Multi-Paradigm Spatial Information Processing

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

von

Bernhard Lorenz

18. Oktober 2006

Erstgutachter:
Prof. Dr. Hans Jürgen Ohlbach, Ludwig-Maximilians Universität, München
-
Zweitgutachter:
Prof. Anthony G. Cohn, Ph. D., University of Leeds, Leeds, U.K.
-
Tag der mündlichen Prüfung:
12. Dezember 2006
-

Acknowledgements

The work presented in this thesis would not exist today without the support and contribution of many fellow researchers and students at the University of Munich. In particular, my gratitude goes to the following colleagues:

- *Prof. Dr. Hans Jürgen Ohlbach,* University of Munich, for his continued and wise guidance during the course of this thesis, and the many fruitful talks and discussions – not only on the issues concerning this work.
- *Professor Anthony G. Cohn, Ph. D.,* University of Leeds, for accepting the burden of being the second reader of this thesis and for his valuable input in the final stages of this work.
- *Prof. Dr. François Bry,* University of Munich, for his guidance, vision, and leadership as the head of the unit and as the person who has made REWERSE a reality.

Also, I thank my former and current colleagues at the research and teaching unit PMS and the MNM-Team. Without their continued support, their expert knowledge, and the good spirit they created, this work would probably not exist today.

Furthermore, I thank the following students who worked on diploma theses related to my work:

- *Edgar-Philipp Stoffel*, who is now a fellow researcher, developed the TransRoute system in his diploma thesis. A short introduction to his work can be found in section 6.3.3 of this thesis.
- *Roman Flammer*, who significantly contributed to the first implementation of the MPLL prototype in course of his diploma thesis.

But above all, most gratitude goes to my family. I thank my parents and my sister for their unconditional love and their kind care and support which I could enjoy all my life. I especially thank my wife Stephanie for her love and patience during the course of this thesis. And last but not least, I thank my son Eric for being the bright and happy person that he is, and for being the sunshine of our lives.

This research has been partly funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779, `http://rewerse.net`.

Bernhard Lorenz

Munich, October 2006

Abstract

For computer software, the semantics of spatial information is not a trivial issue. Due to the differences between human spatial cognition, reasoning, and communication, and its computerised counterpart, a number of difficulties persist in bringing the two paradigms together. While the qualitative way in which humans reason about space enables them to solve all kinds of spatial problems in their environment, the mostly quantitative mechanisms of computerised systems often fail to achieve the same results in a similarly elegant manner. Computers have difficulties to "understand" spatial notions and expressions such as *"near the University"*, *"along the river"*, *"in the southern part of the city"*, *"behind the post office"*, or *"on the shelf next to the sugar"*. For some tasks, however, quantitative methods implemented in software applications offer solutions far superior to qualitative spatial reasoning conducted by humans. This applies, for example, to route planning and navigation.

When reviewing existing work, it becomes apparent that available spatial models and processing/reasoning techniques are mostly limited to certain application domains. A number of systems based on quantitative paradigms offer sophisticated and effective solutions, for example for route planning and navigation. Other work has focussed on qualitative aspects and addresses other problems with similar success. In both cases, there exist several very efficient and well-investigated methods to deal with certain tasks. What is still missing is a unifying component. There is very scarce evidence of work concentrating on making different paradigms available in a single system architecture.

The Multi-Paradigm Location Language (MPLL) presented in this work aims at bridging the gap between quantitative and qualitative spatial models, representations, and processing techniques, in the sense of providing a means for humans to individually specify the way they understand space, and a means for machines to apply quantitative and qualitative methods to these specifications. It provides a framework which facilitates implementing access to different services, which are specifically suited to perform special tasks, such as qualitative reasoning on spatial relations between different entities, or strictly geometrical computations. Spatial notions, however, are often highly subjective. MPLL, therefore, offers flexible means to adapt the specifications to individual needs and preferences, as well as the context of use.

This thesis illustrates the approach to reach the objective of providing a flexible specification language embedded into an extensible system architecture for the purpose of providing the means to specify and process spatial information in *geospatial scenarios*. These scenarios primarily pertain to tasks which concern, and are conducted by, human beings, such as route planning in indoor and outdoor scenarios, and spatial relations between, and interaction with, real world objects.

Zusammenfassung

Die Semantik räumlicher Information mit Computerprogrammen zu verarbeiten, ist kein triviales Vorhaben. Aufgrund der Unterschiede zwischen der Art und Weise wie Menschen räumliche Problemstellungen kognitiv modellieren, diese kommunizieren und lösen, und derjenigen, wie Maschinen dies bewerkstelligen, bestehen einige grundlegende Probleme, beide Paradigmen zu vereinen. Computersysteme können die qualitativen Techniken, die es Menschen erlauben, eine Vielzahl verschiedener Probleme in ihrer räumlichen Umwelt zu lösen, mittels quantitativer Mechanismen oft nur unzureichend nachbilden. Maschinen haben Schwierigkeiten, räumliche Begriffe und Ausdrücke zu "verstehen", wie zum Beispiel *"Nahe der Universität"*, *"entlang dem Fluß"*, *"im Süden der Stadt"*, *"hinter der Post"*, oder *"im Regal neben dem Zucker"*. Quantitative Methoden sind jedoch, in Form entsprechender Softwareapplikationen, in der Lage, einige spezielle Problemstellungen weit besser zu lösen als der Mensch mit seinem räumlichen Denken dazu in der Lage wäre.

Vorhandene Arbeiten auf diesem Gebiet legen nahe, dass verfügbare Modelle und Techniken weitgehend auf bestimmte Anwendungsgebiete zugeschnitten sind. Zahlreiche Systeme basierend auf quantitativen Paradigmen offerieren ausgefeilte und effektive Lösungen. Andere Arbeiten betrachten qualitative Aspekte und widmen sich anderen Problemstellungen mit ähnlichem Erfolg. In beiden Fällen existieren gleichermaßen ausgereifte und effiziente Methoden zur Lösung bestimmter Aufgaben. Was bisher fehlt ist eine übergreifende Komponente. Es gibt nur spärliche Information über Arbeiten, die paradigmenübergreifende Lösungen in einer einzigen Systemarchitektur vereinen.

Die Multi-Paradigm Location Language (MPLL) zielt darauf ab, die Lücke zwischen beiden Modellen, Repräsentationen und Verarbeitungstechniken zu schließen. Sie soll es einerseits Menschen ermöglichen, ihr individuelles räumliches Verständnis spezifizieren zu können, andererseits diese Spezifikationen für quantitative und qualitative maschinelle Verarbeitung zugänglich machen. Im vorgestellten Systemgerüst können Zugänge zu unterschiedlichen Modulen und Diensten implementiert werden, die jeweils einzelne Aspekte räumlicher Problemstellungen lösen können. Dazu gehören z.B. qualitatives räumliches Schließen, als auch strikt geometrische Berechnungen. Da räumliche Begriffe und Ausdrücke einen stark subjektiven Charakter haben, stellt MPLL die notwendigen Werkzeuge bereit, um die Spezifikationen an den individuellen Fall, Kontext und weitere Vorgaben anzupassen.

Die vorliegende Arbeit illustriert den Ansatz, diese Ziele zu erreichen. Sie bestehen darin, eine Spezifikationssprache für geographische Szenarien zu schaffen, die die angegebenen Eigenschaften erfüllt und in ein erweiterbares System eingebettet ist. Hierbei liegt die Betonung auf *geographischen* Szenarien. Die typischen Probleme betreffen Menschen, und werden von ihnen in entsprechendem Umfeld bearbeitet. Darunter fallen z.B. Routenplanung innerhalb und ausserhalb von Gebäuden, räumliche Relationen, sowie die Interaktion mit realen Objekten.

