Student project

Implementation of algebra for representing and querying moving objects in object-relational database

Student: Remigijus Staniulis
The project leader: prof. Tore Risch

Uppsala

2002

Index

The task – system vision

Nowadays the demand of specialized databases exists. One of them is querying spatial and temporal (moving) objects. Güting et al [1] have proposed an algebra suitable for querying mobile and spatial objects in object oriented databases. The task of this project is to analyze how Güting algebra could be implemented in AMOS II [2] – an object relational DBMS, and to establish the architectural base for further development.

The principles of Güting algebra are firstly to define basic data types and operations, and then to use the algebra operators to extend them to moving data types and operations. It should be mentioned that basic data types are chosen in a way to make software implementation possible.

AMOS II is an object relational DBMS. Its main features are inheritance, overloading, and late binding support. Late binding and overloading are especially important to make Güting algebra implementation possible. Also, AMOS II functions may be defined as foreign functions and implemented outside AMOS II, i.e. in Java, C, or Lisp languages. This helps to implement more complex functions.

The first part of this paper deals mainly with analysis. That is, firstly the analysis of requirements is presented and the architectural background (languages, system, etc.) is chosen. For establishing the architectural base, significant Güting algebra parts are identified and analyzed in detail.

The second part explains the ideas of the architecture. Most of them are presented as UML diagrams and explained further. They are implemented as well. See Appendix B for included files.

The third part contains test cases descriptions. They are needed to test the architecture. They may be used for regression tests.

Finally, in the forth part, implementation is compared to Güting algebra and then some further work is suggested.

I. Requirements analysis

Analyze the requirements - defining uncertain objects

Some kinds of objects were left undefined in [1]. That is the result of abstracting and making more creative space for implementation. For these object types and their containers some further requirements are needed and postponed to the design and implementation phases. Here we try to identify them.

Object	Point
Requirements	A point in a plane. No special requirements.
Container	Points
Requirements for container	Definition 2: Points is a set of points, therefore, it shouldn't contain the same two points.

Object	Curve
Requirements	Definition 3 from [1]: the curve should be continuos**,** not self-intersecting, and have end and start, the ends of the curve may meet. Definition 5: an intersection of any two curves should produce a finite number of isolated points.
Suggested implementing object and comments	A straight line with a start point and end point. This definition fulfills object requirements.
Container	Line
Requirements for container	Line is a finite set of curves. Definition 7: curves can't intersect, they can only meet at the ends, and at least three ends should meet in one point. This requirement is being reduced to: (i) curves can't intersect, they can only meet at the ends, (ii) at least three ends should meet in one point, or there can meet two curves, but without possibility to join them. This requirement also serves for line being a unique collection of curves, that is a "canonical" representation of line. This requirement change is risky, so it is **added to risks list** and has a priority in implementation and testing. The benefits are much simpler implementation of the curve object.

Object	Face
Requirements	Requirement (page 10 in [1]): faces are regular closed, not empty subsets of plane.
Suggested implementing object and comments	Filled rectangle. Boundaries are parallel to X and Y-axis. Given bottom left point and top right point. This definition fulfills object requirements.
Container	Region
Requirements for container	Region is a finite set of faces. Definition 10: (i) any two faces are disjoint except for finite many "touching points" at the boundary, (ii) boundary should be possible to express as Line object (fulfilled by implementing object definition).

Object	Instant
Requirements	To be linear and continuos.
Suggested	Real numbers. Dates and normal time may be derived later, either changing

implementing object and comments	real into timestamp structure and overloading functions, or adding a few operations to convert real to date and back.

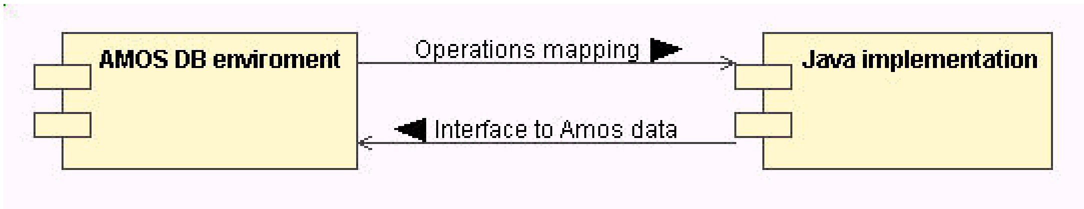
Object	Trajectory(a) – added for implementation purposes
Requirements	To be continuos.
Suggested implementing object and comments	Represents a continuous trajectory of (a). A list of pairs: <instant, a>, that is a list of intime(a) objects. Between adjacent intime(a) objects, (a) is considered as moving in linear and continuos manner. This definition fulfills object requirements.
Container	Moving(a)
Requirements for container	A finite set of trajectory (a). Definition 13: Consists of a finite number of continuo components. That is moving(a) projection into (a) has a finite number of components (intervals, regions, etc). For every instant object moving(a) is either undefined or has only one corresponding (a) (to be like a function).

Object	Interval(a) – added for implementation purposes
Requirements	Definition 15: an interval, boundaries may be open or closed.
Suggested implementing object and comments	Data structure with min and max values. Boolean values: left open, right open.
Container	Range(a)
Requirements for container	Range is a finite set of intervals. Definition 15 and 16: intervals are disjoint and nonadjacent.

Architectural background

Objects described in [1] should be represented in the object-relational database AMOS II. However, it is suggested that most of implementation and inner classes should be implemented in Java. That helps to handle the complexity of functions. Java was chosen as AMOS II has pretty good support (binding and interfaces) for Java code.

