Transactions in GIS, 2010, 14(2): 131–146

Research Article

The Space Package: Tight Integration between Space and Semantics

Willem Robert van Hage
Department of Computer Science
Vrije Universiteit Amsterdam

Jan Wielemaker
Department of Computer Science
Vrije Universiteit Amsterdam

Guus Schreiber
Department of Computer Science
Vrije Universiteit Amsterdam

Abstract
Interpretation of spatial features often requires combined reasoning over geometry and semantics. We introduce the Space package, an open source SWI-Prolog extension that provides spatial indexing capabilities. Together with the existing semantic web reasoning capabilities of SWI-Prolog, this allows efficient integration of spatial and semantic queries and provides an infrastructure for declarative programming with space and semantics. There are few systems that provide indexing and reasoning facilities for both spatial and semantic data. A common solution is to combine separate semantic reasoning and geospatial services. Such loose coupling has the disadvantage that each service cannot make use of the statistics of the other. This makes optimization of such a service-oriented architecture hard. The SWI-Prolog Space and Semantic web packages provide a native Prolog interface to both spatial and semantic indexing and reasoning, which makes it easy to write combined query optimizers. Another advantage of the Space package is that it allows declarative logic programming, which means in practice that you say what you want to compute instead of how to compute it. The actual indexing machinery is encapsulated inside Prolog predicates. In this article we describe the interface of the Space package, compare its functionality to alternative software libraries, and show how to work with it using three example applications. These example illustrations include reasoning over movement patterns, dynamically loading geospatial linked data off the semantic web, and setting up a simple KML server.

Address for correspondence: Willem Robert van Hage, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081HV Amsterdam, The Netherlands. E-mail: wrvhage@few.vu.nl

doi: 10.1111/j.1467-9671.2010.01187.x

1 Introduction

Geographical Information Systems have been used successfully to analyze spatial concepts for about five decades. The use of ontologies in such analyses is a relatively recent development (cf. Bernard et al. 2003, Fonseca and Egenhofer 1999, Kolas et al. 2005). A limitation of current state-of-the-art GISs is that they do not support semantics. Most GISs use local identifiers for features as opposed to global URIs. Information about the features is usually stored with "flat" attribute-value pairs. Most GISs do not natively support hierarchical typing of features, property hierarchies, or rules. On the other hand, most semantic reasoning systems support very little "concrete domain" reasoning, with support limited to logical inference. Complex analysis of spatial concepts, such as the interpretation of moving object behavior (Orellana and Renso 2010, Orellana et al. 2009), or the classification of terraced houses based on their relative position (Lüscher et al. 2009), requires software that can deal with both the spatial and semantic aspects of features.

This article presents an infrastructure to reason declaratively over spatial objects. We introduce the Space package, a module for SWI-Prolog that provides spatial indexing. More information about the package can be found at http://www.SWI-Prolog.org/pldoc/package/space.html and the source code itself can be downloaded from the GIT repository at http://www.SWI-Prolog.org/git/space.git (or git://www.swi-prolog.org/home/pl/git/space.git).

In Section 2 we will discuss the motivation for this work. In Section 3 we compare the Space package to related work. In Section 4 we will describe the interface of the Space package in detail. In Section 5 we will describe the architecture of the package and technical implementation issues. In Section 6 we give an indication of the performance of the system. In Section 7 we describe application examples. In Section 8 we discuss future work related to the Space package and in Section 9 we wrap up with some conclusions.

2 Logic Programming and Spatial Reasoning

The goal of our work is to provide an infrastructure for declarative programming over both space and semantics. We choose to do this in SWI-Prolog, because it provides a fast declarative rule-based reasoning platform that provides smooth integration to a general-purpose programming language (Liang et al. 2008), and because of its support of semantic web technology (Wielemaker et al. 2008). However, it does not provide support for geometric operations and spatial indexing. For these two tasks we use external libraries, respectively Geometry Engine Open Source (GEOS; see http://geos.refractions.net/ for additional details) and the Spatial Index Library (Hadjieleftheriou et al. 2005; see http://trac.gispython.org/spatialindex/ for additional details). We had to take the following important decision when designing the Space package:

1. We had to decide at which level of abstraction we make our declarative interface. Some things are easier to write declaratively (e.g. symbolic spatial reasoning, like Region Connection Calculus (RCC-8; Bennett et al. 1998), while other things are easier to write imperatively. In section 4 we will describe the interface we chose and motivate our decisions.
2. We had to decide how profound the integration between spatial and semantic constructs should be. There are many possible degrees of integration. On one side of

the spectrum it would have been possible to wrap existing GISs as a service or with a database wrapper and disclose this to the remainder of Prolog through a declarative interface. On the other side, it would have been possible to write a basic GIS in Prolog. It is very hard to write an efficient query optimizer on a loosely coupled system that combines two different kinds of indices. We decided on an interface that allows us to reuse existing libraries, while still allowing tight enough integration to be able to write query optimization routines that use properties of both the spatial and the semantic index.

3. We had to bridge the gap between spatial databases and geometric operations on one side and pure Prolog predicates on the other in some way. Pure Prolog predicates should always have the same behavior, regardless of the instantiation order determined by the program context. They work though unification of variable arguments, not by side effects like destructive assignment. In section 5 we discuss the implementation issues of spatial queries as pure prolog predicates.