Contents

List of Figures

List of Tables

1. Introduction and Motivation

In recent years, the importance of geospatial information has substantially grown in several aspects. There has been a shift from special purpose Geographic Information Systems (GISs) towards general purpose applications for end users. Geospatial data in general is more readily available to a wider audience on the web. Regarding both quality and type, a wider range of data can be accessed by a greater number of people in different application scenarios. This development has in part been fuelled by the availability of more affordable and more powerful stand-alone and mobile devices. Especially the availability of the latter triggers new impulses in research, for example in the field of navigation and way finding, location based services, and other areas which can utilise spatio-temporal data. As the web is concurrently evolving to the Semantic Web, the meaning of spatial information on the web is also becoming increasingly important, because the web represents the major source of information – semantic or not.

Quantitative Data

Traditionally, GISs operate on complete and highly accurate spatial information [148], such as cadastral plans, data for construction and maintenance of different parts of the infrastructure, land survey. GISs are very suitable for processing, integrating, and analysing this kind of *quantitative* spatial information. Processing quantitative data in this context refers to quantitative calculations in discrete space. By definition, quantitative data can be measured by discrete scale: 15 ft., 24.37 m^2, 272°, (48° 08′ 58.69″ N, 11° 35′ 38.33″ E). In general, it is recognised that quantitative approaches are not good for representing human cognition [95]. Basic Euclidean concepts are also not sufficient.

Qualitative Data

Whenever humans interact with their environment, they usually do so in a *qualitative* manner. Humans are not well equipped to handle exact measurements of direction or distance, or other quantitative expressions, such as those stated above. They tend to use relations between spatial entities, often focussing on relations between their own position and entities called landmarks (e.g. *"the petrol station"*, *"the TV tower"*, *"the city centre"*). Human cognition further involves qualitative angular and distal expressions, such as *"in the direction of ..."*, *"behind ..."*, *"along ..."*, *"near ..."*, *"until you reach ..."*. The underlying continuous numeric structures (angles, coordinate systems, metric distances)

are translated into small sets of symbols or labels on an ordinal and nominal scale in discrete space. The area around the numeric position (48° 08′ 58.69″ N, 11° 35′ 38.33″ E) is represented by an expression such as *"University of Munich, Institute for Informatics"*, *"left"* means something around 270°, a kitchen of 24.37 m^2 can – by the standard of kitchens in the city of Munich – count as a *"rather spacious"* kitchen, and the distance of 15 ft. between a mail box and the bakery on the corner can be regarded as *"near"*. The reader will notice the vagueness, ambiguity, and imprecision in these statements. We will duly go into detail at a later point, mainly in course of section 2.7, about the difficulties for computation regarding these kinds of fuzziness. Furthermore, human communication about spatial issues, as well as a substantial part of resources of spatial information, is mostly incomplete and fuzzy, and suffers from ambiguities in interpretation. This fact can be easily confirmed by looking at basically any data source containing spatial expressions in natural language, such as news articles, route descriptions, or sources of similar kind. Unfortunately, incompleteness and imprecision are two properties which GISs cannot handle very well.

In the scope of this work, the distinction between quantitative and qualitative data is crucial. On the one hand, computer systems are perfectly suited for processing quantitative data, but humans are used to and – by nature – equipped for handling qualitative data. On the other hand, quantitative methods are well suited for certain problems, whereas qualitative methods are well suited for other problems. The former cannot be used with incomplete or imprecise (or purely qualitative) data, the latter are usually only useful for certain confined domains and are not generically applicable to real world scenarios due to the required grade of abstraction.

Spatial Semantics

For computer science in general, and artificial intelligence in particular, the semantics of spatial information is not a trivial issue. Due to the differences between human spatial cognition, reasoning, and communication, and the computerised counterparts, a number of difficulties persist. While the qualitative way in which humans reason about space enables them to solve all kinds of spatial problems in their environment, the mostly quantitative mechanisms of computerised systems often fail to achieve the same results in a similarly elegant manner. Computers have difficulties to "understand" spatial notions and expressions such as *"near the University"*, *"along the river"*, *"in the southern part of the city"*, *"behind the post office"*, or *"on the shelf next to the sugar"*.

MPLL

The Multi-Paradigm Location Language (MPLL) presented in this work aims at bridging the gap between quantitative and qualitative spatial models, representations, and information processing techniques, in the sense of providing a means for humans to individually specify the way they understand space, and a means for machines to apply quantitative and qualitative methods to these specifications. In other words, spatial notions specific to human cognition, which are contained, for example, in queries to a database, can be interpreted by the computer *in the intended sense*. If the user wants to find, for example, the *nearest* pharmacy, he/she is not looking for the shortest metric distance, but for the shortest way regarding travel time. This particular problem amounts

to a route planning problem which depends on the individual user and context. The user might, for example, sit in a car during daytime, or walk on foot along a street after opening hours. Even particularly exotic cases can be handled by MPLL, for example if the user prefers completely different spatial notions, such as *"inland/seaward"* instead of the cardinal directions *"north/south/east/west"*, because he/she is influenced by the peculiarities of oceanic languages.

Creating Synergy

Other problems require equally specific, but different, techniques. Sometimes techniques involving multiple spatial paradigms have to be used in connection with each other in order to find specific solutions. One fundamental assumption of this thesis is that there does not exist a single correct and best way to process spatial issues in their entirety. On the contrary, solving a broad range of problems sometimes requires outright application of a combination of complementary methods, representations, and models.

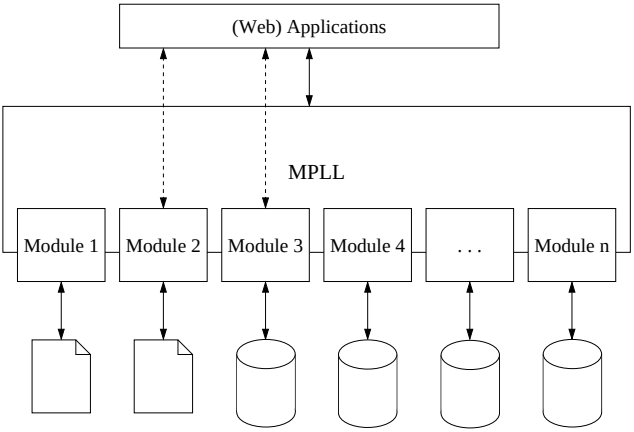


Figure 1.1.: System Architecture Overview

A preliminary overview of the basic system components is shown in Fig. 1.1. MPLL serves as a central component which provides a unified way to specify, process, and exchange spatial data. To this aim, MPLL employs the functionalities of several extendable *modules*. The modules usually have access to specific (web) data sources (e.g. documents, data bases) and provide very specific services. These services include, for example, routing and wayfinding in graph structures, processing of coordinates and access to different geospatial reference systems, access to data streams including transformation and filtering mechanisms. External applications can utilise the combined functionality either through the Application Programming Interface (API) or via other (web-) interfaces.

The language GeTS, a Specification Language for Geo-Temporal Notions [126, 123], provided a framework for the development of MPLL. Like its temporal counterpart, MPLL is not intended as a general purpose programming language, although it has many features of a functional programming language. However, it is a specification language for spatial notions with a concrete operational semantics.

MPLL offers a number of basic constructs and data types which facilitate expressions such as *"location A lies between location B and C"* but also *"the application shortcuts on the screen are located on the lower left, next to the Desktop Preview & Pager"*. More complex notions can be represented by the composition of basic constructs, as for example the notion *"between"* can rely on the *"bearing"* construct providing the bearing between points in space. However, another *type* of *"between"* pertaining to network routing could be defined by using a shortest path algorithm and by comparing path lengths and/or nodes traversed along the shortest path.