Functions mapping from AMOS II to Java (foreign functions) should be used as described in [2]. Java side code uses callin.* and callout.* packages to access and modify the data in AMOS II DBMS. These packages may be found in javaAMOS.jar file.



Identifying architecturally significant use cases

The use cases below are taken from Güting algebra description [1]. They try to cover most interesting (important) and architecturally risky parts of Güting algebra implementation.

Use case	**Manipulating Güting algebra objects.** Creating, adding, deleting of objects. All the time the requirements should hold as described in analysis part. Establishing some type control, i.e., the trajectory(real) can not be added to moving(point).
Detailed use case for architectural base	Full support of Point and Points as described in requirements. Other containers should be implemented just to manipulate items without respect to requirements. Establishing type control for Real, Points and Region.
What should be implemented and tested	Data structures in AMOS II (including type control). Java interfaces to AMOS II data structures. Data support functions (add, remove, get) for Points in AMOS II and JAVA.

Use case	**Calling not lifted Güting algebra operations.**
Detailed use case for architectural base	Operation “intersection” was chosen. It has several types of signatures, but only two of them are taken for implementation purposes – just to show how to resolve different signatures. For architectural base two concrete signatures are chosen: Intersection(Points, Region); Intersection(Region, Points);
What should be implemented and tested	Operations and AMOS II data types resolving in Java AMOS II functions mapping to Java functions. Implementation of intersection(Points, Region) and intersection(Region, Points) in Java.

Use case	**Calling lifted Güting algebra operations.** Lifting should be fully automated to avoid writing pretty similar code.
Detailed use case for architectural	Operations intersection(Region, Points) and intersection(Points, Region) should be lifted to: Intersection(MPoints, Region);

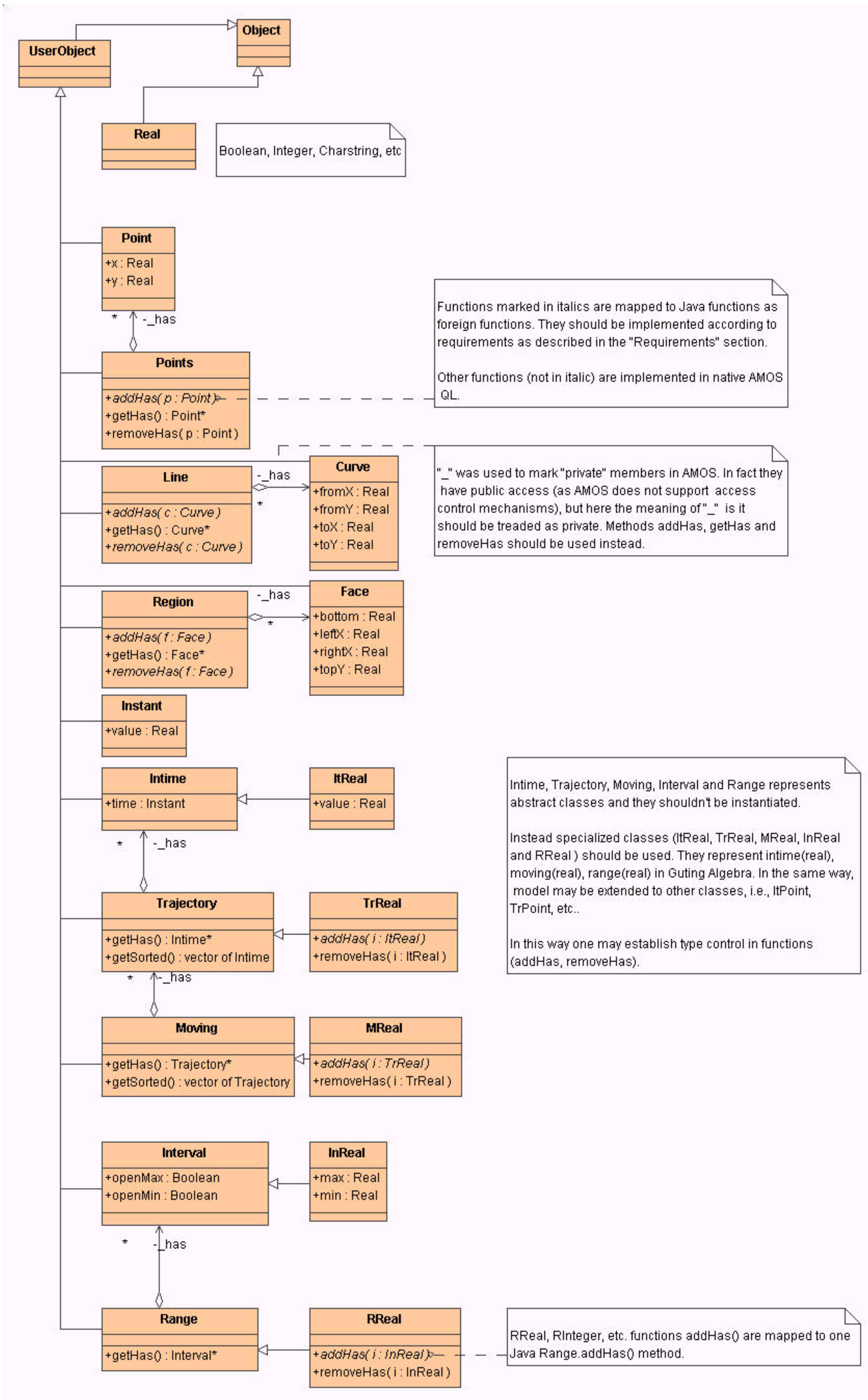
base	Intersection(Points, MRegion); Intersection(MPoints, MRegion); Intersection(Region, MPoints); Intersection(MRegion, Points); Intersection(MRegion, MPoints);
What should be implemented and tested	Automated lifting not depending on signature or data types.

