The Space package: Tight Integration Between Space and Semantics

Willem Robert van Hage, Jan Wielemaker, and Guus Schreiber

Vrije Universiteit Amsterdam,
de Boelelaan 1081a, 1081HV Amsterdam, the Netherlands
wrvhage@few.vu.nl, J.Wielemaker@cs.vu.nl, schreiber@cs.vu.nl

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.

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.* [3, 4, 7]). 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, limiting themselves to logical inference. Complex analysis of spatial concepts, such as the interpretation of moving object behavior [11, 12], or the classification of terraced houses based on their relative position [9], requires software that can deal with both spatial and semantic aspects of features.

This paper 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`.[1]

In Sec. 2 we will discuss the motivation for this work. In Sec. 3 we compare the Space package to related work. In Sec. 4 we will describe the interface of the Space package in detail. In Sec. 5 we will describe the architecture of the package and technical implementation issues. In Sec. 6 we give an indication of the performance of the system. In Sec. 7 we describe a practical use-case. In Sec. 8 we discuss future work related to the Space package. And in Sec. 9 we wrap up with a conclusion.

[1] or `git://www.swi-prolog.org/home/pl/git/space.git`

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 (*cf.* [8]), and because of its support of semantic web technology [15]. 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)[2] and the Spatial Index Library[3] [5]. 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) [2]), 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 profoundly 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 rest 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[4]; the Jena [10] extension Geospatialweb[5]; and the framework built around Jena and PostGIS by Lüscher et al.[6] for the classification of types of houses [9]. AllegroGraph and SWI-Prolog have native RDF and RDFS++ [1] support, and use a DIG interface for interfacing with an external DL reasoner [15]. Geospatialweb uses Jena for storage, which also uses an external system for DL reasoning. AllegroGraph is built on the Allegro Common Lisp sytem, which also has Prolog rule support. Jena has a

[2] `http://geos.refractions.net/`

[3] `http://trac.gispython.org/spatialindex/`

[4] `http://www.franz.com/agraph/`

[5] `http://code.google.com/p/geospatialweb/`, `http://geosparql.appspot.com/`

[6] `http://www.dagstuhl.de/Materials/Files/09/09161/09161.LuescherPatrick.Slides.pdf`

forward chaining rule reasoner.[7] The greatest functional difference between the Space package and AllegroGraph is that in AllegroGraph shapes are lists of coordinates, not typed structures; that it only supports polygons as queries, not as indexable objects; and that 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. The system by Lüscher et al. uses PostGIS, which is a more powerful spatial query system than the Spatial Index Library used by the Space package. 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 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 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.

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 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.[8]

[7] `http://jena.hpl.hp.com/juc2006/proceedings/reynolds/rules-slides.ppt`

[8] The current version of the Space package, 0.1.1, only supports points and polygons (with holes) and box regions. Development on the other shape types is underway.

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 [9] and with the GeoRSS Simple properties and the GeoRSS `where` property leading to an XML literal consisting of a GML element.[10] 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: $\langle u$, `wgs84_pos:lat` ,$lat\rangle$ and $\langle u$, `wgs84_pos:long` ,$long\rangle$; or when it contains one of the following triples:
$\langle u$, `georss:point`,"*lat long*"$\rangle$ or $\langle u$, `georss:where`,"`<gml:Point><gml:pos>` *lat long* `</gml:pos></gml:Point>`"$\rangle$. 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.

```
?- 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).
```

Fig. 1. Examples of supported shapes that can be used both as data and queries in Space package version 0.1.1. Shapes are associated to a URI by the `uri_shape/2` predicate and verified with the `shape/1` predicate.

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.[11] The actual indexing of the shapes is per-

[9] `http://www.w3.org/2003/01/geo/`

[10] *cf.* `http://georss.org/`

[11] 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.

formed 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 and single assertions is given in figure 9 in section 6.

```
%% 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.
```

Fig. 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.
```

Fig. 3. Bulkloading is done 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.

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 9 in section 6.

```
%% 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
```

Fig. 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.

4.4 Importing and Exporting Shapes

Besides supporting input from RDF we support input and output for other standards, like GML,[12] KML[13] and WKT.[14] 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.

```
% 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>' .
```

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

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

[12] http://www.opengeospatial.org/standards/gml
[13] http://code.google.com/apis/kml/
[14] http://en.wikipedia.org/wiki/Well-known_text

requested. 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 6.