Fuzzy Notions

Since imprecision, uncertainty, ambiguity, and vagueness are a part of human spatial cognition, the possibilities of using fuzzy notions are evaluated as well, in order to overcome the inherent difficulties quantitative models display in connection with these issues.

The next section lists some examples which illustrate the problems encountered in this kind of spatial information processing, followed by a compilation of possible solutions. Subsequently, an overview of the system architecture is presented, showing the layout of the different components and how they work together to provide solutions for the problems stated before.

1.1. Examples

In this section, a few examples are laid out to illustrate the typical problems which are to be faced when querying and processing spatial information. The list of problems illustrated by these examples is by no means exhaustive, but it serves well to illustrate its diversity and the problems' specificities. In a similar manner, the solutions presented thereafter are merely selected alternatives of several different approaches, since the problems presented allow for very different interpretations and solutions.

1.1.1. Data and Queries

Facts and Inference

Example 1.1 *Suppose we have some data about cities, states and countries. Entries could be:*

(1) San Francisco is a city
(2) San Francisco is in California
(3) San Francisco has 3 million inhabitants
(4) California is in the U.S.

A query could be: "give me all metropolises in the U.S.". *How can the necessary information be derived from the given facts?* ■

Route Planning

Example 1.2 *Suppose a database contains the yellow pages entries, i.e. businesses and their addresses. A query could be:* "give me the nearest pharmacy"*, with the spatial context that the user is at a particular location X in the city. Other context information about the user's current situation can include, for example, mode(s) of transport available, age and gender, preferences, abilities.*

This query could be evaluated in a naive way by selecting the pharmacy with the smallest geographic distances between it and the location X. This might be a first approximation, but it can give completely useless results. A pharmacy which is located very close by, but is difficult to reach, may not be a good choice. The pharmacy might be located on the other side of a river and the next bridge could be miles away. Getting someplace might involve going uphill or entering unsafe parts of a city. Factors like these can greatly and very individually bias the notion of distance for a person or group. There is no trivial answer to this query, nor is there only a single "correct" one. ■

Spatial Relations

Example 1.3 *Consider the query* "give me all cities between Munich and Frankfurt"*. What does* "between" *mean here? If we take a map of Germany and draw a straight line from Munich to Frankfurt, it does not cross many cities – if this was the means to decide on this issue.*

In fact, this is a good example of the diversity of possible solutions to a problem, since the notion "between" *is not only subjective, but in addition depends very much on the individual context.* ■

Fuzziness

Example 1.4 *Consider a database with, say, all cinemas in Munich. A query could be* "give me all cinemas in the south of Munich"*. The notion of* "Munich" *itself has no precise boundaries, except artificial ones. However, any artificial boundaries may yield strange results for many users. What about cinemas which are located just beyond city limits? Furthermore, there are manifold interpretations of the notion* "in the south of"*. How can these two statements be combined in order to produce the desired result?* ■

Proximity Relations

Example 1.5 *Suppose a company looks for a building site for a new factory. The site should be* close to *the motorway. What does the notion* "close to" *pertain to in this special case? Due to the semantics of the structure of a motorway it is not a simple point-line relation.* ■

Domain Knowledge

Example 1.6 *Suppose the database contains a road map, together with dynamic information about, say, traffic jams. The information about traffic jams is usually not very precise. It could be something like* "there is a traffic jam on the M25[1] of 2 miles length between junction 8 and junction 10"*. How is a statement like this to be interpreted? Can enough "hard facts" be extracted to actually put this information to use?* ■

[1] The M25 is the orbital motorway around greater London.

Figure 1.2.: A Sample Application for Text Annotation – Web Pages

Example 1.7 *Named entities (e.g.* "Kofi Annan"*,* "Coca-Cola"*,* "World War II"*, see section 2.6.2) are ubiquitous in web pages and other text documents; an example is shown in Fig. 1.2.*

Text Annotation

In the context of this work, we focus on spatial named entities, which include not only officially denominated states, governmental regions, cities, or rivers, but also other places, regions, or spatial features. These spatial entities can be clearly defined (e.g. "the West Bank"*,* "the Mediterranean"*), or in some way imprecise or vague (e.g.* "Northern China"*,* "the Outback"*,* "Ground Zero"*).*

In the following excerpt of a news item, spatial named entities and expressions have been set in boldface. This excerpt serves to illustrate the different semantics and ambiguities of the process.

> *"[...] The troops will be deployed in what U.N. peacekeeping officials described Tuesday as a "rolling exercise" replacing Israeli troops with Lebanese and U.N. troops starting from* ***the northeast*** *[of the occupied territory]* ***at Marjayoun, Lebanon****, and moving* ***southwest****.*
>
> *Once in place, the U.N. troops will work along with Lebanese troops to try to create a* ***demilitarized zone between the Litani River*** *and the* ***"blue***

line" *– the* ***border between Israel and Lebanon****.*

Dozens of countries, including ***Italy****,* ***Malaysia****,* ***Indonesia*** *and* ***Turkey****, have expressed interest in taking part in the force and attended technical meetings* ***at U.N. headquarters*** *– but no country had pledged troops as of Tuesday. [...]"*[2]

Difficulties arise in several aspects. Peoples' names might be the same as those of cities (e.g. "George Washington"*), and in general, some expressions might have multiple (completely different) meaning (e.g. the word* "Bank" *in* "West Bank" *and* "Bank of America"*). Some elements of composite expressions are responsible for the expression's overall spatial quality (e.g.* "U.N. troops" *vs.* "U.N. headquarters"*). Named entities can represent all three generic types of spatial features, i.e. points, lines, and regions. It is very difficult to determine the semantic sense of named entities. It is even more difficult to determine the semantic sense of spatial relations and named entities in natural language.* ■

Route Descriptions

Example 1.8 *Route descriptions should look like the examples presented in section 2.1, pp. 27. They should contain expressions such as* "turn right at the petrol station", "continue straight until you reach the bridge"*, or* "the entrance to the hospital is located on your right hand side, opposite from the church".

How are suitable landmarks identified and what ranking mechanisms exist in order to decide which to incorporate into a route description? What kind of spatial relations (such as "opposite"*,* "near"*,* "behind"*, or* "left of"*) are necessary and how can they be processed?* ■

1.1.2. Sample Solutions

Solutions to the given examples are themselves mostly, well, examples, since there almost always exist multiple ways to solve the associated tasks and subtasks.

Solution 1.1 *One way to evaluate the query* "give me all metropolises in the U.S." *could be:*

- *Formulation of the database entries in a logic based knowledge representation language, for example in the Web Ontology Language (OWL) [161] or its underlying Description Logics (DL).*
- *Definition of the concept "metropolis" in the same knowledge representation language, for example*

$$\texttt{city}(C) \wedge (\texttt{hasInhabitants}(C) \geq 1000000) \mapsto \texttt{metropolis}(C) \quad (1.1)$$

(A metropolis is a city with at least 1 million inhabitants.)

[2] This item was taken at random from a news article on www.cnn.com, on August 16, 2006.

- *Executing a so-called* instance test *for the database entries. The instance test would conclude from (2) and (4) that San Francisco is in the U.S., and from (1) and (3) that San Francisco is a metropolis.*

■

Solution 1.2 *Much better answers to the query* "give me the nearest pharmacy" *could be produced if we used, instead of the geographic distance, a metric which is determined by the available forms of locomotion, for example going on foot, public transport and so forth. Subsequently, the nearest pharmacy would be the one which can be reached at optimal cost, which usually means "in the shortest time". This problem amounts to a route planning problem, which are traditionally modelled using weighted graph structures. The system must compute the shortest route from the location X to a number of pharmacies in the vicinity and choose the one with the shortest route. The route planner must take into account the transport networks (road maps, tram lines, bus lines etc.), as well as the context information about the user's current situation.* ■

In fact, it turns out that in many cases the formalisation of *distal relations*[3] involves the solution of a route planning problem. Route planning is one of the preferred means to model distal relations, if the goal is to express *qualitative distance*.