The implemented architecture should be these detailed use cases driven. Other use cases should be not so difficult to implement. They are discussed and explained in more detail in the part IV.

II. System design and implementation

AMOS II data structures

The diagram below represents data structures and data support functions in AMOS II:



UserObject
Object
Real
Boolean, Integer, Charstring, etc
Point
+x : Real
+y : Real
* -_has
Points
+addHas(p : Point)
+getHas() : Point*
+removeHas(p : Point)
Functions marked in italics are mapped to Java functions as foreign functions. They should be implemented according to requirements as described in the "Requirements" section.
Other functions (not in italic) are implemented in native AMOS QL.
Line
+addHas(c : Curve)
+getHas() : Curve*
+removeHas(c : Curve)
-_has *
Curve
+fromX : Real
+fromY : Real
+toX : Real
+toY : Real
"_" was used to mark "private" members in AMOS. In fact they have public access (as AMOS does not support access control mechanisms), but here the meaning of "_" is it should be treaded as private. Methods addHas, getHas and removeHas should be used instead.
Region
+addHas(f : Face)
+getHas() : Face*
+removeHas(f : Face)
-_has *
Face
+bottom : Real
+leftX : Real
+rightX : Real
+topY : Real
Instant
+value : Real
Intime
+time : Instant
ItReal
+value : Real
* -_has
Intime, Trajectory, Moving, Interval and Range represents abstract classes and they shouldn't be instantiated.
Instead specialized classes (ItReal, TrReal, MReal, InReal and RReal) should be used. They represent intime(real), moving(real), range(real) in Guting Algebra. In the same way, model may be extended to other classes, i.e., ItPoint, TrPoint, etc..
In this way one may establish type control in functions (addHas, removeHas).
Trajectory
+getHas() : Intime*
+getSorted() : vector of Intime
TrReal
+addHas(i : ItReal)
+removeHas(i : ItReal)
* -_has
Moving
+getHas() : Trajectory*
+getSorted() : vector of Trajectory
MReal
+addHas(i : TrReal)
+removeHas(i : TrReal)
Interval
+openMax : Boolean
+openMin : Boolean
InReal
+max : Real
+min : Real
* -_has
Range
+getHas() : Interval*
RReal
+addHas(i : InReal)
+removeHas(i : InReal)
RReal, RInteger, etc. functions addHas() are mapped to one Java Range.addHas() method.

The displayed attributes corresponds to stored functions ("-> smth. as stored") in AMOS QL. Some of them are marked (and results) with multiplicity specified as "*", what means "bag of" in AMOS QL.

Functions marked in italic are mapped to Java functions as foreign functions. They should be implemented according to requirements as described in the "Requirements" section.

Other functions (not in italic) are implemented in native AMOS QL.
"_" was used to mark "private" members in AMOS II. In fact they have public access (as AMOS II does not support access control mechanisms), but here the meaning of "_" is that this member should be treaded as private. Methods addHas, getHas and removeHas should be used instead. These methods should cover requirements presented in part I.

Intime, Trajectory, Moving, Interval and Range represents abstract classes and they shouldn't be instantiated. Instead specialized classes (ItReal, TrReal, MReal, InReal and RReal) should be used. They represent intime(real), moving(real), range(real) in Güting Algebra. In the same way, model may be extended to other classes, i.e., ItPoint, TrPoint, etc. Each specialized data type has its own defined addHas() and removeHas() methods, for example, MReal.addhas(TrReal). **In this way one may establish type control in functions (addHas, removeHas).**

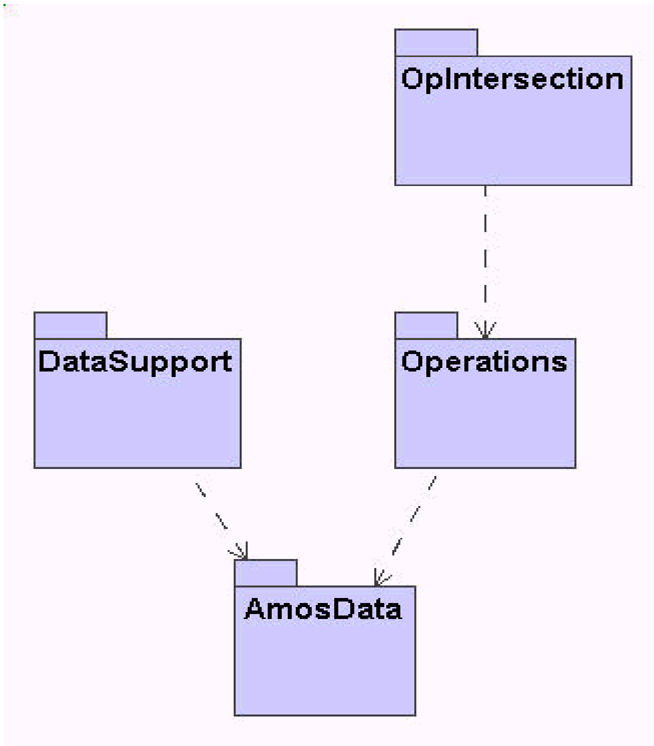
In the implementation all these data types and ItPoints, TrPoints, MPoints, ItRegion, TrRegion and MRegion are defined. Range type wasn't defined for Points and Region, because they are not eligible for it (see [1] Table II).