```
% 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' 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 name of 'Near'
   uri_shape(Near, Station),  % fetch shape of station
   % compute actual distance in km
   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.
```

Fig. 6. 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 done 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 7.

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

Fig. 7. Example code showing nested spatial queries.

5 Architecture

The Space package consists of C++ and Prolog code. The division into components is shown in figure 8. The main component is the Prolog module `space`. All parsing and generation of input and output formats is done in Prolog. All index manipulation is done through the foreign language interface (FLI) from Prolog to C++. 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.

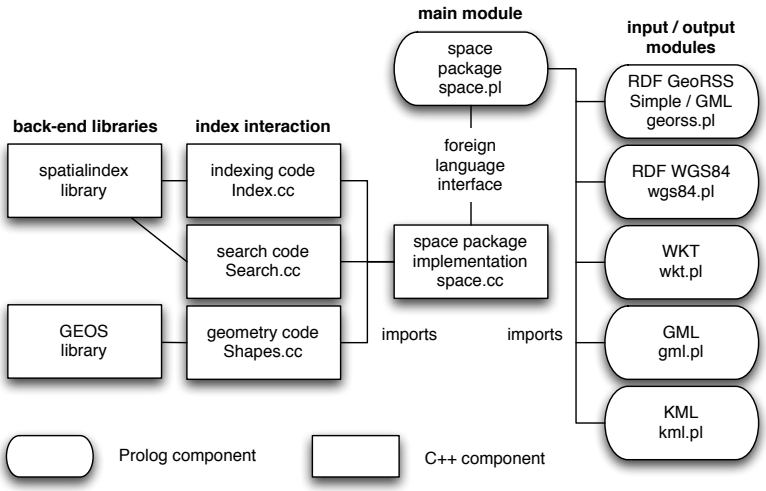

Fig. 8. The architecture of the Space package.

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.

Internally, the search operations are handled by C++ functions that work on an R*-tree index from the Spatial Index Library [5]. 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 described in [6]. 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[15] set of OpenStreetMap data. We used a Intel Core 2 Duo 2.66GHz with 4GB of main memory and 6MB of L2 Cache, and a bus speed of 1.07GHz. 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 is takes linear time to load points into memory, while single assertions take exponential time. For small data sets (hundreds of points) bulkloading is only a slightly faster than single assertions, but at 100.000 points the difference is already over a factor 10. An overview is shown in figure 9. 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 0.1.1 lies around 250B per point. The overhead is larger for smaller data sets.

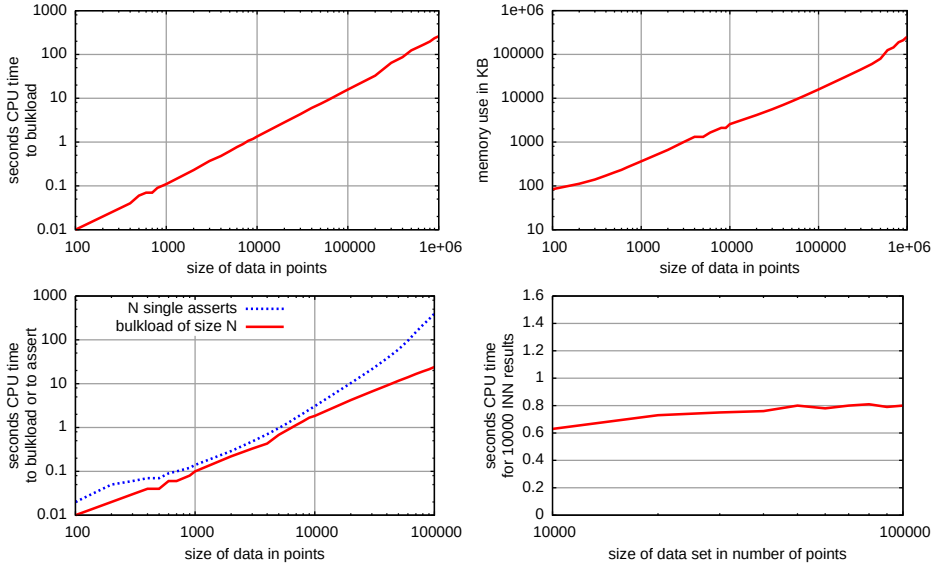

Fig. 9. Tentative performance figures on LinkedGeoData points: (upper left) CPU time taken to bulkload n points; (upper right) Memory taken to load n points; (lower left) CPU time taken to bulkload versus to make single assertions; (lower right) CPU time taken to computer 10.000 nearest neighbors on varying data set sizes, about constant around 0.8s. All figures concern version 0.1.1 of the Space package.

[15] `http://linkedgeodata.org/` and `http://www.openstreetmap.org/`

7 Example Use Case – Ship Behavior

To show the applicability of the Space package we refer to a paper [13] in which describe using the Space package to reason over ship behavior. For this we use ship location information from AIS messages[16] and we extended GeoNames[17] with locations of harbors. On top of these two sources we define declarative rules to qualify ship behavior. One example of such 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 ¬*stopped* for a while, and then *stopped* near a different harbor. On top of such *trips*, defining the behavior of a ferry can be done 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 10.