Solution 1.3 *The query* "give me all cities between Munich and Frankfurt" *can be evaluated in different ways. One possible formalisation of* "between" *(a rather simple one) could be: in order to check whether a city B is* "between" *the cities A and C, compute the shortest route R_1 from A to B, the shortest route R_2 from B to C and the shortest route R_3 directly from A to C. If the extra distance $d = length(R_1) + length(R_2) - length(R_3)$, I need to travel from A to C via B, compared to the direct route from A to C, is small enough, B can be considered to be* "between" *A and B. Since the condition "is small enough" is not very precise, one could use the distance d directly to order the answers to the query.* ■

Obviously, some problems arise from this individual definition of *"between"*. A city B which is either very close to A or C could qualify as being *"between"* the two. Theoretically, B could be located anywhere around A or C, notably also on the respectively opposite side. In these cases human cognition would not render B *"between"* A and C. A possible refinement would be to restrict the lengths of R_1 and R_2 to be roughly equal.

Solution 1.4 *This case can be modelled using fuzzy distributions instead of arbitrary and artificial crisp boundaries. Therefore,* "in the south of" *is a two dimensional fuzzy distribution over the Munich region. The fuzzy distribution could even be non-zero for places outside, but close to the Munich border. Consider the given query* "give me all cinemas in the south of Munich". *This method assigns a fuzzy value to the location of each cinema. The fuzzy values can then be used to order the answers to the query.* ■

[3] In the literature these are also sometimes called *proximity relations*, although in the scope of this work we stick with the expression *distal relations*. There is no semantic difference.

Solution 1.5 *The* "close to" *relation in this example depends on the semantics of a motorway in the sense that there are only designated spots (on-ramps and off-ramps) at which the motorway can be entered or left. Furthermore, this query pertains to a route planning problem, because what is needed is the time it takes for a car or for a lorry to get to the nearest junction of the motorway. The length of the shortest path to the next junction can be used to order the answers to the query.* ■

Solution 1.6 *Extracting usable information from the statement* "there is a traffic jam on the M25 of 2 miles length between junction 8 and junction 10" *depends on the semantics of the underlying domain of Traffic Information System (TIS). Therefore we need domain knowledge about traffic jams and their specific features. A possible approach for modelling the purely spatial aspects*[4] *of this problem could be the following.*

If the M25 is taken as a straight line then the traffic jam is a one-dimensional interval whose location is not exactly determined. Instead, we have some constraints: length equals 2 miles, start after coordinate of junction 8, and end before coordinate of junction 10. This particular problem is well suited to be modelled using Allen's interval relations [3, 64].

Queries like "is there a traffic jam on the western part of the M25" *gives then rise to a constraint solving problem (see, for example, Dechter [40]).* ■

Solution 1.7 *Named Entity Recognition (NER) (see section 2.6.2) is only a subtask in the process of determining the semantics of natural language. Part of the research of Weigel, Schulz, Brunner, and Torres-Schumann [17, 165, 166] deals with the annotation of web pages. An example of their work is shown in Fig. 1.3. This figure shows an annotated version of the web page from Fig. 1.2. On the left hand side there are the concepts and instances, which are linked and highlighted within the annotated page on the right hand side. Although this work does not focus especially on spatial issues, it illustrates the application of ontologies and taxonomies, and their concepts and instances, in the annotation process. A specialised spatial annotation could be done in a similar way.*

This example also displays the limits of pure string matching. Although the names of locations, denoted by green background, are correctly identified as such (there exist cities in Germany called "Born", "Hamburg", "Horn", *and* "Marxen"*), this is not the semantic meaning of the words within the text (which – apart from the correctly identified meaning of* "Hamburg" *– pertains to the verb* "born"*, the music instrument* "horn" *and the last name of Mr. Eduard* "Marxen"*). Obviously, simple string matching alone is not sufficient for thorough annotation and capturing the semantics of natural language. The general problems and issues discussed by Weigel et al. also pertain to the spatial subset of named entities.*

Albeit MPLL cannot solve all problems of NER, it can be used to alleviate some problems in the process by checking and verifying statements extracted from documents. This

[4] This solution does not pertain to any dynamic features of a traffic jam, such as whether the traffic jam is moving forward or backward, whether it is growing or dissolving, the nature of the incident, etc.

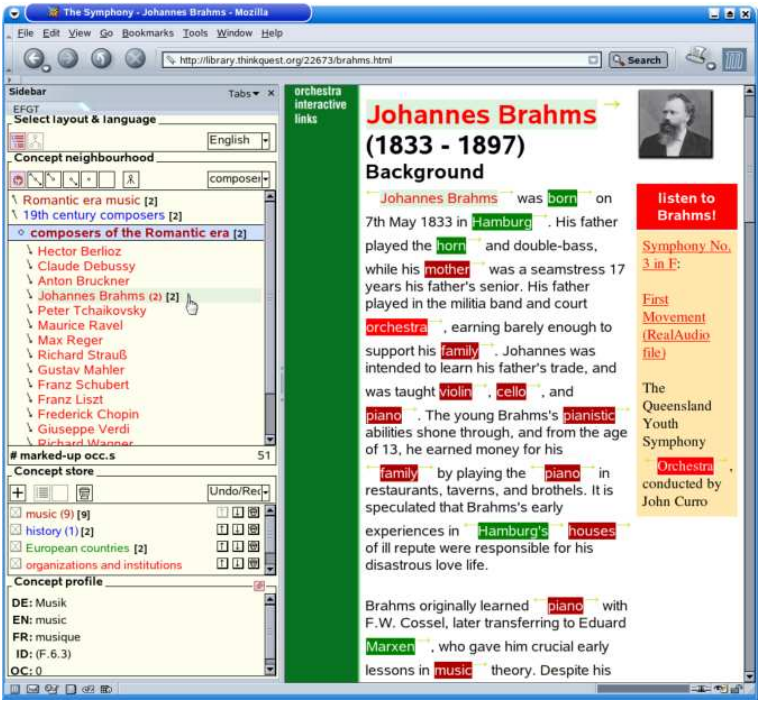


Figure 1.3.: Web Page Annotation with EFGT-Net [166]

can be done, for example, by checking for conflicting statements, or by providing access to a set of predefined and checked spatial entities, facts, and relations. Restricting the processing to a certain domain – for example spatial semantics – could ease the disambiguation of expressions within the context of a document. The correct determination of subjects and objects could be asserted with the help of predicates and vice versa (e.g. "[a location] lies..." *vs.* "[a person] lies"*).*

In fact, there exist two connections to MPLL. First, named entities can be used as landmarks (see sections 2.6 and 2.6.2 respectively) in the processing of spatial relations. Second, within a system providing named entities and relations, such as EFGT-Net, MPLL could aid in identifying named entities denoting locations and verifying their (spatial) semantics. ■

Solution 1.8 *There is no trivial or quick solution for generating natural language route descriptions, such as the ones given in example 1.8, or the ones in section 2.1 on page 27.*

This is a complex process, which involves several different techniques ranging from scene analysis and heuristic methods to natural language generation, to name but a few.

MPLL can be integrated, for example, in the analytic phase to determine useful spatial entities and their relations. Furthermore, it may be used to provide spatial relations which can then be translated to natural (or controlled [62]) language. This section briefly sketches the possibilities without getting into too much detail.