3 Related Work

The three systems that are most similar to the Space package are: Franz Inc.'s AllegroGraph (http://www.franz.com/agraph/); the Jena (McBride 2002) extension Geospatialweb (see http://code.google.com/p/geospatialweb/ and http://geosparql.appspot.com/ for additional details); and the framework built around Jena and PostGIS by Lüscher and colleagues (see http://www.dagstuhl.de/Materials/Files/09/09161/09161.Luescher Patrick.Slides.pdf for additional details) for the classification of types of houses (Lüscher et al. (2009).

AllegroGraph and SWI-Prolog have native RDF and RDFS++ (Allemang and Hendler 2008) support, and use a DIG interface for interfacing with an external DL reasoner (Wielemaker et al. 2008). Geospatialweb uses Jena for storage, which also uses an external system for DL reasoning. AllegroGraph is built on the Allegro Common Lisp system, which also has Prolog rule support. Jena has a forward chaining rule reasoner (see http://jena.hp.com/juc2006/proceedings/reynolds/rules-slides.ppt for additional details). AllegroGraph uses grids to index points. All points in a grid cell are evaluated against the query. The user can specify the granularity of the grid. It is hard to compare the performance of this indexing method to R-Trees, which tune themselves to the spatial distribution of the data. For arbitrarily distributed data R-Trees are faster at query time and grids are faster at indexing time.

The greatest functional difference between the Space package and AllegroGraph is that in AllegroGraph shapes are either encoded in a URI by a naming convention or they are lists of coordinates. Either way, they are not typed structures. In addition, AllegroGraph only supports polygons as queries, not as indexable objects and it does not support nearest neighbor queries.

Geospatialweb does support nearest neighbor queries, but only on points. It does not support any other type of shapes. These points are directly derived from W3C WGS84 `lat` and `long` properties in RDF. They are not first class citizens like in the Space package. Transformations of shapes are not possible.

The system by Lüscher et al. (2009) uses PostGIS, which is a more powerful spatial query system than the Spatial Index Library used by the Space package. PostGIS uses Generalized Search Trees for spatial indexing and supports all major GIS standards for

Table 1 Comparison between the functionality of the Space package and related software

tool name	index type	reasoning	geometry types	query types
Space package	R*-tree	RDFS, DIG	points, lines, polygons	INN, intersection, contains, range
AllegroGraph	grid(s)	RDFS, DIG	points (polygon queries)	contains, range
PostGIS	GiST	–	all OpenGIS SFS	KNN, intersection, contains, range
Geospatialweb	R*-tree	OWL, DIG	points	KNN, range

input and output. However, as opposed to the Space package, it loosely couples space and semantics. This makes it hard to control the performance of complex queries in such a system, because the two separate engines each have their own query optimizers that are unable to anticipate based on each other's statistics. For nearest neighbor queries this is more relevant than for containment and intersection queries, because nearest neighbor queries are potentially unbounded in space. For example, consider the query "Find the nearest Chinese restaurant that serves vegetarian dishes." The spatial database knows the heuristics about where the nearest features are. Perhaps it even knows where the nearest restaurants are if there is an attribute: value pair type: restaurant, but the nearest restaurant matching the two very different complex semantic constraints ChineseRestaurant $\sqcap \exists$*serves*.VegetarianDish could very well be on the other side of the Earth even though there are many nearby restaurants. The spatial index has no access to heuristics about semantics, while an RDF store would have this information, but would be unable to efficiently determine which of the matching individuals is near to the query shape.

Table 1 gives an overview of the capabilities of the space package and the software tools mentioned above.

4 The Space Package Interface

The interface of the Space package was designed to make declarative multimodal statements easy to write. For example, "Scientists born near Amsterdam", using the DBpedia data set looks like:

```
scientist_born_near_amsterdam(Scientist, BirthPlace) :-
    rdfs_individual_of(Scientist, db:'Scientist'),
    rdf(Scientist, dbp:birthPlace, BirthPlace),
    uri_shape(AmsterdamURI, AmsterdamShape),
    space_nearest(AmsterdamShape, BirthPlace).
```

4.1 Shapes as Prolog Terms

The central objects of the Space package are pairs, $\langle u, s \rangle$ of a URI, u, and its associated shape, s. The URIs are linked to the shapes with the `uri_shape/2` predicate. This is

```
?- shape(point(52.3325,4.8673)),
   shape(box(point(52.3324,4.8621),point(52.3348,4.8684))),
   shape(
   polygon([[point(52.3632,4.981)|_],  % the outer shell of the polygon
            [point(52.3631,4.9815)|_] |_  % any number of holes 0..*
           ])).
true.
%% uri_shape(?URI, ?Shape) is nondet.
?- uri_shape('http://www.example.org/myoffice', Shape). % read from RDF
Shape = point(52.3325,4.8673).
```

Figure 1 Examples of supported shapes that can be used both as data and queries in Space package version 0.1.2. Shapes are associated to a URI by the `uri_shape/2` predicate and verified with the `shape/1` predicate

illustrated in Figure 1. We will support all OpenGIS Simple Features, points, linestrings, polygons (with ≥0 holes), multi-points, multi-polygons, and geometry collections; and some utility shapes like box and circle regions.[1]

