 0.

MT = Π Moving_Point T ∪ Π Moving_LineString T ∪ Π Moving_Rectangle T ∪ Π Moving_Circle T ∪ Π Moving_Polygon T ∪ Π Moving_Collection T ∪ Π Moving_Object T.

First of all let us define the unit moving types, which are the basic building components of the spatio-temporal data types. The simpler of the unit moving types, the *Unit_Moving_Point*, upon which, is based the definition of all the others, needs for its construction two kind of objects, namely the *D_Period_Sec* and the *Unit_Function*. The D_Period_Sec is formally described in Appendix B as the *period⟨SECOND⟩* type. The *Unit_Function* is defined as follows:

Unit_Function $=_d$ *⟪a:double, b:double, c:double, descr:TypeOfFunction⟫*

where

Π TypeOfFunction T = { PLNML_1, PLNML_2, SQRT_PLNML_2, CONST }

As such,

Unit_Moving_Point $=_d$ *⟪p: period⟨SECOND⟩, x: Unit_Function, y: Unit_Function⟫*

Unit_Moving_Rectangle $=_d$ *{ ⟪ ll: Unit_Moving_Point, ur: Unit_Moving_Point ⟫ | equal (ll.p, ur.p) }*

Unit_Moving_Circle $=_d$ *{ ⟪ f: Unit_Moving_Point, s: Unit_Moving_Point, t: Unit_Moving_Point ⟫ | equal (f.p, s.p, t.p) }*

Unit_Moving_Segment $=_d$ *{ ⟪ b: Unit_Moving_Point, e: Unit_Moving_Point, m: Unit_Moving_Point, kind:TypeOfSegment ⟫ | (kind=SEG ⇒ equal (b.p, e.p)) ∧ (kind =ARC ⇒ equal (b.p, e.p, m.p)) }, where Π TypeOfSegment T = { SEG, ARC }*

Unit_Moving_Linestring $=_d$ *{ l: set⟨Unit_Moving_Segment⟩ | ∀ i, j ∈ ulong: i≠j ⇒ equal (*l_i*.b.p,* l_j*.e.p) }*

Unit_Moving_Polygon $=_d$ *{⟪ l: set⟨Unit_Moving_Segment⟩, hole:boolean ⟫ | ∀ i, j ∈ ulong: i≠j ⇒ equal (*l_i*.b.p,* l_j*.e.p) }*

Having defined all the unit-moving types we are now ready to formalize the description of our moving types:

Moving_Point $=_d$ *{ p: set⟨Unit_Moving_Point⟩ | ∀ i, j ∈ ulong, 1≤ i, j≤ |set⟨Unit_Moving_Point⟩|: j= i+1 ⇒* p_i*.p <* p_j*.p ∧ ¬overlaps(*p_i*.p,* p_j*.p) ∧ ∀ t ∈ double: inside(t,* p_i*.p) ⇒ at_instant(p, t) ∈* $SDO\_GEOMETRY_{SDO\_GTYPE=2001}$ */*point geometry*/}*

Moving_Rectangle $=_d$ *{ r: set⟨Unit_Moving_Rectangle⟩* $\mid \forall$ *i, j* $\in$ *ulong,* $1 \leq i, j \leq |$*set⟨Unit_Moving_Rectangle⟩*$|$*: j= i+1* $\Rightarrow r_i.ll.p < r_j.ur.p \wedge \neg$*overlaps(*$r_i.ll.p$*,* $r_j.ur.p$*)* $\wedge \forall$ *t* $\in$ *double: inside(t,* $r_i.ll.p$*)* $\Rightarrow$ *at_instant(r, t)* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2003 \wedge SDO\_ELEM\_INFO=(1,3,3)}$ */*rectangle geometry*/ }*

Moving_Circle $=_d$ *{ c: set⟨Unit_Moving_Circle⟩* $\mid \forall$ *i, j* $\in$ *ulong,* $1 \leq i, j \leq |$*set⟨Unit_Moving_Circle⟩*$|$*: j= i+1* $\Rightarrow c_i.f.p < c_j.s.p \wedge \neg$*overlaps(*$c_i.f.p$*,* $c_j.s.p$*)* $\wedge \forall$ *t* $\in$ *double: inside(t,* $c_i.f.p$*)* $\Rightarrow$ *at_instant(c, t)* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2003 \wedge SDO\_ELEM\_INFO=(1,3,4)}$ */*circle geometry*/ }*

Moving_LineString $=_d$ *{ line: set⟨Unit_Moving_LineString⟩* $\mid \forall$ *i, j* $\in$ *ulong,* $1 \leq i, j \leq |$*set⟨Unit_Moving_LineString⟩*$|$*: j= i+1* $\Rightarrow line_i.l_1.b.p < line_j.l_1.e.p \wedge \neg$*overlaps(*$line_i.l_1.b.p$*,* $line_j.l_1.e.p$*)* $\wedge \forall$ *t* $\in$ *double: inside(t,* $line_i.l_1.b.p$*)* $\Rightarrow$ *at_instant(line, t)* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2002}$ */*linestring geometry*/ }*