A route description traditionally consists of a series of quantitative positions *(nodes in a graph, associated with a position in space) and* actions *(e.g. "turn* $-84°$*"). To translate these into a human understandable language, these quantitative statements must be assigned a respective quality. The expression "*$-84°$*" could, for example, be translated into* "turn left"*, using a template of cardinal directions (see section 2.5.1). Furthermore, the location, where the action has to be performed, should be unambiguously identified, usually by relating it to a landmark. A not too large set of landmarks can, for example, be computed using a geometric* k-nearest neighbour search*. A pair of coordinates denoting an intersection could then, for example, be expressed as* "the intersection" *(if there is only one in the vicinity),* "at the traffic lights" *(if there are no other intersections featuring traffic lights), or* "the intersection with McDonald's on the corner" *(if this is its salient feature). Note that it could be useful to check first, whether a landmark can be found, which is located* directly *at the position of the action itself (i.e.* "the intersection"*). If this is not possible, landmarks which are* "near" *the action's position need to be related to it (i.e.* "...McDonald's on the corner"*). In order to generate these relations, certain attributes of network nodes and edges are needed, along with sets of landmarks and/or points of interest. Deciding which relations and which landmarks are suitable is, however, non-deterministic.*

The individual issues in preparing suitable route descriptions also depend on the environment. Route descriptions in outdoor environments are different from route descriptions in indoor environments. The types and forms of landmarks typically vary significantly. Remarkably, outdoor environments usually contain more artificial than natural landmarks (see distiction in section 2.6). ■

MPLL addresses the associated issues and provides means for processing spatially relevant information. Spatial relations are, for example, not only hard coded, but can be defined according to the context and according to the individual user profile. For example, the expression *"near"* has a very different interpretation when the user is sitting in a car or bus, than when the user is walking around the city on foot. Many other relations can be – or need to be – individually defined to be usable in a certain scenario.

1.2. Modelling Techniques

In several research areas, methods have been developed which can help solving problems in "spatial information processing". Since spatial information processing is an extremely

broad notion, which has connections to many areas in computer science, we can only mention here a few ones.

On the very concrete side there are GISs, i.e. databases and algorithms which deal with concrete geographical data, road maps, land coverage etc. The GIS techniques depend on the availability of concrete coordinates. If coordinates are not available, symbolic data representation and information processing is necessary. A broadly accepted symbolic spatial reasoning system is the Region Connection Calculus [30] (see section 2.5.3). It conveys the ideas of Allen's interval calculus [3] from the one-dimensional case to topological spaces. RCC-8 provides basic relations between arbitrary regions and rules for reasoning with the relations.

'Shortest path' algorithms have been developed to solve the path planning problems, for example in transport networks. The path planning problem in a concrete two- or three-dimensional environment is one of the robot navigation problems, and there are a number of more or less practically useful algorithms to solve it [68].

The 'shortest path' algorithms do not take into account context information about the traveller. It is a totally different situation if the traveller has a car available, or if he depends on public transport systems. One way to use context information in a shortest path algorithm is to construct a problem specific graph for the shortest path algorithm. For example, if the traveller has a bicycle, the system might construct a graph consisting of paths and roads, together with those railway and bus lines where a bicycle can be taken into the coaches.

A very general knowledge representation and reasoning technique are the Description Logics (DL) [8], with OWL as its semantic Web version [161]. In DL one can define so-called 'concepts' which correspond to sets of objects, and one can relate individuals to the concepts. Formula (1.1) on page 7 is an example of a concept definition in a DL.

The following sections illustrate some techniques for modelling spatial information, each with their own inherent advantages and disadvantages.

1.2.1. Graphs, Graph Transformations

As previously stated, the solution of a number of problems involves spatial data that can be represented as graphs. Different problems might require different kinds of graphs, but these are often closely related to each other. Therefore, a huge amount of modelling techniques, tools, and algorithms pertaining to graph theory, which were produced by the research community over the past decades, can be employed to solve spatial problems.

Any spatial data that can be reduced to some form of network is predestined to be represented by one or more graphs. The applications range from purely symbolic structures on the abstract end (jurisdictional or administrative regions of a country) to very concrete physical networks on the other end (street networks, public transport, buildings).

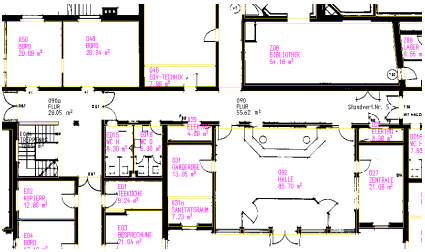
The main application of graphs is finding certain paths through the graph which satisfy particular constraints. Typically, these shortest paths are used to find the *quickest*,

shortest, or *least expensive* path, although there are no restrictions. Optimisation essentially depends on a suitable cost function and the formalisation of the desired input. Basically any numeric values that can be stored in the graph and processed by a suitable cost function can be used.

The following examples illustrate graph representations of concrete indoor and outdoor networks. The idea of graph transformation and different levels of detail is shown in connection with the outdoor example. The use of graphs in modelling more abstract structures is illustrated later in this chapter, in connection with Region Connection Calculus (RCC)-8 relations (see section 1.2.3).

Example 1.9 (Floor Plans) *Indoor navigation of autonomous vehicles, i.e. robots, requires a detailed floor plan, as shown in Fig. 1.4 (1). For guiding human users through buildings, for example from the entrance to a particular office, such a detailed floor plan is not necessary. A simplified net plan, such as shown in Fig. 1.4 (2) is much more suitable for this purpose. The simplified plan can be generated from the detailed floor plan.* ■

(1)



(2)

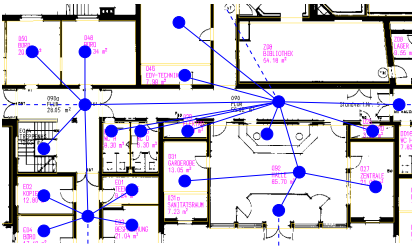


Figure 1.4.: Floor Plan without (1) and with Network Overlay (2)

In a hierarchical system of graphs, one can collapse a whole building to a single node in a graph on an upper level, for example a graph representing a map containig several

buildings, subsequently parts of the city and the street network, and so on. Path planning on one of the upper levels concerns a different scale (e.g. building to building, not room to room). Concrete data about the internals of a building are, therefore, not needed at upper levels. Hierarchical systems of graphs are a key feature of a holistic and multimodal [155] approach of path planning in multiple interconnected networks.

Another important issue is graph generation and transformation. Not only can different graphs be connected in a major effort of path planning, but also there is a need for accessing different levels of detail within a single graph structure. Different levels of planning require different levels of detail in the underlying data structures.

Planning a trip by car from Munich to Hamburg, for example, requires a less detailed representation, than, say, the generation of so-called driver-level instructions [156], on the same route such as *"take right lane"*, or *"take next exit to get on the highway A9"*.

The necessity of graph transformation partly derives from these requirements, as graphs of lower detail should automatically be generated from graphs of higher detail. Due to semantic leeway, this process cannot easily be automated, at least not fully. Another motivation comes from the formats of available data. Data about road construction and maintenance has been dominated by specialised GIS applications for some time now. Therefore, data about basically the complete street network of a nation is waiting to be put to use in applications such as path planning. Unfortunately, these data are overly precise and held in different proprietary formats. Since extraction of abstract graph information about the underlying networks is widely considered to be rather difficult and largely hindered by these proprietary standards, companies specialising in navigation data [115, 153] did resort to other measures. They collect data in a huge effort by independently charting the street networks using specially equipped vehicles. The following example illustrates the discussed issues.