Both the URIs and the shapes are represented as Prolog terms. This makes them first-class Prolog citizens, which allows the construction and transformation of shapes using regular Prolog clauses, or Definite Clause Grammars (DCGs). We support input from locations encoded in RDF with the W3C WGS84 vocabulary (http://www.w3.org/2003/01/geo) and with the GeoRSS Simple properties and the GeoRSS `where` property leading to an XML literal consisting of a GML element (cf. http://georss.org/). The `uri_shape/2` predicate searches for URI-Shape pairs in SWI-Prolog's RDF triple store. It matches URIs to Shapes by using WGS84 and GeoRSS properties. For example, a URI *u* is associated with the shape *s* = `point(`*lat*, *long*`)` if the triple store contains the triples: ⟨*u*, `wgs84_pos:lat`, *lat*⟩ and ⟨*u*, `wgs84_pos:long`, *long*⟩; or when it contains one of the following triples: ⟨*u*, `georss:point`, "*lat long*"⟩ or ⟨*u*, `georss:where`, "`<gml:Point><gml:pos>` *lat long* `</gml:pos></gml:Point>`"⟩. The XML literal containing the GML description of the geometric shape is parsed with a DCG that can also be used to generate GML from Prolog shape terms. By allowing the shapes to be formulated as XML literals we can leave the specification of many issues, like various coordinate spaces, to GML. Also, it makes it possible to extend the system in the future with other relevant XML Schemas, like CityGML.

4.2 Adding, Removing, and Bulkloading Shapes

The spatial index can be modified in two ways: By inserting or retracting single URI-shape pairs respectively using the `space_assert/3`, or the `space_retract/3` predicate; or by loading many pairs at once using the `space_bulkload/2` predicate or its parameterless counterpart `space_index_all/0` which simply loads all the shapes it can find with the `uri_shape/2` predicate into the default index. The former method is best for small manipulations of indices, while the latter method is best for the loading of large numbers of URI-shape pairs into an index. The Space package can deal with multiple indices to make it possible to divide sets of features. Indices are identified with a name handle, which can be any Prolog atom.[2] The actual indexing of the shapes is

```
%% space_assert(+URI, +Shape, +IndexName) is det.
%% space_retract(+URI, +Shape, +IndexName) is det.
%% space_index(+IndexName) is det.
?- space_assert(ex:myoffice, point(52.3325,4.8673),
                demo_index). % only adds it to the 'demo_index' queue
true.
?- space_contains(box(point(52.3324,4.8621), point(52.3348,4.8684)),
                  Cont, demo_index).
% uses 'demo_index', so triggers a call to space_index('demo_index').
Cont = 'http://www.example.org/myoffice' . % first instantiation, etc.
```

Figure 2 The `space_assert/3` and `space_retract/3` predicates put modifications to the index in a queue that is processed by `space_index/1` before the execution of a query on the index (lazy evaluation). `ex:myoffice` is a QName using an example namespace

```
%% space_bulkload(:Closure, +IndexName) is det.
%% uri_shape(?URI, ?Shape) is nondet.
?- space_bulkload(uri_shape, demo_index).
true.
```

Figure 3 Bulkloading is accomplished with the `space_bulkload/2` predicate, which creates a new index of all URI-Shape pairs it can find with the supplied predicate. In this example we use the `uri_shape/2` predicate from the `space` module to find candidates for indexing

performed using lazy evaluation (i.e. indexing is delayed as long as possible). Assertions and retractions are put on a queue that belongs to an index. The queue is committed to the index whenever a query is performed, or when a different kind of modification is called for (i.e. when the queue contains assertions and a retraction is requested or vice versa). Index modification operations are illustrated in Figure 2. An indication of the performance of bulkloading (Figure 3) and single assertions is given in Figure 11.

4.3 Query Types

We chose the three most common spatial query types as our basic building blocks: *containment*, *intersection*, and *nearest neighbor*. These three query types are implemented as pure Prolog predicates, respectively `space_contains/3`, `space_intersects/3`, and `space_nearest/3`. These predicates work completely analogously, taking an index handle and a query shape to retrieve the URI of a shape matching the query, which is bound to the second argument. Any successive calls to the predicate try to re-instantiate the second argument with a different matching URI. This is illustrated in Figure 4. The results of containment and intersection queries are instantiated in no particular order, while the nearest neighbor results are instantiated in order of increasing distance to the query shape. The `space_nearest_bounded/4` predicate is a containment query based on `space_nearest/3`, which returns objects within a certain range of the query shape in order of increasing distance. An indication of the performance of nearest neighbor queries is given in Figure 11.

```
%% space_contains(+QueryShape, -ContainedURI, +IndexName) is nondet.
%% space_intersects(+QueryShape, -IntersectedURI, +IndexName) is nondet.
%% space_nearest(+QueryShape, -NearURI, +IndexName) is nondet.
%% space_nearest_bounded(+Query, -NearURI, +Range, +IndexName) is nondet.
?- space_nearest(point(52.3325,4.8673), N, demo_index).
N = 'http://sws.geonames.org/2759113/' ;   % retry, ask for more
N = 'http://sws.geonames.org/2752058/' ;   % retry
N = 'http://sws.geonames.org/2754074/' .   % cut, satisfied
```