Moving_Polygon $=_d$ *{ pol: set⟨Unit_Moving_Polygon⟩* $\mid \forall$ *i, j* $\in$ *ulong,* $1 \leq i, j \leq |$*set⟨Unit_Moving_Polygon⟩*$|$*: j= i+1* $\Rightarrow pol_i.l_1.b.p < pol_j.l_1.e.p \wedge \neg$*overlaps(*$pol_i.l_1.b.p$*,* $pol_j.l_1.e.p$*)* $\wedge \forall$ *t* $\in$ *double: inside(t,* $pol_i.l_1.b.p$*)* $\Rightarrow$ *at_instant(pol, t)* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2003}$ */*polygon geometry*/ }*

In order to define the Moving_Collection and subsequently the Moving_Object data types, we first need to describe formally the multi object types for each one of the moving types:

Multi_Moving_Point $=_d$ *{ multi_mpoint: set⟨ Moving_Point⟩* $\mid \forall$ *i, j* $\in$ *ulong* $\wedge \forall$ *t* $\in$ *double: inside(t,* $multi\_mpoint_i.p_j.p$*)* $\Rightarrow \cup_i$ *(at_instant(*$multi\_mpoint_i$*, t))* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2005}$ */*multi-point geometry*/ }*

Multi_Moving_LineString $=_d$ *{ multi_mline: set⟨ Moving_LineString⟩* $\mid \forall$ *i, j* $\in$ *ulong* $\wedge \forall$ *t* $\in$ *double: inside(t,* $multi\_mline_i.line_j.l_1.b.p$*)* $\Rightarrow \cup_i$ *(at_instant(*$multi\_mline_i$*, t))* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2006}$ */*multi-linestring geometry*/ }*

Multi_Moving_Circle $=_d$ *{ multi_mcircle: set⟨ Moving_Circle⟩* $\mid \forall$ *i, j* $\in$ *ulong* $\wedge \forall$ *t* $\in$ *double: inside(t,* $multi\_mcircle_i.c_j.f.p$*)* $\Rightarrow \cup_i$ *(at_instant(*$multi\_mcircle_i$*, t))* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

Multi_Moving_Rectangle $=_d$ *{ multi_mrectangle: set⟨ Moving_Rectangle⟩* $\mid \forall$ *i, j* $\in$ *ulong* $\wedge \forall$ *t* $\in$ *double: inside(t,* $multi\_mrectangle_i.r_j.ll.p$*)* $\Rightarrow \cup_i$ *(at_instant(*$multi\_mrectangle_i$*, t))* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

Multi_Moving_Polygon $=_d$ *{ multi_mpolygon: set⟨ Moving_Polygon⟩* $\mid \forall$ *i, j* $\in$ *ulong* $\wedge \forall$ *t* $\in$ *double: inside(t,* $multi\_mpolygon_i.pol_j.l_1.b.p$*)* $\Rightarrow \cup_i$ *(at_instant(*$multi\_mpolygon_i$*, t))* $\in$ *SDO_GEOMETRY*$_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

As such, *Moving_Collection* $=_d$ *{ ⟪ multi_mpoint: Multi_Moving_Point,*
multi_mline: Multi_Moving_LineString,
multi_mcircle: Multi_Moving_Circle,
multi_mrectangle: Multi_Moving_Rectangle,
multi_mpolygon: Multi_Moving_Polygon ⟫ $\mid$

$\forall$ *i, j* $\in$ *ulong* $\wedge$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mpoint_i.p_j.p$*)* $\wedge$ *inside(t,* $multi\_mline_i.line_j.l_1.b.p$*)* $\wedge$ *inside(t,* $multi\_mcircle_i.c_j.f.p$*)* $\wedge$ *inside(t,* $multi\_mrectangle_i.r_j.ll.p$*)* $\wedge$ *inside(t,* $multi\_mpolygon_i.pol_j.l_1.b.p$*)*

$\Rightarrow$ *[* $(\cup_i$ *(at_instant(*$multi\_mpoint_i$*, t)))* $\cup(\cup_i$ *(at_instant(*$multi\_mline_i$*, t)))* $\cup(\cup_i$ *(at_instant(*$multi\_mcircle_i$*, t)))* $\cup(\cup_i$ *(at_instant(*$multi\_mrectangle_i$*, t)))* $\cup$ $(\cup_i$ *(at_instant(*$multi\_mpolygon_i$*, t)))* *]* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2004}$ */*collection geometry*/ }*

Moving_Object $=_d$ *{《 mobject: Moving_Object,*

mpoint: Moving_Point,

mline: Moving_LineString,

mcircle: Moving_Circle,

mrectangle: Moving_Rectangle,

mpolygon: Moving_Polygon,

mcolection: Moving_Collection,

geometry: SDO_GEOMETRY,

gtype: GeometryType,

optype: string,

arg1: ushort,

arg2: ushort,

input: Union_Input 》 | section<3.5> }》

where

Π GeometryType T = { MOBJECT, MPOINT, MLINE, MCIRCLE, MRECTANGLE, MPOLYGON, MCOLLECTION }

Union_Input $=_d$ *《mask: string, tolerance: double, distance: double》*

13 Appendix D – Description of HERMES-MDC's operations

13.1 Maintaining the Database Consistent

The two subsequent sections present how *HERMES-MDC* facilitates a user checking the construction data of two moving objects and as such maintaining the database in a consistent state:

13.1.1 Validation of a Moving LineString

In the following figure we demonstrate a series of visual transformations (virtual movements) of a moving linestring. The corresponding linguistic description of the figure as well as events raised by *HERMES-MDC* is given in Table 3. By this way we show special features, interesting and degenerated cases as well as rules and constraints that we impose in this type. HERMES-MDC identifies and reports such phenomena in order to maintain the consistency of the database. In the figure below the black solid lines represents snapshots of the moving objects. The various spatio-temporal transformations T_i that are of interest are depicted by the grey dashed arrows from some initial positions of the unit-moving points to some others (that are differently coloured).