```
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), % call spatial index
    rdf(Harbor, geo:featureCode, geo:'H.HBR').  % semantics of place
```

Fig. 10. 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 [13].

8 Future Work

At this moment, in version 0.1.1, 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 [14]. This will allow us to take advantage of the tight integration between space and semantic offered by the Space package.

[16] `http://en.wikipedia.org/wiki/Automatic_Identification_System`
[17] `http://www.geonames.org/`

9 Conclusion

We presented the Space package, an open source library that adds spatial indexing capabilities to SWI-Prolog and allows declarative programming over spatial concepts. The two main strengths of the Space package are its tight integration with the rest 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.

Acknowledgements

Thanks go to Marios Hadjieleftheriou 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.

References

1. Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist*. Morgan Kaufmann, 2008.
2. Brandon Bennett, Amar Isli, and Anthony G. Cohn. A system handling rcc-8 queries on 2d regions representable in the closure algebra of half-planes. In *Methodology and Tools in Knowledge-Based Systems*, 1998.
3. L. Bernard, U. Einspanier, S. Haubrock, S. Hübner, W. Kuhn, R. Lessing, M. Lutz, and U. Visser. Ontologies for intelligent search and semantic translation in spatial data infrastructures. *Photogrammetrie - Fernerkundung - Geoinformation*, 2003(6):451–462, 2003.
4. Frederico T. Fonseca and Max J. Egenhofer. Ontology-driven geographic information systems. In *GIS '99: Proceedings of the 7th ACM international symposium on Advances in geographic information systems*, pages 14–19, New York, NY, USA, 1999. ACM.
5. Marios Hadjieleftheriou, Erik Hoel, and Vassilis J. Tsotras. Sail: A spatial index library for efficient application integration. *Geoinformatica*, 9(4), 2005.
6. Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.
7. Dave Kolas, John Hebeler, and Mike Dean. Geospatial semantic web: Architecture of ontologies. In *GeoSpatial Semantics*, pages 183–194. Springer Berlin / Heidelberg, 2005.
8. Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: An analysis of the performance of rule engines. In *Proceedings of the 17th International World Wide Web Conference (WWW2008)*, 2008.
9. Patrick Lüscher, Robert Weibel, and Dirk Burghardt. Integrating ontological modelling and bayesian inference for urban pattern classification in topographic vector data. *Computers, Environment and Urban Systems*, 2009.
10. Brian McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.
11. Daniel Orellana and Chiara Renso. Developing an interactions ontology for characterising pedestrian movement behavior. In Monica Wachowicz, editor, *Movement-Aware Applications for Sustainable Mobility: Technologies and Approaches*. IGI Global Publishing, 2009.

12. Daniel Orellana, Monica Wachowicz, Natalia Andrienko, and Gennady Andrienko. Uncovering interaction patterns in mobile outdoor gaming. In *International Conference on Advanced Geographic Information Systems & Web Services*, 2009.
13. Willem Robert van Hage, Véronique Malaisé, Gerben de Vries, Guus Schreiber, and Maarten van Someren. Combining ship trajectories and semantics with the simple event model (sem). In *Proceedings of the 1st ACM International Workshop on Events in Multimedia*. Sheridan Publishers, 2009.
14. Jan Wielemaker. An optimized semantic web query language implementation in prolog. In *Proceedings of the 21st International Conference on Logic Programming (ICLP 2005)*, 2005.
15. Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. Swi-prolog and the web. In A. Bossi, editor, *Theory and Practice of Logic Programming*, volume 8, pages 363–392. Cambridge University Press, 2008.