Figure 4 Three types of queries: containment, intersection, and incremental nearest neighbor. All query types return one value, a URI, at a time. There exist short notations of these predicates with arity two that automatically use the default index

```
% Convert a WKT shape into GML and KML
?- wkt_shape('POINT ( 52.3325 4.8673 )', Shape), % instantiate from WKT
   gml_shape(GML, Shape),
   kml_shape(KML, Shape).
Shape = point(52.3325, 4.8673),
GML = '<gml:Point><gml:pos>52.3325 4.8673</gml:pos></gml:Point>',
KML = '<Point><coordinates>4.8673,52.3325</coordinates></Point>' .
```

Figure 5 Converting a WKT geometry object into a Prolog shape term, and converting it to GML and KML. The conversion can be accomplished in any direction between these formats using the same predicates with different variables instantiated

4.4 Importing and Exporting Shapes

Besides supporting input from RDF we support input and output for other standards, like GML (http://www.opengeospatial.org/standards/gml), KML (http://code.google.com/apis/kml/) and WKT (http://en.wikipedia.org/wiki/Well-known_text). All shapes can be converted from and to these standards with the `gml_shape/2`, `kml_shape/2`, and `wkt_shape/2` predicates. An illustration of this is shown in Figure 5.

In addition to standard shape input and output predicates, the Space package offers some extra facilities for dealing with KML. You can render KML files from loaded URI-Shape pairs with the `kml_save_header/2`, `kml_save_shape/3`, and `kml_save_footer/1` predicates. This makes it possible to set up a KML server on top of the Space package with a few lines of code. This is illustrated in Figure 14. In addition, you can load URI-Shape pairs directly from KML files if the KML Geometry elements have an XML ID attribute. For example, the KML snippet `<Point ID="http://example.org/x"><coordinates>1.0 2.0</coordinates></Point>` is transformed into the URI-Shape pair `uri_shape('http://example.org/x', point(1.0,2.0))`. This is accomplished by passing the `kml_file_uri_shape/3` predicate to `space_bulkload/2` as a *closure* with the first argument bound to the KML file name and the second and third arguments as free variables. This is illustrated in Figure 6.

```
% If the KML Geometry elements have an ID attribute,
% you can load them from a file, e.g. 'office.kml'
% with kml_file_uri_shape/3, a variant of uri_shape/2
% that loads from KML.
?- space_bulkload(kml_file_uri_shape('office.kml'), 'demo_index').
% Added 12 URI-Shape pairs to demo_index
true.

% You can insert the same objects one by one like this:
?- forall( kml_file_uri_shape('office.kml', Uri, Shape),
           space_assert(Uri, Shape, 'demo_index') ).
true.
```

Figure 6 Using a custom closure to bulkload and insert URI-Shape pairs instead of `uri_shape/2`: in this case `kml_file_uri_shape/3` from the KML module

4.5 Integration of Space and Semantics

The non-deterministic implementation of the queries makes them behave like a lazy stream of solutions. (i.e. computation to find results is delayed until a result is explicitly requested and if only one result is requested then the computation to find additional results is never performed). This allows tight integration with other types of reasoning, like RDF(S) reasoning or other Prolog rules. An example of combined RDFS and spatial reasoning is shown in Figure 7.

4.6 Loading RDF with Shapes from the Web

There are large volumes of interlinked geospatial data on the semantic web: for example, in GeoNames, DBpedia, and LinkedGeoData. Loading all these data sets at the same time is hard for many systems. Handheld devices, with their limited memory capacities, can have an especially hard time. Often it is unnecessary to have an entire data set in memory at the same time. In many cases it is possible to fetch parts of these data sets either by dereferencing the URI of the concept you are interested in (one of the central ideas of the semantic web), which should return RDF about the concept, or by accessing a web service that provides RDF (Hartig et al. 2009). For most data sets in the Linked Data cloud (http://linkeddata.org/) such web access exists. The Space package provides functionality to automatically load RDF from a URL and index all spatial information from the fetched RDF. All this information is stored with its source URL and can be removed from memory again. This is illustrated in Figure 9 and an application example is shown in Section 7.2.

5 Architecture

The Space package consists of C++ and Prolog code. The division into components is shown in Figure 10. The main component is the Prolog module `space`. All parsing and generation of input and output formats is accomplished in Prolog. All index manipulation is accomplished through the foreign language interface (FLI) from Prolog to C++.

```
% Finds nearest railway stations in the province Utrecht (in GeoNames)
uri_shape(ex:myoffice, Office),
   rdf(Utrecht, geo:name, literal('Provincie Utrecht')),
   space_nearest(Office, Near),
   % 'S' stands for a spot, like a building,
   % 'RSTN' stands for railway station
   rdf(Near, geo:featureCode, geo:'S.RSTN'),
   % 'Near' connected to 'Utrecht' by transitive 'parentFeature'
   rdf_reachable(Near, geo:parentFeature, Utrecht),
   rdf(Near, geo:name, literal(Name)),  % fetch the name of 'Near'
   uri_shape(Near, Station), % fetch the shape of the station
   % compute actual distance in kms
   space_distance_greatcircle(Office, Station, Distance, km).
Utrecht = 'http://sws.geonames.org/2745909/', % first instantiation
Near = 'http://sws.geonames.org/6639765/',
Name = 'Station Abcoude' ,
Station = point(52.2761, 4.97904),
Distance = 9.85408 ;                          % etc.
```

Figure 7 Example code showing tight integration of a spatial query and RDFS reasoning. Query optimization would involve reordering the predicates. Integration of multiple spatial queries can be accomplished in the same way. Since the queries return URIs, an intermediate URI-Shape predicate is necessary to get a shape that can be used as a query. An example is shown in Figure 8

```
% Find features inside nearby polygons
?- uri_shape(ex:myoffice, Office),
   space_nearest(Office, NearURI),
   uri_shape(NearURI, NearShape), % look up the shape of 'NearURI'
   NearShape = polygon(_),        % assert that it must be a polygon
   space_contains(NearShape, Contained).
```

Figure 8 Example code showing nested spatial queries

The `space_bulkload/2` predicate also communicates back across the FLI from C++ to Prolog, allowing the indexing functions to ask for candidates to index from the Prolog database, for example, by calling the `uri_shape/2` predicate.