Spatio-Temporal Transformations	**HERMES-MDC Events**
T_1: $u\_m\_p_2 \rightarrow (4, 4)$;	*Raises an application error because $u\_m\_p_2$, $u\_m\_p_3$ & $u\_m\_p_4$ that define an arc segment are becoming co-linear.*
T_2: $u\_m\_p_6 \rightarrow (7.5, 6.5)$;	*Raises an application error because $u\_m\_p_6$ is lying on an interior point of the segment defined by $u\_m\_p_4$ & $u\_m\_p_5$. In other words, segments ($u\_m\_p_4$, $u\_m\_p_5$) & ($u\_m\_p_5$, $u\_m\_p_6$) overlap.*
T_3: $u\_m\_p_6 \rightarrow (12, 8)$;	*Despite that $u\_m\_p_4$, $u\_m\_p_5$ & $u\_m\_p_6$ are becoming co-linear and as such two consequent linear segments could be replaced by only one, this is an acceptable case.*
T_4 & T_5: $u\_m\_p_6$ & $u\_m\_p_7 \rightarrow (10, 6)$;	*Raises an application error due to that $u\_m\_p_6$ & $u\_m\_p_7$ that define a line segment are becoming the same point and as such the segment is degenerated to a point.*
T_6: $u\_m\_p_7 \rightarrow (7.5, 6.5)$;	*Even though $u\_m\_p_7$, which is the last unit-moving point for the moving linestring, touches the interior of another segment, this is an acceptable case.*
T_7: $u\_m\_p_7 \rightarrow (4, 6)$;	*Despite the fact that $u\_m\_p_7$, which is the last unit-moving point for the moving linestring, crosses another segment, this is an acceptable case.*
T_8: $u\_m\_p_7 \rightarrow (14, 7)$; & $u\_m\_p_6 \rightarrow (15, 5)$;	*Raises an application error due to that the trajectories of $u\_m\_p_6$ & $u\_m\_p_7$, which are sequential unit-moving points, are intersecting, and this is a criterion that the corresponding unit-moving segment is rotating, something we do not accept.*
T_{10} & T_{11}: $u\_m\_p_1$ & $u\_m\_p_7 \rightarrow (7, 2)$;	*Raises an application error because $u\_m\_p_1$ & $u\_m\_p_7$, which are the first and last (respectively) unit-moving points for the moving linestring, are moving to the same point and as such they form a closed polygon.*

Table 3 Spatial Validation of a Moving LineString

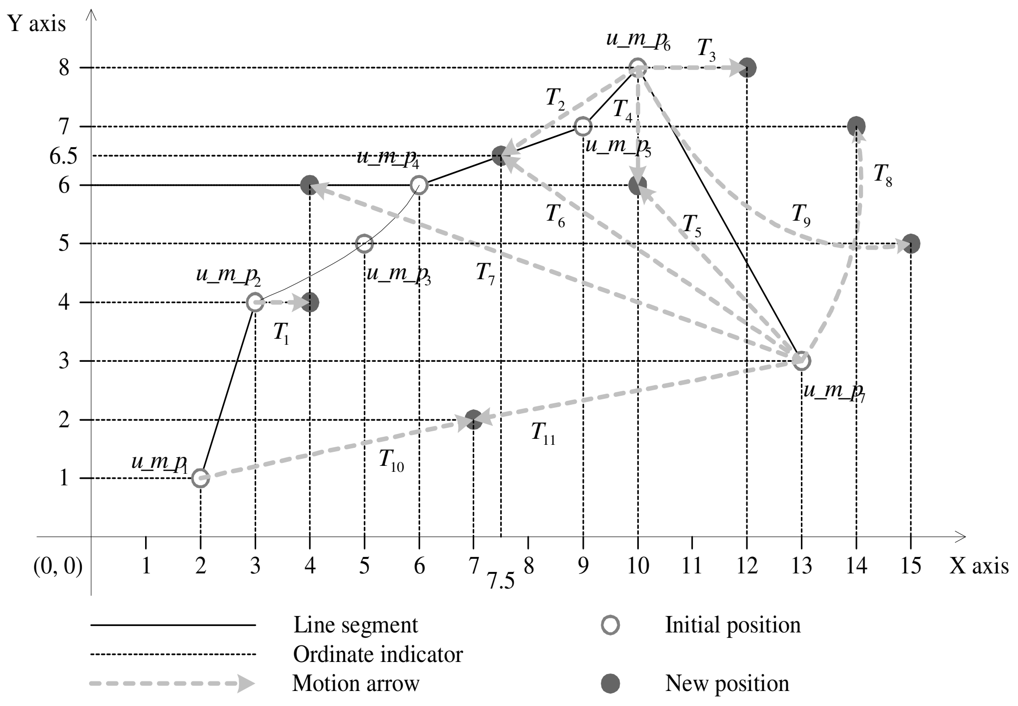


Figure 24 Spatial Validation of a Moving LineString

13.1.2 Validation of a Moving Polygon

As previously in the case of a Moving_LineString, here we follow exactly the same technique to demonstrate the *check_degeneracies* and *validate_geometry* operations that perform a spatial consistency check upon a Moving_Polygon.