Example 1.10 (Road Crossings) *Fig. 1.5 shows a detailed representation of an intersection of two streets, including an underpass (dashed lines) and pedestrian pathways (shown in red). This graph is suitable for guiding an autonomous vehicle through the intersection. Note that traffic regulations are encoded implicitly, i.e. turning left coming from node 1 going to node 6 is not allowed, due to the missing (directed) edge between the nodes. The same manoeuvre going from node 8 to node 2 is ok, as reflected in the connection between node 8 and 2. Even simple routing algorithms can quickly produce an alternative for the transition from 1 to 6: the traversal via nodes 1, 2, 3, 4, 5, and 6.*
A simplified version of this crossing is shown in Fig. 1.6. It contains enough information for a standard navigation system, i.e. basic connections between nodes, which indicate the necessary order of traversal.
Finally, one can collapse the whole road crossing into single nodes of the road network as shown in Fig. 1.7. This is sufficient for planning on a larger scale. In all three figures we see the same road crossing, but at a different level of detail.

The examples illustrate several observations:

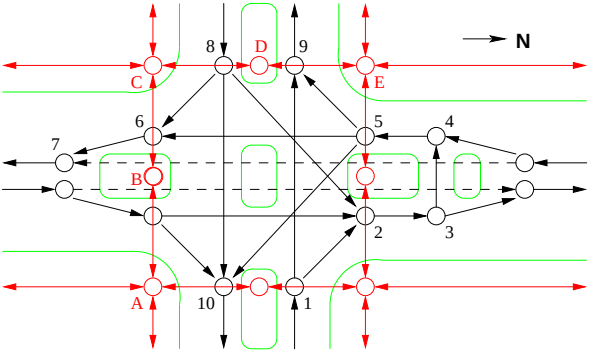


Figure 1.5.: Detailed Road Crossing

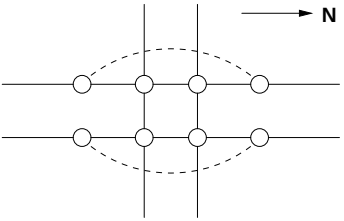


Figure 1.6.: Schematic Road Crossing

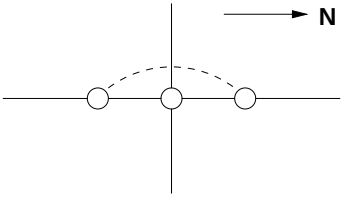


Figure 1.7.: Abstract Road Crossing

1. There is a hierarchy of graphs. At the lowest level there are graphs with the concrete geographical details which are necessary for, say, guiding autonomous vehicles. At the highest level there are graphs which represent logical relations between entities.

2. There are correlations between the nodes and edges of the graphs at different levels of the hierarchy. These need not be a one to one correspondence. Usually a whole subgraph of a lower level graph corresponds to a single node or edge of the higher level graph. A typical example is the representation of the city of Munich in Fig. 1.8, as a polygon in the left hand graph and as a single node in the right hand graph.

3. A transition from a lower level graph to a higher level graph can be facilitated by identifying specific structures in the lower level graph, and transforming them into structures of the higher level graph with the same meaning. In example 1.10

this structure is a road crossing. In example 1.9 these structures are floors, doors, rooms etc. In example 1.11 these are cities, states, etc.

These structures are, in general, part of an *ontology*. In parallel with the development of the graphs, we therefore need to develop the corresponding ontologies. The elements of the ontology are the anchor points for controlling the graph transformations and for choosing suitable graphs to solve a given problem.

4. It is in general not a good idea to put all information into one single graph, even if it is information of the same level of detail. In a typical city we have, for example, a road map as a graph, the bus lines as a graph, the underground lines as a graph etc. We therefore need to consider collections of graphs with transition links between the graphs. Typical transition links between a road map and an underground map are the underground stations. The transition links, can, however, be little graphs themselves, for example the network of corridors and stairs in an underground station.

5. The graphs at the higher levels of the hierarchy can and should usually be extended with additional information which is not represented in the lower level graphs. For example, the graph in example 1.11 with the symbolic information about cities and states can easily be extended by adding further cities and states.

1.2.2. Ontologies

In mathematical theories of space, points are traditionally considered as the basic spatial entities [31], whereas lines and polygons are merely defined as sets of points. However, there are fields in which regions of space are the preferred ontological primitive [160]. Furthermore, there exist several alternatives regarding the type of the embedding space. Depending on the application, continuous, discrete [49], finite [67] or non convex space might be best suited.

Apart from these fundamental ontological questions, a very different use of ontologies presents itself on a higher level. Many application domains (e.g. transport, land survey, electricity or water supply) come with their individual domain concepts, vocabulary, and relations, which are best modelled in an ontology. The concepts found, for example, in the Ontology of Transportation Networks (OTN) (see section 6.3.1) reflect the real world of transport networks with the inherent properties and different modes (pedestrian, car, bus, subway, etc.). Different processes, such as routing and navigation, visualisation, and reasoning, can make use of the instance data according to the ontologies specifications. This approach is very flexible, for example with respect to changes in the data structure. Should a formerly not known type of street be added, for example *one way street*, the visualisation need not be changed immediately, since the new class is a subclass of the class *street* (the visualisation of which is known). The relevant visualisation rules could

be altered manually to fit the new class. Likewise, a procedural definition of the rules could automatically adapt to the new classes properties.

1.2.3. Region Connection Calculus

The fundamental approach of the Region Connection Calculus (RCC) is that, instead of extensionless points, *regions* of space are used as basic entities, and that the only basic type of relation between these regions is that of *connection* – and its variants [30]. The different kinds of connection are described in more detail in section 2.5.3, on page 58, along with a general discussion of RCC-8. The following example shows a possible application of RCC-8 in symbolic knowledge representation.

Example 1.11 (Symbolic Data Representation) *This example illustrates the transition from GIS style data representation to a pure symbolic knowledge representation.*

Fig. 1.8 shows the boundaries of two of the German states and cities on the left hand side. Traditionally, the boundaries can be represented as polygons stored in a GIS. On the right hand side, the polygons are collapsed into single nodes of an abstract graph. The relation "polygon Munich is contained in polygon Bavaria" *is turned into an NTTP relation (Non Tangential Proper Part, see section 2.5.3) in the abstract graph. Likewise, the relation* "polygon Bavaria touches polygon Hesse" *is turned into an EC relation (Externally Connected).* ■

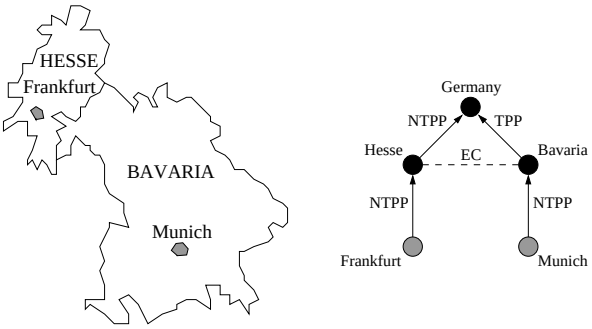


Figure 1.8.: Symbolic Data Representation

RCC-8 is a well developed and broadly accepted theory for topological modelling. Its application in solving certain spatial issues is very sensible, albeit the complexity of computation could become a problem in some cases. For example Jochen Renz [137] has proved that reasoning with RCC-8 is NP-hard by showing that reasoning with a subset of RCC-8 is already NP-hard.