RReal, RInteger, etc. functions addHas() are mapped to one Java Range.addHas() method. That simplifies the implementation in Java.

The method Trajectory.getSorted() returns the sorted vector of Intime objects. They are sorted according to Intime.Time value. Moving.getSorted() returns the sorted vector of Trajectory objects. They are sorted according to the lowest Intime.Time value in each trajectory. That is both these functions should help to follow mobile objects by the time value.

Java implementation packages layout

Bellow the Java packages layout is presented:



Package AMOSData provides the interface to the AMOS II data structures. Also, contains inner implementation of data support functions (addhas, removehas).

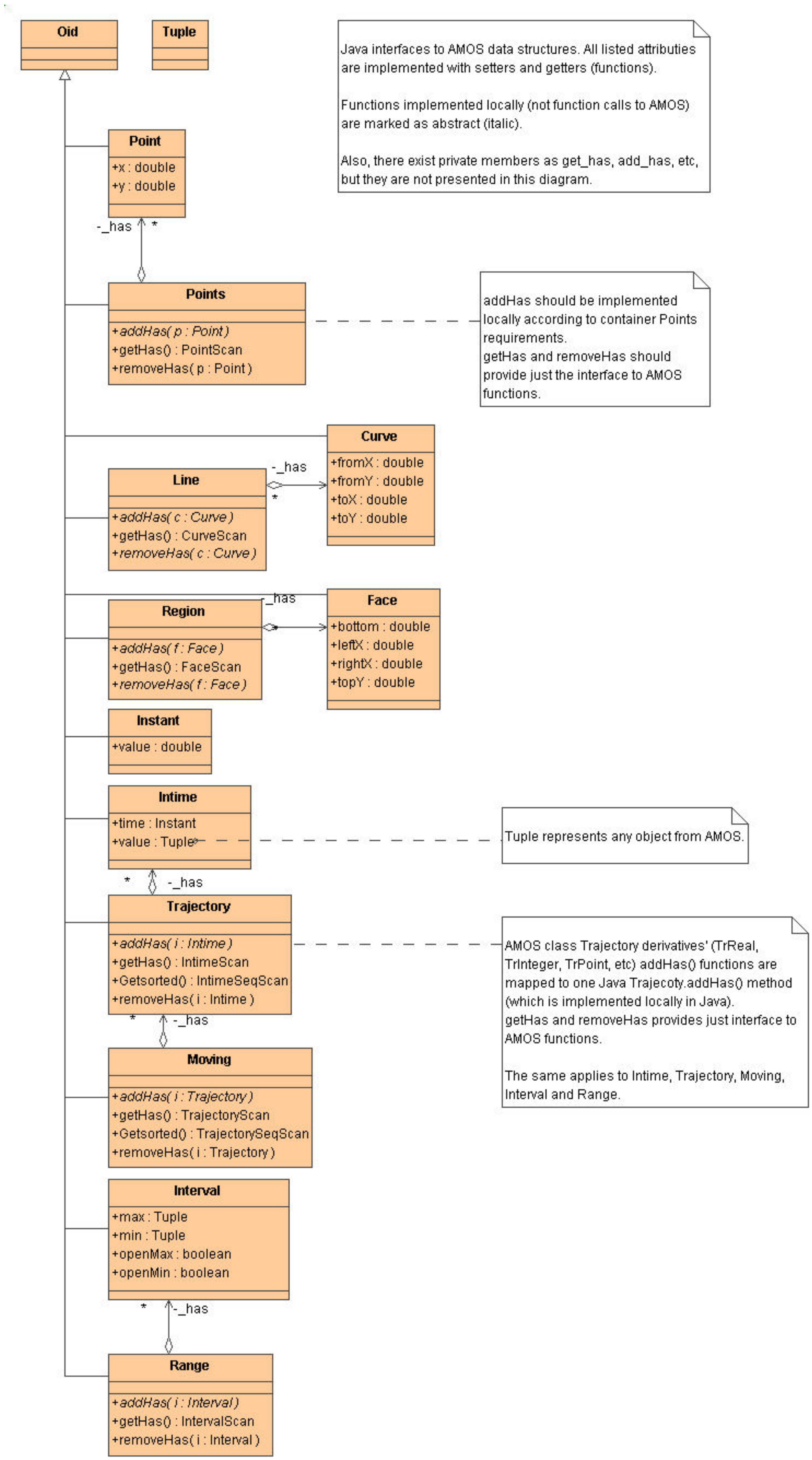
DataSupport provides the mapping of data support functions from AMOS II to Java. That is, foreign functions addhas, removehas should be mapped to the package DataSupport, which is responsible for resolving data types and calling appropriate methods.

Package Operations contains generic mechanism for implementation of Güting algebra operations. That is, resolving signatures and data types, automated lifting.

OperIntersection is an implemented Güting algebra operation that relies on Operations package. It contains implemented only intersection (Region, Points) and intersection (Points, Region) operations. That is enough to show how the further implementation could be done. These operations are automatically lifted to arguments of type moving in the Operations package. In the same way other operations (i.e., onborder, deftime, etc.) packages may be created.