Spatio-Temporal Transformations	**HERMES-MDC Events**
T_1: $u\_m\_p_1$ → (3, 1.5);	*Raises an application error due to that the moving polygon is not “closed”, meaning that the first unit-moving point is not the same with the last one.*
T_2: $u\_m\_p_2$ → (5, 5);	*Despite that $u\_m\_p_1$, $u\_m\_p_2$ & $u\_m\_p_3$ are becoming co-linear and as such two consequent linear segments ($u\_m\_p_1$, $u\_m\_p_2$) & ($u\_m\_p_2$, $u\_m\_p_3$) could be replaced by only one ($u\_m\_p_1$, $u\_m\_p_3$), this is an acceptable case.*
T_3: $u\_m\_p_4$ → (9.5, 7);	*Raises an application error because $u\_m\_p_3$, $u\_m\_p_4$ & $u\_m\_p_5$ that define an arc segment are becoming co-linear.*
T_4: $u\_m\_p_7$ → (13, 7);	*Raises an application error due to that $u\_m\_p_7$ crosses another segment. Self-intersection of unit-moving segments is forbidden in a moving polygon.*
T_5: $u\_m\_p_7$ → (13, 6);	*Raises an application error due to that $u\_m\_p_7$ touches the interior of another segment. Similar situation as the previous one.*
T_6: $u\_m\_p_7$ → $u\_m\_p_8$;	*Raises an application error due to that $u\_m\_p_7$ & $u\_m\_p_8$ that are components of an arc segment are becoming the same point and as such there are not three different unit-moving points to define the arc. The same case can be noticed when a moving segment is degenerated to a point.*
T_7: $u\_m\_p_9$ → (11/3, 4);	*Raises an application error because $u\_m\_p_9$ that is one of the vertices of a hole; touches the exterior boundary of the polygon.*
T_8: $u\_m\_p_9$ → (3, 6);	*Raises an application error because $u\_m\_p_9$ crosses the boundary of the polygon and as such the hole that belongs to, intersects with the exterior*

	polygon.
T_9: $u\_m\_p_{10}$ → (11, 4); *T_{10}: $u\_m\_p_{10}$ → $u\_m\_p_{15}$*	*Both transformations raise an application error because the first implies that two hole-polygons are intersecting, while the second that these interior polygons are touching.*
T_{11}: $u\_m\_p_{12}$ → (15, 6); *T_{12}: $u\_m\_p_{13}$ → (16, 6);* *T_{13}: $u\_m\_p_{14}$ → (15, 4);* *T_{14}: $u\_m\_p_{15}$ → (16, 3);*	*These transformations also raise an application error because they are transferring an interior hole-polygon outside the exterior boundary.*

Table 4 Spatial Validation of a Moving Polygon

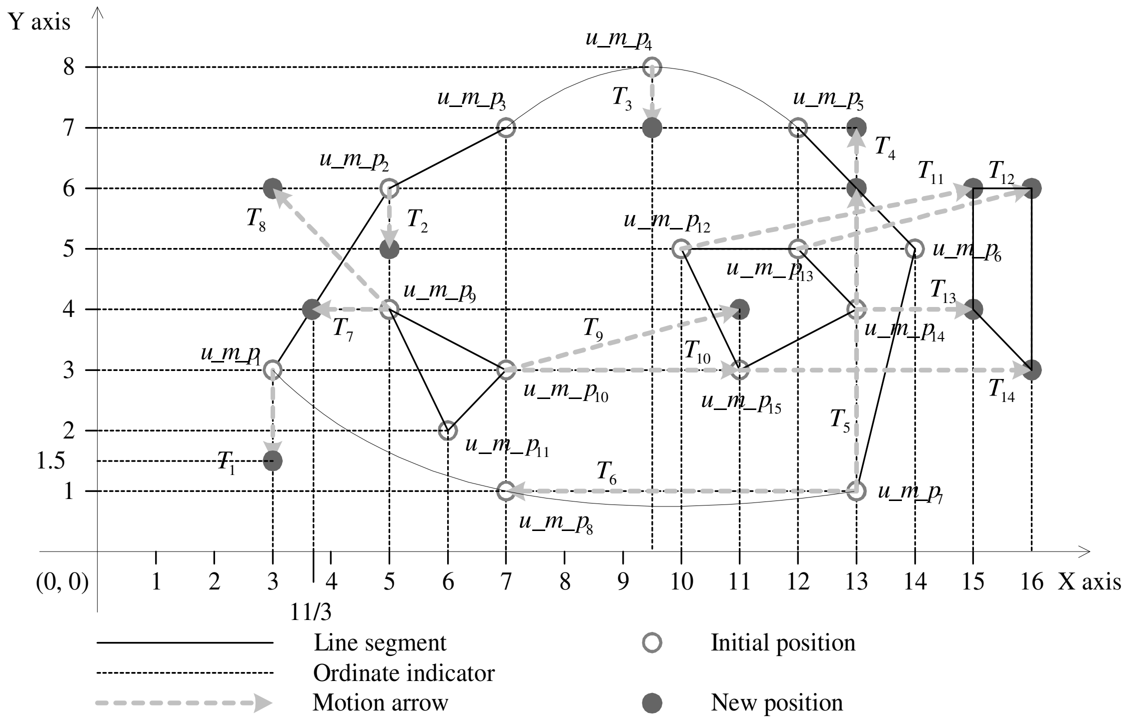


Figure 25 Spatial Validation of a Moving Polygon

13.2 Predicates Modeling Topological Relationships

The user can specify the kind of topological relationships that he requires to check via a *mask* parameter. The following *mask* relationships can be tested in HERMES-MDC:

- *ANYINTERACT* - Returns *TRUE* if the objects are not disjoint.
- *CONTAINS* - Returns *CONTAINS* if the argument moving object is entirely within the caller object and the object boundaries do not touch, at the given instance of time; otherwise, returns *FALSE.*
- *COVEREDBY* - Returns *COVEREDBY* if the parameter object is entirely within the caller object and the object boundaries touch at one or more points; otherwise, returns *FALSE.*
- *COVERS* - Returns *COVERS* if the argument object is entirely within the caller object and the boundaries touch in one or more places; otherwise, returns *FALSE.*

- *DISJOINT* - Returns *DISJOINT* if the objects have no common boundary or interior points; otherwise, returns *FALSE*.
- *EQUAL* - Returns *EQUAL* if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns *FALSE*.
- *INSIDE* - Returns *INSIDE* if the argument object is entirely within the caller object and the object boundaries do not touch; otherwise, returns *FALSE*.
- *OVERLAPBDYDISJOINT* - Returns *OVERLAPBDYDISJOINT* if the objects overlap, but their boundaries do not interact; otherwise, returns *FALSE*.
- *OVERLAPBDYINTERSECT* - Returns *OVERLAPBDYINTERSECT* if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns *FALSE*.
- *TOUCH* - Returns *TOUCH* if the two objects share a common boundary point, but no interior points; otherwise, returns *FALSE*.

Values for *mask* can be combined using a logical boolean operator. For example, *'INSIDE + TOUCH'* returns *'INSIDE + TOUCH'* or *'FALSE'* depending on the outcome of the test.

Generally, the *"relate"* function can return the following types of answers:

- If we pass a *mask* listing one or more relationships, the function returns the name of the relationship if it is true for the pair of geometries. If all of the relationships are false, the procedure returns *FALSE*.
- If we pass the *DETERMINE* keyword in *mask*, the function returns the one relationship keyword that best matches the geometries. *DETERMINE* can only be used when the *relate* predicate is in the SELECT clause of the SQL statement.
- If we pass the *ANYINTERACT* keyword in *mask*, the function returns *TRUE* if the two geometries are not disjoint.

13.3 Projection and Interaction to Temporal and/or Spatial Domain

The signatures of the object methods as these are defined for the Moving_Object type that HERMES-MDC provides for handling the projection and interaction of moving types to temporal and/or spatial domain are given in Section 13.4. Here we present the algorithms of an interesting as well as representative subset of these methods.

Below the reader can find an abstract description of the algorithm of the *at_instant* operation for a Moving_Object in the form of PL/SQL-like pseudo-code. Due to space limitations we present only the parts of the algorithm that have to do with Moving_Polygon and Moving_Collection objects as these are more interesting. Also, we do not present the algorithms but just the function calls for all the time-specific operations developed in *TAU TLL Data Cartridge*. The reader interested in these operations is referred to [Pel02].

```
FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Union_Output is

result Union_Output;
geom MDSYS.SDO_GEOMETRY;
geom1 MDSYS.SDO_GEOMETRY;
geom2 MDSYS.SDO_GEOMETRY;
BEGIN
    IF m_object.gtype IS NOT NULL THEN
      SWITCH (m_object.gtype)
      CASE 'MOBJECT':
          result :=  m_object.at_instant(tp);
      CASE 'MPOINT':
          geom := m_object.m_point.at_instant(tp);
      CASE 'MLINE':
          geom := m_object.m_line.at_instant(tp);
      CASE 'MCIRCLE':
          geom := m_object.m_circle.at_instant(tp);
      CASE 'MRECTANGLE':
          geom := m_object.m_rectangle.at_instant(tp);
      CASE 'MPOLYGON':
          geom := m_object.m_polygon.at_instant(tp);
      CASE 'MCOLLECTION':
          geom := m_object.m_collection.at_instant(tp);
      END SWITCH;
        result := Construct Union_Output from geom or result.geom;
    ELSE
      IF m_object.optype is unary THEN
          SWITCH (m_obj.arg1)
              CASE 1: result :=  m_object.at_instant(tp);
              CASE 2: geom := m_object.m_point.at_instant(tp);
              ...
              CASE 7: geom := m_object.m_collection.at_instant(tp);
              CASE 8: geom := geometry;
          END SWITCH;
          result := invoke_unary_operation(m_object.optype, geom1 or result.geom, m_object.input);
      ELSIF m_object.optype is binary THEN
          SWITCH (m_object.arg1)
              CASE 1: result := m_object.at_instant(tp);
              CASE 2: geom1  := m_object.m_point.at_instant(tp);
              ...
              CASE 7: geom1  := m_object.m_collection.at_instant(tp);
              CASE 8: geom1  := m_object.geometry;
          END SWITCH;
          SWITCH (m_object.arg2)
              CASE 1: result := m_object.at_instant(tp);
              CASE 2: geom2  := m_object.m_point.at_instant(tp);
              ...
              CASE 7: geom2  := m_object.m_collection.at_instant(tp);
              CASE 8: geom2  := m_object.geometry;
          END SWITCH;
          result := invoke_binary_operation(m_object.optype, geom1, geom2, m_object.input);
      ELSE
          raise_application_error('At_Instant operation is invalid for this kind of Moving_Object');
      END IF;
   END IF;
   return result;
END;
```