5.1 Incremental Search and Non-determinism

The three search operations provided by the Space package all yield their results incrementally, i.e. one at a time. Prolog predicates actually do not have return values, but instantiate parameters. Multiple return values are returned by subsequently instantiating the same variable, so the first call to a predicate can make different variable instantiations than the second call. This standard support of non-deterministic behavior makes it easy to write incremental algorithms in Prolog.

```
%% space_load_url(+URL, +IndexName) is det.
%% space_unload_url(+URL, +IndexName) is det.
?- space_load_url('http://sws.geonames.org/2756888/', 'demo_index').
% Parsed "http://sws.geonames.org/2756888/" in 0.01 sec; 31 triples
% Added 1 URI-Shape pair to demo_index
true.

% URI-Shape pairs derived from the given URL
?- uri_shape(URI, Shape, 'http://sws.geonames.org/2756888/').
URI = 'http://sws.geonames.org/2756888/',
Shape = point(52.3383, 4.95972)

?- space_unload_url('http://sws.geonames.org/2756888/', 'demo_index').
% Removed 1 URI-Shape pair from demo_index
```

Figure 9 Load RDF describing GeoNames feature #2756888 from the web and automatically add all URI-Shape pairs that can be inferred from the RDF into an index

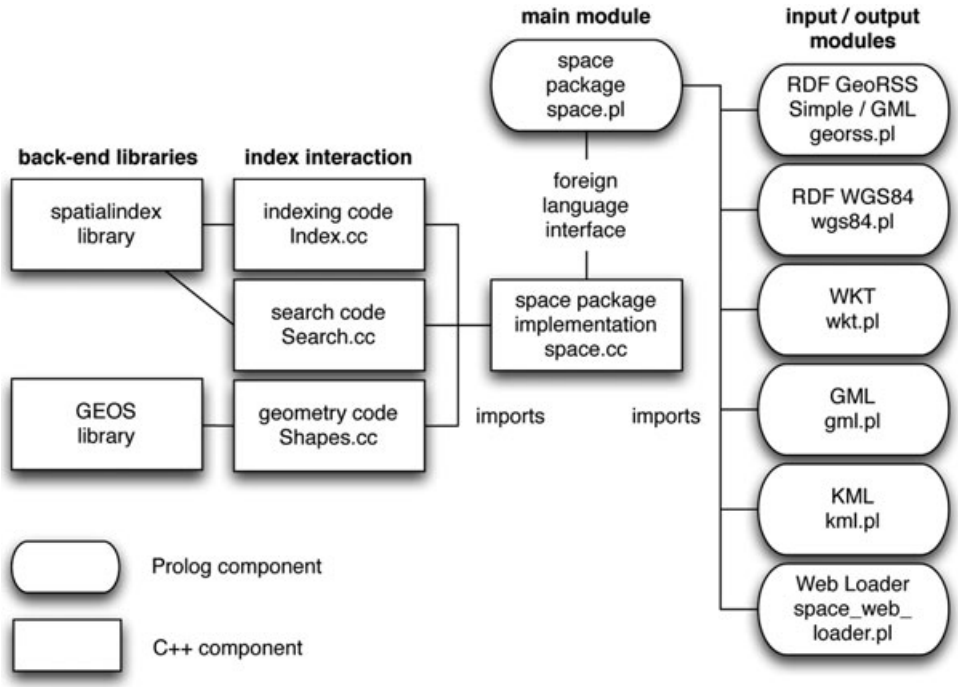

Figure 10 The architecture of the Space package

Internally, the search operations are handled by C++ functions that work on an R*-tree index from the Spatial Index Library (Hadjieleftheriou et al. 2005). The C++ functions are accessed with the SWI-Prolog foreign language interface. To implement non-deterministic behavior the query functions have to store their state between successive calls and Prolog has to be aware which state is relevant to every call.

The Spatial Index library does not include an incremental nearest neighbor, so we implemented an adaptation of the algorithm as described in Hjaltason and Samet (1999). The original algorithm emits results, for example, with a callback function, without

breaking from the search loop that finds all matches. Our adaptation breaks the search loop at every matching object and stores a handle to the state (including the priority queue) so that it can restart the search loop where it left off. This makes it possible to tie the query strategy into the non-deterministic foreign language interface of SWI-Prolog with very little time overhead.