AMOSData package

This is the basic package that provides the Java interface to data structures and functions in AMOS.


Oid
Tuple
Java interfaces to AMOS data structures. All listed attributies are implemented with setters and getters (functions).
Functions implemented locally (not function calls to AMOS) are marked as abstract (italic).
Also, there exist private members as get_has, add_has, etc, but they are not presented in this diagram.
Point
+x : double
+y : double
-_has
*
Points
+addHas(p : Point)
+getHas() : PointScan
+removeHas(p : Point)
addHas should be implemented locally according to container Points requirements.
getHas and removeHas should provide just the interface to AMOS functions.
Curve
+fromX : double
+fromY : double
+toX : double
+toY : double
Line
+addHas(c : Curve)
+getHas() : CurveScan
+removeHas(c : Curve)
Face
+bottom : double
+leftX : double
+rightX : double
+topY : double
Region
+addHas(f : Face)
+getHas() : FaceScan
+removeHas(f : Face)
Instant
+value : double
Intime
+time : Instant
+value : Tuple
Tuple represents any object from AMOS.
Trajectory
+addHas(i : Intime)
+getHas() : IntimeScan
+Getsorted() : IntimeSeqScan
+removeHas(i : Intime)
AMOS class Trajectory derivatives' (TrReal, TrInteger, TrPoint, etc) addHas() functions are mapped to one Java Trajecoty.addHas() method (which is implemented locally in Java).
getHas and removeHas provides just interface to AMOS functions.
The same applies to Intime, Trajectory, Moving, Interval and Range.
Moving
+addHas(i : Trajectory)
+getHas() : TrajectoryScan
+Getsorted() : TrajectorySeqScan
+removeHas(i : Trajectory)
Interval
+max : Tuple
+min : Tuple
+openMax : boolean
+openMin : boolean
Range
+addHas(i : Interval)
+getHas() : IntervalScan
+removeHas(i : Interval)

Object proxy Oid and any data container Tuple are the classes of the callin package from the javaAMOS.jar file. Other classes are implemented in the AMOSData package.

All listed attributes (_has, fromX, fromY, etc.) are implemented with setters and getters (that is functions), but they are not presented in the diagram. Functions implemented locally (not function calls to AMOS II) are marked in italic.

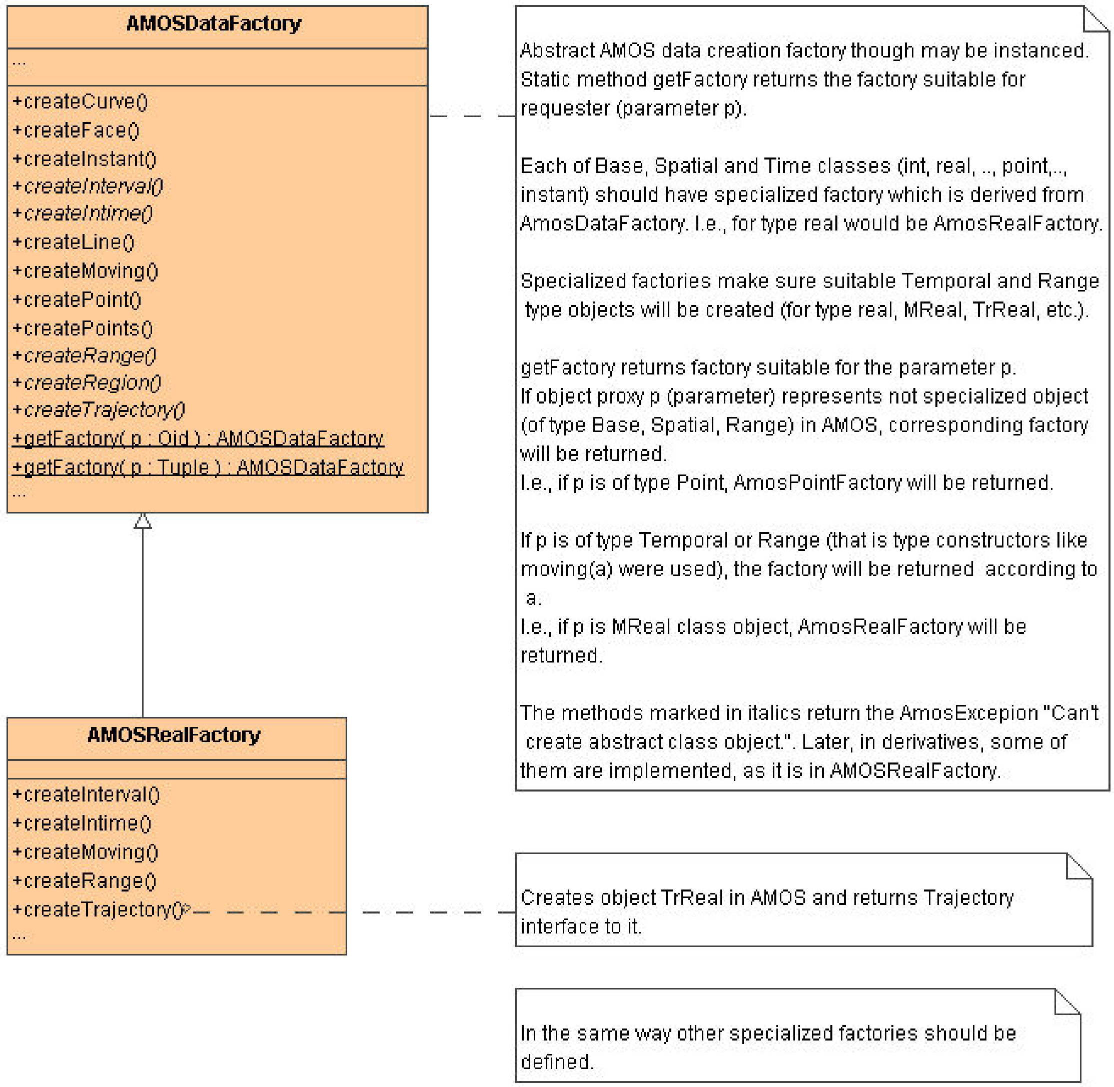
For example, Points.addHas should be implemented locally according to the container Points requirements. Points.getHas and Points.removeHas should provide just the interface to AMOS II functions (because they are not so complex and may be implemented in native AMOS QL).