Figure 26 The *at_instant* algorithm for a Moving_Object

The pseudo-code of the *at_instant* operation for a Moving Polygon is given below:

```
FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Sdo_Geometry is

 t double;
 BEGIN
   IF check_periods_equality() <> TRUE THEN
       raise_application_error('Periods in at least one entry of the nested table of type
       Unit_Moving_Polygon are NOT equal');
   END IF;

   IF check_sorting() <> TRUE THEN
       raise_application_error(‘Periods in the nested table of type Unit_Moving_Polygon are NOT
       sorted');
   END IF;

   IF check_disjoint() <> TRUE THEN
       raise_application_error(‘Periods in the nested table of type Unit_Moving_Polygon are NOT
       disjoint');
   END IF;

   /* OPTIONAL -   IF check_meet() <> TRUE THEN
       raise_application_error(‘Periods in the nested table of type Unit_Moving_Polygon do NOT meet');
   END IF; */

   i := pol.FIRST;  -- get subscript of first unit moving polygon
   WHILE i IS NOT NULL LOOP
       contain_flag := pol(i).b.p.f_contains(pol(i).b.p, tp); --Check if tp is “inside” the period of pol(i)
       IF contain_flag = TRUE THEN
           t := tp.get_Abs_Date(); -- Map Timepoint object to real number (instant on time-line).
           result := merge_polygons(i, t);
           // The merge_polygons algorithm is given in Figure 28
           exit;
       END IF;

       i := pol.NEXT(i);  -- get subscript of next unit moving polygon
   END LOOP;

   IF result is not null THEN
       err_msg := VALIDATE_GEOMETRY(result);
       IF err_msg = 'TRUE' THEN
         return result;
       ELSE
         raise_application_error('Geometry validation failed'||err_msg);
       END IF;
   ELSE
       raise_application_error('The Timepoint is NOT contained in any of the Periods in the nested table
       of type Unit_Moving_Polygon');
   END IF;

 END;
```

Figure 27 The *at_instant* algorithm for a Moving_Polygon

The algorithm *merge_polygons* invoked in the *at_instant* method of a *Moving_Polygon* is given in Figure 28:

```
FUNCTION merge_polygons (i, integer, t double) return Sdo_Geometry is

BEGIN
 LOOP FOREVER
   j := pol(i).l.FIRST;  -- get subscript of first moving segment
   WHILE j IS NOT NULL LOOP
     Interpolate the Unit_Moving_Points of current moving segment at instance t
     Add the description of the linestring element in the Elem_Info_Array
     Add the corresponding co-ordinates in the Ordinates_Array
     Check for degenerecies in the linestring element
     j := pol(i).l.NEXT(j);  -- get subscript of next moving segment
   END LOOP;

   result := Construct the polygon formed by the Elem_Info_Array & Ordinates_Array;
   Check for degenerecies in the polygon geometry;

   IF i <> pol.LAST THEN
     i := i + 1;
     Initialize flags & local variables;
    Extend Elem_Info_Array & Ordinates_Array for probable addition of holes;

     IF pol(i).hole = FALSE THEN
         return result;
     END IF;
   ELSE
     return result;
   END IF;
  END LOOP;
END;
```

Figure 28 The *merge_polygons* algorithm

The pseudo-code of the *at_instant* operation for a *Moving_Collection* is given below:

```
FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Sdo_Geometry is

BEGIN
     FOR each m_collection.multi_moving_type in {multi_mpoint, multi_mline, multi_mcircle,
     multi_mrectangle, multi_mpolygon}
       i := m_collection.multi_moving_type.FIRST;  -- get subscript of first element
      WHILE i IS NOT NULL LOOP
         mtype := m_collection.multi_moving_type(i);
         IF i = 1 AND result IS NULL THEN
           current_homogeneous_collection := mtype.at_instant(tp);
         ELSE
           current_geom := mtype.at_instant(tp);
           current_homogeneous_collection:=ADD(current_homogeneous_collection, current_geom);
         END IF;

          i := m_collection.multi_moving_type.NEXT(i);  -- get subscript of next element
       END LOOP;

       heterogeneous_collection:=ADD(heterogeneous_collection, current_homogeneous_collection);
     END FOR;

    return heterogeneous_collection;
```

```
        END;
```