1.2.4. Hard Coded Spatial Functions

Consider the spatial relation *"between"*. How many different interpretations of this notion are there? Here are some suggestions regarding point entities:

Ordering Relation – House number schemes are usually ordering relations. If one is looking for a particular address in a street, this order relation is very helpful, because, say, the number 1500 usually lies *between* 1000 and 2000. This example raises a question, though. While both numbers 1999 and 1001 are, mathematically, perfectly *between* 1000 and 2000, the houses with these numbers, would intuitively not qualify as being located *between* the houses with the numbers 1000 and 2000. More likely, the ternary relation *"between"* would be replaced by a binary relation like *"next to"*. This is because an intuitive interpretation of *"between"* most likely requires the *referent* to be located *halfway* between the *relati*[5].

Network Based – Also in this case there are several possibilities of defining the notion *"between"*.

In a network of street segments (edges) and junctions (nodes), a junction A is located *between* two other junctions B and C, if A is part of the shortest path between B and C.

A more exact definition (analogous to the "order relation" before) could include the restriction that the number of nodes between A and C should be not too different from the number of nodes between A and B.

Yet another variant would allow other nodes, which are close to the shortest path (that is, ideally, close to A), also to qualify as being *between* B and C.

Geometric – Determining objects in the line of sight, for example, requires yet another definition of the notion *between*. If the difference in *bearing* from a point A to points B and C amounts to π (or 180°), then the object A is geometrically between B and C (to keep it simple, we stay in planar space and don't regard elevation).

In all three cases, a crisp definition could possibly lead to undesired results. The use of fuzzy logic could alleviate this issue. These matters are further complicated if other entities besides point entities are introduced. However, some issues regarding lines and regions can most likely be solved in a similar manner as in point-point relations.

We see from these examples that it is quite impossible to design and implement a programming library, which holds the implementation of *all* possible interpretations of the notion *"between"*. There will always be some issue in some scenario, which is not covered by the library – not to mention the consequential problems with the interface, which would certainly be very complex and heavily overloaded.

[5] See section 2.2 on page 32 for a definition of *referent* and *relatum*.

1.2.5. Predicate Logic

Symbolic data about spatial relations can be specified as axioms in predicate logic. Such axioms could be, for example,

(1) *the vehicle "X" is located within the city of Munich*,

(2) *the city of Munich is part of Bavaria*, and

(3) *the radio station "Bayern 3" can be received throughout Bavaria.*

Subsequent reasoning on sets of such axioms can, for example, prove the statements to be consistent (or inconsistent), or produce statements such as *the radio station "Bayern 3" can be received with the radio in vehicle "X"*. An article [150] by Mark Stickel, for example, provides more detail on such issues.

The problem with reasoning on a symbolic level is that real world scenarios often feature complexity which is not fully covered by the underlying logics. Furthermore, qualitative data, which is required for such reasoning, is much harder to come by than quantitative data. Of course, some qualitative information can be derived from quantitative data sources, but the resulting quality varies. In many cases qualitative data of, well, good quality, is the result of extensive manual processing.

One possibility is to reduce the complexity of the real world data to fit the abstract model. However, this often limits the possibilities of the subsequent reasoning on the model, and results are often not very practical. Another possibility is to extend the model to fit real world requirements, although this often leads to problems with complexity of computation.

1.2.6. Spatial Specification Language

As we have seen from the examples above and from the previous sections on modelling techniques it is not possible to provide a spatial model which satisfies all possible requirements in any application scenario thinkable. On the contrary, each individual case comes with certain requirements and many spatial issues are prone to several equally justified interpretations.

One possibility to counter these effects is the use of a *flexible* specification and programming language. In order to provide solutions for problems in many different applications scenarios, the basic types and constructs of such a language should be largely sufficient. Whatever special functions and relations are missing from the language should be possible to be expressed using available language constructs. Already defined functions should be possible to overload or overwrite.

In the case of MPLL, which aims to be such a specification language, these requirements are fulfilled. A certain interpretation of, say, the notion *"between"* can be specified according to the individual requirements and used as any other function provided by MPLL. In most cases, the implementation need not be changed to add new functional-

ity, only some additions to the standard library (or any other library for that matter) are needed.

Existing Languages

However, one could argue that it would be possible to extend an existing programming language with spatial types and functions, instead of designing a new language from scratch. This way, generic functionality would not have to be implemented once more and generic concepts of existing and proven languages could be used.

The most important reason for specifically not using an existing programming language is that the specification language shall not be dependent on any single host language. If the functionality of the language was implemented as a library usable with, for example, C or C++, it would not be possible to use it with other programming languages, such as Java, Prolog, SML, or Haskell. The library would always be tailored to the host language.

Another important issue is the separation of specification and implementation. Designing the specification language from scratch ensures that it is designed as a specification language with all inherent features of such a language. There is no need to compromise between the generic programming features of the language and the specification features and spatial types and functions of the language.

1.2.7. Conclusion

This introduction made clear that in the field of spatial information processing synergy between multiple processing and reasoning techniques is needed. Applications in this field have to deal with a broad range of tasks, each requiring certain distinct data models and specifically suited processing and reasoning techniques.

Quantitative Methods

Advantages of quantitative methods include the fact that they have been well researched and are in widespread use for several years now. The majority of available data are of quantitative nature and there are clear formalisms available for processing.

Disadvantages include issues regarding the high number of highly different and proprietary standards and data formats and the inherent difficulties of applying quantitative methods to incomplete, vague, or imprecise data. Additionally, quantitative methods are very different from human spatial cognition and reasoning.

Qualitative Methods

The advantages of qualitative methods inversely reflect the properties of quantitative methods. They are akin to human spatial cognition and reasoning and they are specifically well suited for tasks involving incomplete or imprecise data.

On the downside, the mediocre availability of qualitative spatial data has to be mentioned, as well as the issues concerning difficulties in interpretation and formalisation of qualitative data.

Qualitative data can mainly be obtained from the following sources:

- **explicit** – data are (manually) generated and stored, for example in a relational data base

- **inferred** – inference over existing data sets can be used to generate additional data
- **derived** – in cases where quantitative data is available, and whenever qualitative data can be transferred to (and from) the qualitative domain, quantitative processing can be used to generate new data

As a consequence, a suitable geospatial information processing system should

- facilitate the use of different processing and reasoning methods, for example in a modular architecture
- be very flexible to accommodate different interpretations and user definable spatial expressions, notions, and functions
- provide means to facilitate and ease the transition between qualitative and quantitative information

The following section briefly sketches a possible architecture and some of the necessary components which allow for an integration of a number of different processing and reasoning techniques using a spatial specification language as a unifying component.

1.3. The System Architecture at a Glance

As shown in Fig. 1.1 on page 3, MPLL serves as a central component which provides a unified way to specify, process, and exchange spatial data. To this aim, MPLL fulfils several functions.

First and foremost, MPLL provides a number of basic constructs and operations, as well as basic data structures for spatial data. It facilitates the specification of spatial entities, such as points, lines, and polygons, and provides a basic set of spatial operations, which include different spatial relations, manipulation and transformation, and algorithms. This way, spatial models can be constructed, which consist, for example, of points (the user's position, points of interest, junctions), lines (streets, rivers, borders), and regions (natural or artificial boundaries, parcels, regions). Specific entities within this model can then be manipulated (e.g. (continuously) changing the user's position, as he/she moves around) and spatial relations can be processed: Which way is the user facing? Which direction does he/she have to go? How can this direction be communicated?

With the tools provided by MPLL, a number of spatial issues can be tackled. However, a basic predefined set of operations cannot be suitable for all possible problems. Therefore, MPLL also provides mechanisms for constructing individual functions by combining basic functions into more complex ones (see, for example, the different definitions of *"between"* in section 2.5.4, pp. 60). As soon as functions other than the ones provided are required, these can be defined accordingly.

Furthermore, MPLL acts as a single interface for several services which otherwise would have to be accessed separately from the host application. With a single connection to MPLL, therefore, an application can access not only the built-in functionality of MPLL, but also the functionality offered by other services, which are in turn accessed by MPLL. Some of these services are briefly described in the following paragraphs (see section 3.2 on page 84 for more details).