6 Performance

Currently, so few systems exist that can deal with space and semantics that there are no existing benchmarks for queries that require both. In order to give an impression of the performance of the Space package we have computed the CPU time and memory costs of some typical bulkloading, assert/retract, and 10.000 nearest neighbor query statements (nearest neighbor being the slowest query type) on an arbitrary selection of the LinkedGeoData set of OpenStreetMap data (see http://linkedgeodata.org/ and http://www.openstreetmap.org/ for additional details). We used a Intel Core 2 Duo 2.66 GHz with 4 GB of main memory and 6 MB of L2 Cache, and a bus speed of 1.07 GHz. A million points load from RDF in memory in about four minutes. A nearest neighbor query takes around 0.8s to retrieve 10,000 matches, regardless of the size of the index. Bulkloading takes linear time to load points into memory, while single assertions take exponential time. For small data sets (i.e. for hundreds of points) bulkloading is only slightly faster than single assertions, but at 100,000 points the difference is already greater than a factor of 10. An overview is shown in Figure 11. Given the decreasing price of memory we decided to use a memory store by default, although the Space package can be set to use a file store with a memory buffer. Memory use in version 0.1.2 lies around 250B per point.

7 Example Applications

In this section we will outline a few application examples to illustrate how the space package can be used in practice. The examples show respectively: the combination of movement pattern rules and spatial indexing; traversing the geospatial web; and setting up a KML server.

7.1 Example 1: Ship Behavior

To show the applicability of the Space package for the recognition of meaningful movement patterns we refer to van Hage et al. (2009) who describe using the Space package to reason over ship behavior. For this we use ship location information from AIS messages (see http://en.wikipedia.org/wiki/Automatic_Identification_System for additional details) and we extended GeoNames (http://www.geonames.org) with locations of harbors. On top of these two sources we define declarative rules to qualify ship behavior. One example of such a rule is the definition of *trip*. By means of a compression algorithm the streams of AIS messages are segmented into intervals where a ship is *speeding up*, *slowing down* or *stopped*. A *trip* can then be defined as *stopped* near a harbor (GeoNames featureCode `H.HBR`), then $\neg$*stopped* for a while, and then *stopped* near a different harbor. On top of such *trips*, defining the behavior of a ferry can be accomplished by declaring that there are consecutive trips that lead back to the same harbor. The connection between the Space package, RDF reasoning, and behavior rules is illustrated in Figure 12.

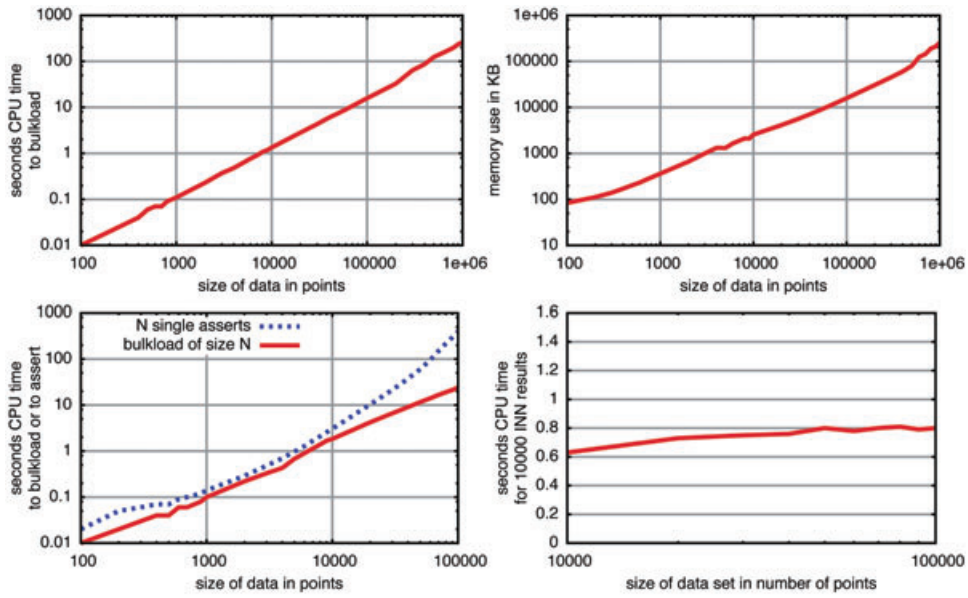

Figure 11 Tentative performance figures on LinkedGeoData points: (upper left) CPU time taken to bulkload *n* points; (upper right) Memory taken to load *n* points; (lower left) CPU time taken to bulkload versus to make single assertions; (lower right) CPU time taken to computer 10.000 nearest neighbors on varying data set sizes, about constant around 0.8s. All figures concern version 0.1.2 of the Space package

7.2 Example 2: Dynamic Loading of Linked Data

Figure 13 shows how the Space package can be used to find parking lots near to the "Amsterdam Zuid" train station. Information about the station is looked up from the GeoNames search engine. This is done by constructing a URL that encodes the query for the station's name. Dereferencing this URL by calling `space_load_url/1` yields a piece of RDF that contains WGS84 properties that specify its location. All location information from the RDF is automatically indexed by the `space_load_url/1` predicate. The resulting URI-Shape pair can then be used to construct a query to the LinkedGeoData search engine, which returns nearby features from OpenStreetMap (http://www.openstreetmap.org/). The RDF returned by this query contains parking lots that can be found with a Prolog RDFS query. We order the results using the `space_nearest/2` predicate. This example uses constructed URLs, but many URIs can be dereferenced to retrieve RDF as in the case of the URIs of GeoNames Features.