AMOS II class Trajectory derivatives' (TrReal, TrInteger, TrPoint, etc) addHas() functions are mapped to one Java Trajecoty.addHas() method (which is implemented locally in Java). getHas and removeHas provides just the interface to AMOS II functions. The same applies to Intime, Trajectory, Moving, Interval and Range.

Methods Gethas and Getsorted returns the classes PointScan, IntimeScan, etc. These are simply the iterators to browse the list of items. The methods are current (), eos(), nextRow().

Each of Oid derivatives has the delete method implemented (not displayed in the diagram), which simply deletes the object from AMOS II DBMS.

For creating the objects, the Factory design pattern is used:



AMOSDataFactory is an abstract AMOS II data creation factory though may be instanced. Static method getFactory returns the factory suitable for requester (parameter p).

Each of Base, Spatial and Time classes (int, real, .., point,.., instant) should have specialized factory which is derived from AMOSDataFactory. I.e., for the type real the factory should be AMOSRealFactory.

Specialized factories make sure suitable Temporal and Range type objects will be created (for type real, MReal, TrReal, etc.). getFactory returns factory suitable for the parameter p. If the object proxy p (parameter) represents not specialized object (of type Base, Spatial, Range) in AMOS, corresponding factory will be returned. I.e., if p is of type Point, AMOSPointFactory will be returned. If p is of type Temporal or Range (that is type constructors like moving(a) were used), the factory will be returned according to a. I.e., if p is MReal class object, AMOSRealFactory will be returned.

If no specialized factory can be resolved, the AMOSDataFactory should be returned. It still may create most of the objects.

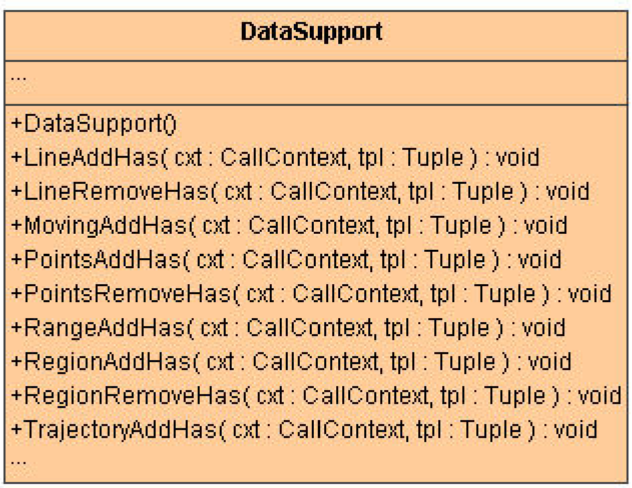
The methods marked in italics return the AMOSExcepion "Can't create abstract class object.". Later, in derivatives, some of them are implemented, as it is in AMOSRealFactory.

In the implementation of the architecture AMOSRealFactory, AMOSPointsFactory and AMOSRegionFactory are included.

To resolve the AMOS II class types from Tuple or Oid object, the class AMOSClassResolver which is pretty simple and not presented here.

DataSupport package and mapping from AMOS to it

This package has only one class and it only provides the interface from the AMOS II to Java implementation of addHas, removeHas.



So Points support functions in AMOS QL are defined like this:

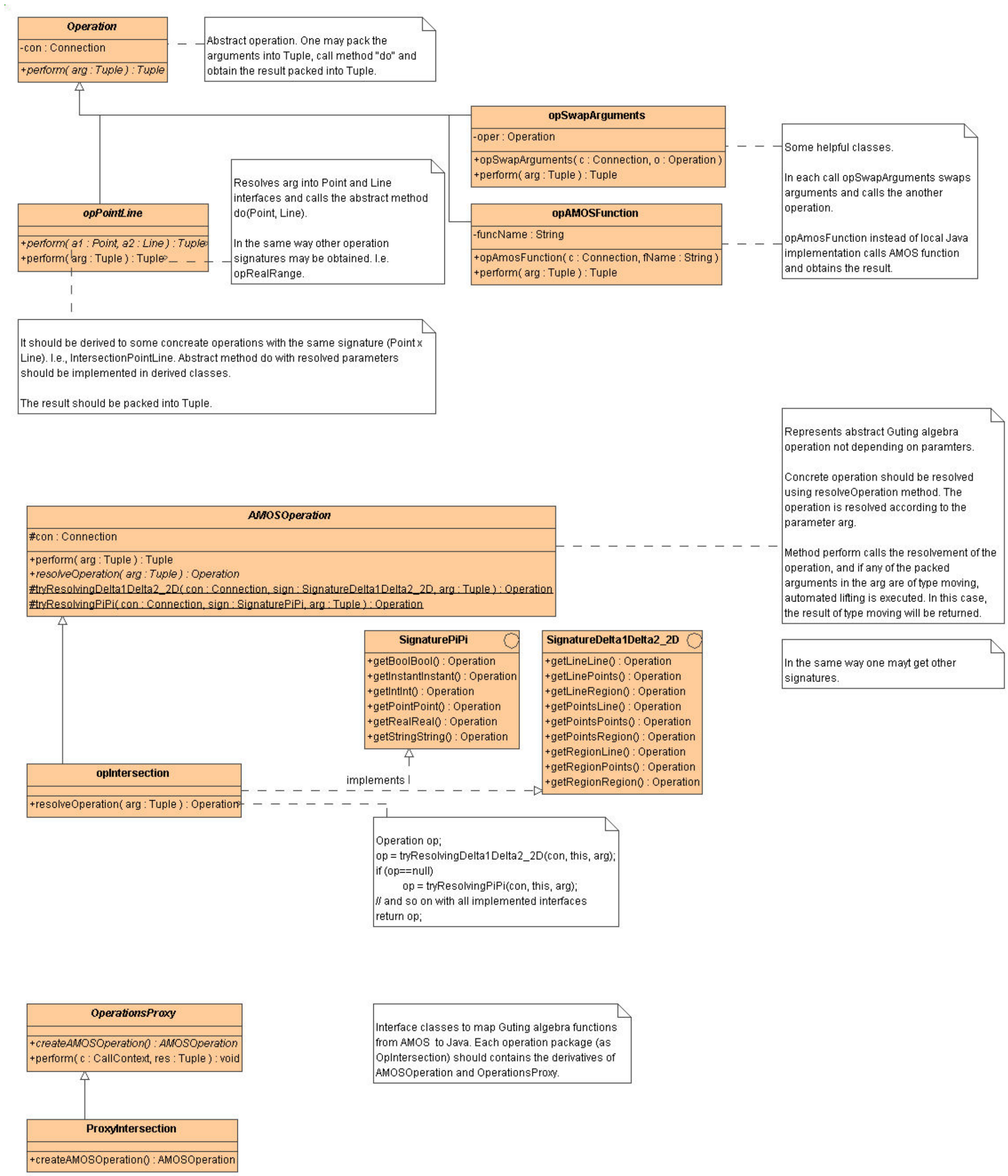
```
create function addHas(Points self, Point p) -> boolean as
     foreign "JAVA:DataSupport.DataSupport/PointsAddHas";
create function getHas(Points self) -> bag of Point as
     select _has(self);
create function removeHas(Points self, Point p) -> boolean
as
     begin remove _has(self)=p end;
```

getHas and removeHAs are implemented locally, because they are not so complex while addHas should be implemented in Java.

The class DataSupport resolves the data types and creates the interfaces to them from the CallContext and calls appropriate method in AMOSData package.

Operations and OperIntersection packages, mapping from AMOS II to OperIntersection

This diagram represents the key ideas:



The class Operation represents an abstract operation. Derivatives of it are an operation with signature, for example opPointLine. The derivatives of this are concrete operations with this signature, for example, IntersectionPointLine. The body of the operation should be implemented in the method “perform (Point, Line)”. The method “perform (arg : Tuple)” unpacks arg into Point and Line interfaces and calls “perform (Point, Line)”. In the same way other concrete signatures and operations implementation can be obtained.

opSwapArguments and opAMOSFunction are some helpful classes. In each call opSwapArguments swaps arguments and calls the another specified operation.

opAMOSFunction instead of local Java implementation calls AMOS function and obtains the result.

AMOSOperation represents abstract Güting algebra operation not depending on parameters. Concrete operation should be obtained using resolveOperation method. The operation is resolved according to the parameter arg. The method “perform” calls the resolving of the operation, and if any of the packed arguments in the arg are of type moving, automated lifting is executed. In this case, the result of the type moving will be returned.

The derivatives of AMOSOperation are Güting algebra operations (i.e., intersection) with implemented interfaces (i.e., SignaturePiPi, etc). These signatures are needed to make sure all needed operations are implemented and make the resolving of operations easier (with static methods tryResolvingPiPi, etc.).

Derivatives of OperationsProxy are interface classes to map Güting algebra functions from AMOS II to Java. Each operation package (as operIntersection) should contain the derivatives of AMOSOperation and OperationsProxy. All “intersection” functions in AMOS II should be mapped to “Operations.operIntersection.ProxyIntersection/perform” disregarding what parameters are specified. For example:

```
create function intersection(Points a1,  Region a2) -> Points as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(Region a1,  Points a2) -> Points as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(MPoints a1,  Region a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(Region a1,  MPoints a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(Points a1,  MRegion a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(MRegion a1,  Points a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(MPoints a1,  MRegion a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
create function intersection(MRegion a1,  MPoints a2) -> MPoints as
      foreign
"JAVA:Operations.operIntersection.ProxyIntersection/perform";
```

The mechanism explained before implements the resolving of arguments and operation.

III. Prepared test cases

The tests were prepared to evaluate the specified use cases.

Detailed use case	Manipulating Güting algebra objects
Test file	testPoints.AMOSql
Explanation	5 objects of type Point are created and 1 of the type Points. Two Point objects have the same (x,y) values. All Point objects are added to Points (some of them several times). All the time, the container Points should contain only 4 items (according to the requirements). Then, it is attempted to remove not included item. Finally, all Point objects are removed from the container Points.