Figure 29 The *at_instant* algorithm for a Moving_Collection

Figure 30 depicts the algorithm of the *at_period* operation for the case of a *Moving_LineString* object.

```
FUNCTION at_period(p TAU_TLL.D_Period_Sec) return Moving_LineString is
new_line set<Unit_Moving_LineString>;
new_p TAU_TLL.D_Period_Sec;
BEGIN
    i := line.FIRST;  -- get subscript of first Unit_Moving_LineString
    WHILE i IS NOT NULL LOOP
      /* Check if period that characterizes the current Unit
      Moving LineString overlaps with the argument period */
      overlaps_flag := line(i).b.p.f_overlaps(line(i).b.p, p); --Check if  p "overlaps" the period of line(i)

      /* If YES take the period formed as the intersection of the two
      overlapped periods and update every Unit_Moving_Point */
      IF overlaps_flag = TRUE THEN
        new_p := line(i)b.p.intersects(line(i).b.p, p);

        j := line(i).l.FIRST;  -- get subscript of first element
        WHILE j IS NOT NULL LOOP
          line(i).l(j).b.p := new_p;
          line(i).l(j).e.p := new_p;
          line(i).l(j).m.p := new_p;

          j := line(i).l.NEXT(j);
        END LOOP;

        new_line(i) := line(i);
      END IF;

      i := line.NEXT(i);  -- get subscript of next Unit_Moving_LineString
    END LOOP;

    return  Moving_LineString(new_line);
END;
```

Figure 30 The *at_period* algorithm for a Moving_LineString

In Figure 31, we provide the reader with the pseudo-code of the *at_temp_element* operation for the case of a *Moving_Point* object, where it is obvious the different strategy of restricting the temporal domain with a temporal element, rather than with a period.

```
FUNCTION at_temp_element(te TAU_TLL.D_Temp_Element_Sec) return Moving_Point is

new_point set<Unit_Moving_Point>;
intersection_te TAU_TLL.D_Temp_Element_Sec;
new_period TAU_TLL.D_Period_Sec;
BEGIN
      new_point := p;

      /* First find the temporal element object that is the intersection of the argument
      temporal element with the temporal element returned by f_temp_element function */
      intersection_te := intersection(f_temp_element(), te);

      /* For each period <new_period> composing the previous resulted temporal element,
      update those periods of the Unit Moving Points that "contain" the <new_period>. */
      k := intersection_te.te.FIRST; -- get the subscript of first period of the temporal element
      WHILE k IS NOT NULL LOOP
        new_period := intersection_te.te(k);

        i := new_point(i).FIRST;
        WHILE i IS NOT NULL LOOP
          contain_flag := new_point(i).p.f_contains(new_point(i).p, new_period);
          IF contain_flag = 1 THEN
            new_point(i).p := new_period;
          END IF;

        i := new_point.NEXT(i);
        END LOOP;

        k := intersection_te.te.NEXT(k); -- get the subscript of next period of the temporal element
      END LOOP;

      return Moving_Point(new_point);
END;
```

Figure 31 The *at_temp_element* algorithm for a Moving_Point

In Figure 32 we provide the reader with the pseudo-code of the *f_traversed* operation for the case of a Moving_LineString object. We should mention that the current implementation supports only time-changing geometries whose vertices move linearly. What is more, the soundness of the algorithm presumes that during the period associated with the linear functions describing the motion of the vertices, rotation of the segments is forbidden by a condition of the model.

```
FUNCTION f_traversed return MDSYS.SDO_GEOMETRY is

result, prev_result, line_1, line_2 MDSYS.SDO_GEOMETRY;
tp_curr TAU_TLL.D_Timepoint_Sec;
BEGIN
    i := line.FIRST; -- get subscript of first unit moving linestring;
    WHILE i IS NOT NULL LOOP
      tp_curr := line(i).l(1).b.p.b; -- get the first instant of the period of the current unit moving linestring;
      line_1 := at_instant(tp_curr); -- Project the Moving LineString at the spatial domain at this time point;
      -- Access the Elem_Info_Array & Ordinates_Array of line_1;
      tp_curr := f_decr( line(i).l(1).b.p.e );  -- get the last instant of the period of the current unit moving
      linestring;
      line_2 := at_instant(tp_curr); -- Project the Moving LineString at the spatial domain at this time point;
      -- Access the Elem_Info_Array & an Ordinates_Array of line_2;
      -- Initialize an Elem_Info_Array & an Ordinates_Array object for constructing the traversed polygon;
      -- Depending on the type of the projected linestrings (1 & 2) construct an Elem_Info_Array that will
      represent a polygon geometry with elements the union of the elements of the Elem_Info_Arrays of line_1
      & line_2;
      -- Similarly, construct an Ordinates_Array that will represent a polygon geometry, whose boundary will
      be composed by the two projected lines connected at their end points by linear segments;
      For example...
      IF line_1 & line_2 are linearly interpolated THEN
        -- Construct Elem_Info_Array for a linerly interpolated polygon;
        -- Extend the Ordinates_Array as the size of the Ordinates_Array of line_1 and transfer all the ordinates
          from the second to the first array;
        ordinates.EXTEND(ordinates_1.LAST);
        WHILE ordinates_offset_1 IS NOT NULL LOOP
          ordinates(ordinates_offset_1) := ordinates_1(ordinates_offset_1);
          ordinates_offset_1 := ordinates_1.NEXT(ordinates_offset_1);
        END LOOP;

        -- Extend the Ordinates_Array as the size of the Ordinates_Array of line_2 and transfer all the ordinates
        from the second to the end of the first array;
        ordinates.EXTEND(ordinates_2.LAST);
        WHILE ordinates_offset_2 IS NOT NULL LOOP
          ordinates(ordinates_1.LAST + ordinates_offset_2) := ordinates_2(ordinates_offset_2);
          ordinates_offset_2 := ordinates_2.NEXT(ordinates_offset_2);
        END LOOP;

        ordinates.EXTEND(2); -- Connect first point to last point to form a polygon
        ordinates(ordinates_1.LAST + ordinates_2.LAST + 1) := ordinates_1(1);
        ordinates(ordinates_1.LAST + ordinates_2.LAST + 2) := ordinates_1(2);

        result := Construct the traversed polygon formed by the Elem_Info_Array & Ordinates_Array;
      ELSIF line_1 & line_2 are arc-interpolated OR are compound linestrings THEN
        Similarly...
      END IF;

      -- The final traversed polygon is the union of the traversed areas at all time periods for which the Moving
      LineString is defined;
      IF i <> line.FIRST THEN
        result := UNION(prev_result, result);
      END IF;

      prev_result := result;
      i := line.NEXT(i); -- get subscript of last unit moving linestring
    END LOOP;
    return result;
END;
```