7.3 Example 3: A Simple KML Server

The Space package can be used to quickly set up a KML visualization of indexed URI-Shape pairs. In Figure 14 we show a simple example of this without any sophisticated selection procedure or rendering styles for the sake of simplicity. This example shows how you can import spatial and semantic information from a remote source and host visualize it for yet another remote source.

```
% rule defining ferry behavior semantics
% trip from Harbor0  to Harbor1 and back
% via consistent movement Segment Seg0, Seg1, and Seg2
ferry_trip(Seg0, Harbor0, Seg1 ,Harbor1, Seg2, Harbor0) :-
    trip(Seg0, Harbor0, Seg1, Harbor1),
    trip(Seg1, Harbor1, Seg2, Harbor0).

% rule defining trip behavior semantics
trip(Seg0, Harbor0, Seg1, Harbor1) :-
    % ship is stopped at harbor Harbor0 during Seg0
    stopped_at_harbor(Seg0, Harbor0),
    % in successive segments between Seg0 and Seg1 it is moving
    move_from_to(Seg0, Seg1),  % actual code omitted in this example
    % ship is stopped at harbor Harbor1 during Seg1
    stopped_at_harbor(Seg1, Harbor1).

% rule defining trip behavior semantics
stopped_at_harbor(Segment, Harbor) :-
    stopped(Segment),  % semantics of behavior
    % fetch location of segment
    location_of_segment(Segment,Location)
    % find nearest place within margin
    space_nearest_bounded(Location, Harbor, 0.175),  % spatial index
    rdf(Harbor, geo:featureCode, geo:'H.HBR').  % semantics of place
```

Figure 12 Selected SWI-Prolog rules that illustrate the linking of domain-level data to place and behavior semantics using the Space package. The rules in this example come from code used to classify ferry behavior described in van Hage et al. (2009)

We set up a web server with the `http_server/2` and attached it to a Prolog predicate that handles all requests to this server with `http_handler/3`. The handler fetches a parameter from the HTTP GET request, which in this example denotes a URL from which we fetch RDF with geospatial information. This RDF is processed by the Space package like in the example in Section 7.2. Then we write a KML document as a reply to the HTTP request. This document is constructed using the `kml_save_*` predicates (see Section 4.4). For each URI-Shape pair indexed by the space package we construct a KML Placemark tag containing the KML translation of the shape. In addition to the tag describing the shape, we add a `name` tag that contains the URI of the shape. After writing the document as a response to the request we unload the data downloaded from the URL.

8 Conclusions and Future Work

We presented the Space package, an open source library that adds spatial indexing capabilities to SWI-Prolog and allows declarative programming over spatial concepts.

```
% Construct a URL for the GeoNames search engine
?- atomic_list_concat([ 'http://ws.geonames.org/search?type=rdf&',
                        'name=Station%20Amsterdam%20Zuid' ],
                        GeoNamesURL),
   space_load_url(GeoNamesURL),  % Load RDF from GeoNames
   rdf(SAZ, geo:name, literal('Station Amsterdam Zuid')),
   uri_shape(SAZ, point(Lat,Long)),
   % Construct a URL for the LGD search engine
   atomic_list_concat([ 'http://linkedgeodata.org/triplify/near/',
                        Lat, ',', Long, '/1000'],
                        LinkedGeoDataURL),
   space_load_url(LinkedGeoDataURL),   % Load RDF from LGD
   space_nearest(point(Lat,Long), P),  % Order by distance
   rdfs_individual_of(P, lgd:parking). % Find parking lots
SAZ = 'http://sws.geonames.org/6639947/',
Lat = 52.3389,
Long = 4.87235,
LinkedGeoDataURL =
   'http://linkedgeodata.org/triplify/near/52.3389,4.87235/1000',
P = 'http://linkedgeodata.org/triplify/node/34206635#id'
```

Figure 13 Dynamically load RDF over the web and automatically index additional retrieved locations. This example loads RDF about the train station Amsterdam Zuid from the GeoNames search engine. We take the location of the station from the RDF and use it to find RDF about nearby features from the LinkedGeoData OpenStreetMap data set. In this data we search for nearby car parks ordered by distance from the station

```
server(Port) :- http_server(http_dispatch, [port(Port)]).

kml_handler(Request) :-
http_parameters(Request, [ url(URL,[]) ], []),  % get URL
space_load_url(URL),  % get RDF and URI-Shape pairs from URL
format('Content-type: application/vnd.google-earth.kml+xml~n~n'),
kml_save_header(current_output, []),
forall( uri_shape(URI, Shape),  % render KML for each pair
        (   kml_save_shape(current_output,
            placemark(Shape,['ID'=URI],[]),
            [ content([name(URI)]) ]) % add name tag to Placemark
        )
      ),
kml_save_footer(current_output),
space_unload_url(URL).  % unload the RDF and URI-Shape pairs

% Listen for requests on http://localhost:8080/space.kml
:- http_handler('/space.kml', kml_handler, []).
:- server(8080).
```

Figure 14 Set up a simple KML server that fetches RDF from a URL that is passed as a HTTP parameter and render KML of all the URI-Shape pairs described by the RDF

The two main strengths of the Space package are its tight integration with the remainder of SWI-Prolog, which allows relatively easy query optimization for multimodal queries; and its declarative interface, which allows the formulation of short, understandable code, while not limiting expressivity. The Space package supports common geospatial and web standards, such as GML, KML and WKT, and in combination with RDF: GeoRSS Simple and GML, and the W3C Basic Geo (WGS84) Vocabulary.