Detailed use case	Calling not lifted Güting algebra operations
Test file	TestIntersection.AMOSql
Explanation	There are created container Points with 4 points, and Region with 2 Faces. The intersection of them should produce container Points with 2 objects of type Point. The calls to intersection (Points, Region) and intersection (Region, Points) are made.

Detailed use case	Calling lifted Güting algebra operations
Test file	TestMovingIntersection.AMOSql
Explanation	There are created container Points :ps1, with 4 points, and Region :reg1, with 2 Faces. The intersection of them should produce container Points with 2 objects of type Point. Then MPoints is created. The time values are defined as: [2,8,14,20],[26,32,38,44],[50] and the value of :ps1 is assigned to each. Brackets mean the trajectories, so the time values are grouped into 3 trajectories. MRegion is creted and the time values are: [0,10,20], [30,40,50]. The value :reg1 is assigned to each. The call to intersection (MPoints, MRegion) should produce MPoints object with time values [2, 8, 10, 14, 20], [30, 32, 38, 40,44], [50].

IV. Comparison to Güting algebra

The identified use cases in the section “Identifying architecturally significant use cases” covers most of Güting algebra issueses, but there left some uncertaintces and differences from the article [1].

Aggregate functions	There shouldn't appear any significant problems implementing them. It is explained in [2] how aggregate functions should be defined in AMOS QL (functions of a bag of some objects) and AMOSData package provides a good base (interfaces to AMOS II objects and scanning bags) for implementing these functions.
Keyword "when"	There is no direct way of implementing the keyword "when". So it is suggested to use the sentence "atperiods(x, deftime(at(f(x), TRUE)))" in QL instead as it is explained in [1].
Implementation of operators "<", etc.	AMOS II is flexible at this point and the overloading of these operators is possible. They may be implemented in the same way as other operations (like intersection). The difference is, that the most of these operations will be implemented locally in AMOS QL, but, i.e., implementation of lifting will remain in Java.
Too high requirements	In [1] pretty high requirements are formulated for data containers (like Line, Region, etc). The drawback of this very complicated and slow implementation. It may happen for the implementation of operations such complexity is not needed. The suggestion is to implement them only if they are needed for implementation of Güting algebra operations. This trick is risky and should be observed, so it is **added to the risks list**.

Summary and future work

During this project analysis of requirements and establishment of architecture was done. Most of the requirements are presented in [1]. Here the analysis mainly deals with defining uncertain data classes (faces, curves, etc.). They are chosen to satisfy the definitions in [1] and to make them possible to implement. Then the chosen use case try to cover most significant parts of Güting algebra. All work building the base of the architecture was done implementing these use cases. Some tests were created to evaluate the implementation.

In the design and implementation, Güting algebra objects were defined in AMOS II and we succeeded to define and establish Java interface to AMOS II objects and operations. Operations in AMOS II were defined using typed parameters of functions mapped to corresponding Java types. Also, a number of problems (e.g. resolving AMOS II data types, implementing Güting algebra constructors) in Java were solved using design patterns and connection to AMOS II. It was shown how one may use these interfaces and generic established mechanisms to implement operations and extend the architecture to new object types. Finally we succeeded to automate operations lifting and that is an advantage for making further implementation easier. A set of operations and data types were chosen and implemented to show the correctness of the architecture.

Nnext, the work should be done finishing the architecture: defining other data structures (like moving(Line), Range(Integer), etc), defining more signatures (that may be done fully implementing the “intersection” operation).

Then, more use cases may be identified, to make the architecture more complete, for example, aggregate functions, binary predicates and lifting of them, other Güting algebra operations.

Appendix

A. Risks List

Name	Description	Priority	Status and comments
Curve requirement change	Curve requirement (definition 7) has been changes. May cause failure of some functions working correctly.	Medium	Identified, postponed to the implementation and testing.
Not implementing complex requirements	Most of the requirements are too complex to implement effectively, so it suggested do not implement them unless they are needed in operations implementation.	Medium	Identified, postponed to the implementation part and testing parts. Should be continuously observed.

B. Included files

createDataStructures.AMOSql	AMOS QL file creating data structures and data support functions (addhas, gethas, etc., most of them are mapped to the DataSupport package)
TestPoints.AMOSql	Test case for “Manipulating Güting algebra objects”
TestIntersection.AMOSql	Test case for “Calling not lifted Güting algebra operations”
TestMovingIntersection.asmosql	Test case for “Calling lifted Güting algebra operations”
Güting.dmp	AMOS II image file with objects created by “createDataStructures.AMOSql”. May be loaded with command: JavaAMOS.bat Güting.dmp
Folder AMOSData	Implemented AMOSData package
Folder DataSupport	Implemented DataSupport package
Folder Operations	Implemented Operations package and OperIntersection sub-package.

C. References

[1] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis: A foundation for representing and quering moving objects. *ACM Trans. Database Syst.* 25 (1): 1-42 (2000)

[2] T.Risch, V.Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers. *Concurrency and Computation: Practice and Experience J*. 13(11), John Wiley & Sons, September, 2001.

[3] *http://www.csd.uu.se/~udbl/AMOS/doc/AMOS_users_guide.html*

[4] *http://www.csd.uu.se/~udbl/AMOS/doc/AMOS_concepts.html*