Figure 32 The *f_traversed* algorithm for a Moving_LineString

13.4 Signatures of operations

The signatures of the object methods described throughout the paper are given below:

Maintaining the Database Consistent

- *MEMBER FUNCTION check_periods_equality return boolean*
- *MEMBER FUNCTION check_sorting return boolean*
- *MEMBER FUNCTION check_disjoint return boolean*
- *MEMBER FUNCTION check_meet return boolean*
- *MEMBER FUNCTION check_degeneracies (tp TAU_TLL.D_Timepoint_Sec) return boolean*
- *MEMBER FUNCTION validate_geometry (tp TAU_TLL.D_Timepoint_Sec, err_msg OUT Varchar2) return Varchar*

Predicates Modeling Topological and Distance Relationships

- *MEMBER FUNCTION f_within_distance (distance NUMBER, m_polygon REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return boolean*
- *MEMBER FUNCTION f_within_distance (distance NUMBER, m_polygon REF Moving_Polygon, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_relate (mask Varchar2, m_polygon REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return Varchar2*
- *MEMBER FUNCTION f_relate (mask Varchar2, m_polygon REF Moving_Polygon, tolerance NUMBER) return Moving_Object*

Projection and Interaction to Temporal and/or Spatial Domain

- *MEMBER FUNCTION unit_type (tp TAU_TLL.D_Timepoint_Sec) return Unit_Moving_Point*
- *MEMBER FUNCTION at_instant (tp TAU_TLL.D_Timepoint_Sec) return Union_Output*
- *MEMBER FUNCTION at_period (p TAU_TLL.D_Period_Sec) return Moving_Object*
- *MEMBER FUNCTION f_temp_element return TAU_TLL.D_Temp_Element_Sec*
- *MEMBER FUNCTION at_temp_element (te TAU_TLL.D_Temp_Element_Sec) return Moving_Object*
- *MEMBER FUNCTION f_buffer (distance NUMBER, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_buffer (distance NUMBER, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_centroid (tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_centroid (tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_convexhull (tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_convexhull (tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_pointonsurface (tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_pointonsurface (tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_initial return Union_Output*
- *MEMBER FUNCTION f_final return Union_Output*

- *MEMBER FUNCTION f_traversed return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_trajectory return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_locations return MDSYS.SDO_GEOMETRY*

Numeric operations

- *MEMBER FUNCTION f_area (tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_area (tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_length (tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_length (tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_num_of_components return Varchar2*
- *MEMBER FUNCTION f_num_of_components(mtype Varchar2) return pls_integer*

Distance and Direction

- *MEMBER FUNCTION f_distance (m_poly_ref REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_distance (m_poly_ref REF Moving_Polygon, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_direction (m_point_ref REF Moving_Point, tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_direction (m_point_ref REF Moving_Point) return Moving_Object*

Set Relationships

- *MEMBER FUNCTION f_intersection (m_poly_ref REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_intersection (m_poly_ref REF Moving_Polygon, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_union (m_poly_ref REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_union (m_poly_ref REF Moving_Polygon, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_difference (m_poly_ref REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_difference (m_poly_ref REF Moving_Polygon, tolerance NUMBER) return Moving_Object*
- *MEMBER FUNCTION f_xor (m_poly_ref REF Moving_Polygon, tolerance NUMBER, tp TAU_TLL.D_Timepoint_Sec) return MDSYS.SDO_GEOMETRY*
- *MEMBER FUNCTION f_xor (m_poly_ref REF Moving_Polygon, tolerance NUMBER) return Moving_Object*

Rate of Change

- *MEMBER FUNCTION f_speed(tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_speed return Moving_Object*
- *MEMBER FUNCTION f_turn(tp TAU_TLL.D_Timepoint_Sec) return number*
- *MEMBER FUNCTION f_turn return Moving_Object*
- *MEMBER FUNCTION f_velocity (tp TAU_TLL.D_Timepoint_Sec) return SDO_GEOMETRY*
- *MEMBER FUNCTION f_velocity return Moving_Point*