At this moment, in version 0.1.2, the Space package only supports points, box regions and polygons with optional holes, but not linestrings and various kinds of multi-shapes. In the near future we will completely support the GML Simple Feature Specification. We would like to extend the Space package with support for common GIS file formats with some methods to connect URIs to the shapes that come from such files. A possible implementation for this could be in the form of a database connector for PostGIS. This would also allow the Space package to consult PostGIS for complex geospatial queries and geometric operations. For a better performance analysis, and comparison to the systems mentioned in Section 3, we will set up a set of representative spatial-semantic queries. Further future work is to make a query optimizer that combines heuristics from the SWI-Prolog Semantic Web Library and Space package along the lines of Wielemaker (2005). This will allow us to take advantage of the tight integration between space and semantic offered by the Space package. Also, we think it could be useful to implement an extension to the SPARQL RDF query language with functions that provide geospatial query functionality, *cf*. Geospatialweb. This can be accomplished using the Henry SPARQL implementation (http://code.google.com/p/km-pdf/) for SWI-Prolog by Raimond and Sandler (2008).

Acknowledgements

Thanks go to Marios Hadjieleftheriou, Howard Butler, and Véronique Malaisé. This work has been carried out as a part of the Poseidon project in cooperation with Thales Nederland, under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

Endnotes

1 The current version of the Space package, 0.1.2, only supports simple shape types: points, linestrings, polygons (with holes), and box regions. Development on the other shape types is underway.
2 Every predicate in the Space package that must be given an index handle also has an abbreviated version without the index handle argument which automatically uses the default index.

References

Allemang D and Hendler J 2008 *Semantic Web for the Working Ontologist*. San Francisco, CA, Morgan Kaufmann

Bennett B, Isli A, and Cohn A G 1998 A system handling rcc-8 queries on 2d regions representable in the closure algebra of half-planes. In Mira J, del Pobil A, and Ali M (eds) *Methodology and Tools in Knowledge-Based Systems*. Berlin, Springer Lecture Notes in Computer Science Vol. 1415: 281–90

Bernard L, Einspanier U, Haubrock S, Hübner S, Kuhn W, Lessing R, Lutz M, and Visser U 2003 Ontologies for intelligent search and semantic translation in spatial data infrastructures. *Photogrammetrie – Fernerkundung – Geoinformation* 2003(6): 451–62

Fonseca F T and Egenhofer M J 1999 Ontology-driven geographic information systems. In *Proceedings of the Seventh ACM International Symposium on Advances in Geographic Information Systems*, Kansas City, Missouri: 14–9

Hadjieleftheriou M, Hoel E, and Tsotras V J 2005 SaIL: A spatial index library for efficient application integration. *Geoinformatica* 9: 367–89

Hartig O, Bizer C, and Freytag J 2009 Executing SPARQL queries over the Web of Linked Data. *Proceedings of the Eighth International Semantic Web Conference*, Washington, D.C.

Hjaltason G R and Samet H 1999 Distance browsing in spatial databases. *ACM Transactions on Database Systems* 24: 265–318

Kolas D, Hebeler J, and Dean M 2005 Geospatial semantic web: Architecture of ontologies. In Rodriguez M A, Cruz I F, Egenhofer M, and Levashkin S (eds) *GeoSpatial Semantics*. Berlin, Springer Lecture Notes in Computer Science Vol. 3799: 183–94

Liang S, Fodor P, Wan H, and Kifer M 2008 Openrulebench: An analysis of the performance of rule engines. In *Proceedings of the Seventeenth International World Wide Web Conference (WWW 2008)*, Beijing, China

Lüscher P, Weibel R, and Burghardt D 2009 Integrating ontological modelling and bayesian inference for urban pattern classification in topographic vector data. *Computers, Environment and Urban Systems* 33: 363–74

McBride B 2002 Jena: A semantic web toolkit. *IEEE Internet Computing* 6(6): 55–9

Orellana D and Renso C 2010 Developing an interactions ontology for characterizing pedestrian movement behavior. In Wachowicz M (ed) *Movement-Aware Applications for Sustainable Mobility: Technologies and Approaches*. Hershey, PA, IGI Global Publishing: in press

Orellana D, Wachowicz M, Andrienko N, and Andrienko G 2009 Uncovering interaction patterns in mobile outdoor gaming. In *Proceedings of the International Conference on Advanced Geographic Information Systems and Web Services (GEOWS 2009)*, Cancun, Mexico

Raimond Y and Sandler M 2008 A web of musical information. In *Proceedings of the Ninth International Conference on Music Information Retrieval (ISMIR 2008)*, Philadelphia, Pennsylvania

van Hage W. R, Malaisé V, de Vries G, Schreiber G, and van Someren M 2009 Combining ship trajectories and semantics with the simple event model (sem). In *Proceedings of the First ACM International Workshop on Events in Multimedia (EiMM09)*, Beijing, China

Wielemaker J 2005 An optimized semantic web query language implementation in prolog. In *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP 2005)*, Barcelona, Spain

Wielemaker J, Huang Z, and van der Meij L 2008 SWI-Prolog and the web. In Bossi A (ed) *Theory and Practice of Logic Programming*. Cambridge, Cambridge University Press: 363–92