A number of issues in spatial information processing require the use of specific data structures and algorithms, which cannot and should not be integrated into a general purpose specification language. Examples for such issues are graph based mechanisms (network related problems, such as path planning), stream processing systems (for data streams, such as traffic or weather information), or special purpose database systems (e.g. directories, web search engines). Each specific issue can be covered by one or more (web) services, which can be accessed by MPLL via standardised interfaces. Some of these services are specifically designed for serving MPLL, for example a service providing the data structures and algorithms necessary for processing data from different reference systems.

1.3.1. A Sample Application

The inter workings of the different components are best explained by once again going over one of the previous examples from section 1.1: The query *"give me the nearest pharmacy"*.

Producing answers to this query requires several different steps, which partly can be accomplished by using MPLL. The following paragraph briefly sketches a series of subtasks, into which the problem can be broken down, followed by a more detailed description, including individual possible solutions.

1. derive the **user's current position** by query or context
2. derive the **context information** relevant for route planning
3. **match the term "pharmacy"** to a category of services in a directory, e.g. the yellow pages
4. **translate addresses into geospatial coordinates**
5. preliminarily **order addresses** (coordinates) according to their geometric distance to the user's position
6. **perform** a series of **shortest-path planning** processes, according to the parameters collected so far, and
7. **order results** (pharmacies), shortest first

8. **provide** suitable **routing instructions** depending on the results (paths) from the path planning process

The following enumeration gives more detail on what has to be accomplished, provides a possible solution, and states, if applicable, the role of MPLL in providing this solution.

1. The user's current position can either be derived from the context (i.e. it is provided implicitly by the device) or it can be queried and manually set by the user. If it is manually set, most likely an address must be translated into numeric coordinates.

   Translating a postal address into geospatial coordinates is a typical example for a (web) service, which can easily be accessed by MPLL. Although MPLL is not directly needed for this task, it can be accomplished through MPLL. Since coordinates are anyways subsequently used by MPLL, this step can well be directly done via MPLL. Deriving numeric coordinates from a device is a simple case of exchanging data and does not include MPLL.

   Usually, the actual task involves a database which provides symbolic (street names, house numbers, zip codes) as well as geospatial (coordinates of junctions) data. If the coordinates of both ends of a street segment are available, along with the street's name and the distribution of house numbers, the approximate position of an individual address can be calculated.

2. A huge amount of information can be relevant for the planning process later on, much of which can be collected from the context and user information. Primarily, this includes the user's position and time and date of the query. Other information can include, for example, mode(s) of transport available, age and gender, preferences, abilities.

   For this example we can assume that the user's position has been determined by GPS and that the user is male, has a monthly ticket for the public transport system, and is capable of traversing stairways (i.e. he is not in a wheelchair, nor has a trolley or pram).

   Like before, determining information of this kind has no real connection to MPLL, as long as the necessary input is given to MPLL.

3. For this subtask, at least two kinds of data are required. First, the term "pharmacy" must be related to some form of categorisation. Ideally, a huge number of concepts is stored in an ontology, so that, for example, different kinds of names (e.g. *"the chemist's"*, *"drug store"*, *"pharmacy"*, *"apothecary"*) for more or less identical concepts can be handled. If this categorisation is problematic, or disambiguation is needed, the user might be able to give a more concrete parameter, possibly with support from the ontologies' concepts.

The second type of database required for this subtask must provide instances that match the concept from above. A directory, such as the yellow pages, holds entries of all kinds of shops, restaurants, hotels, etc. All the entries that fit the category in question are then put into a result set, along with supplementary information (e.g. name, address, phone). If the database also holds geospatial coordinates, the next step is obsolete.

4. If the result set from the previous step contains only addresses, a translation into geospatial coordinates (analogous to 1.) is necessary.

5. If the result set contains a huge number of possible destinations, preliminary sorting by geometric distance might become necessary. This also depends on the time needed for the subsequent path planning. Suppose the result set holds some one hundred destinations, it sounds reasonable, to take a look at the nearest (geometrically) 25 destinations first. Computation of geometric distance is quite inexpensive an can be done quickly.

   At the end of this step, the result set holds all destinations matching the search criteria, sorted by geometric distance, nearest first.

6. The path planning algorithm can now start to process the list of destinations. Ordered by increasing Euclidean distance, several routes from the position given at step 1. to the respective destination from the result set is computed. This, in turn, produces a new result set containing the individual routes.

   In this step, the user and context information is crucial. The path planning process operates on a graph which is constructed from several independent and/or overlapping real world networks: streets, pedestrian pathways, public transport systems (e.g. busses, trams, subways), and so forth. Following the example given in step 2., this means in particular the possibility to use public transport at no additional cost, and the possibility to use stairways. If the user were to use a bicycle, certain types of streets cannot be used and therefore cannot be included in the planning process. Likewise, if the user is equipped with a pram, some kinds of pathways (primarily stairways) cannot be used and must be excluded from planning.

   In this example the overall process shall not be complicated by an overly complex cost function. Therefore we assume that the user does not care about cost or other factors, such as finding very simple routes, or very convenient or scenic ones. The goal is to reach the destination in the shortest time possible.

   In order to conduct this processing, MPLL makes use of a suitable path planning service, such as TransRoute, described in more detail in section 3.2.2 on page 86.

7. In this trivial step, the resulting routes are sorted according to the cost criteria, in this example the travel time. The shortest routes and some additional key information is presented to the user for selection.

MPLL mediates between whatever services had been accessed in the process and the client (application).

8. To derive usable route descriptions from a shortest path within a graph structure requires additional processing. Once the user has chosen a route to his/her satisfaction, the desired output is a human readable route description. It contains not only the bare route, but also trigger cues, reassuring cues, additional landmarks, and some more information.

   MPLL provides the means necessary for producing such route descriptions. Suitable landmarks have to be found and integrated into the route. A directory of landmarks can, for example, be queried and results can be selected by their proximity to decision points contained in the route. Spatial relations between these points along the route and the respective landmarks can be used to generate spatial expressions which the user can understand and relate to (e.g. *"turn left at the gas station"* or *"leave the bus at the second stop after you cross the river"*).

1.3.2. Issues not Covered

Some aspects of spatial information processing could not be addressed within the scope of this work, due to their complementary nature and the enormous added complexity. First and foremost, this concerns comprehensive *context and user modelling*. Both context modelling and user modelling constitute an important but also very broad and complex field in the research community [5, 2, 74, 75, 73, 9, 1]. Regardless of its importance, it was clear from the very beginning of this work that there were only limited possibilities to conduct research in this field, and to present a suitable model which could be integrated into the MPLL architecture. Therefore, it was decided to reduce the effort directed into this field, and to provide the necessary minimum of functionality to deal with issues regarding context and user preferences. This is discussed in more detail in section 2.8, some suggestions pertaining to future work are laid out in chapter 7.

Other fields, such as routing and navigation, qualitative spatial reasoning, or ontology reasoning, had been under extensive research in the past. Therefore, proven mechanisms and techniques, if not already available, are at least on their way. Concentrating on any of these issues would violate the third rule of successful scientific research, according to Edsger W. Dijkstra [44]. The modular architecture of MPLL allows for integration of available mechanisms, albeit each new module requires a suitable interface and protocol.

1.4. Outline

This thesis consists of seven chapters and three appendices. Whenever necessary, cross-references between chapters and sections are made explicit. Generally, the chapters are built upon each other. The outline of the thesis is as follows.

Introduction Chapter 1 is this introduction. By illustrating practical problems in spatial information processing, showing their broad range and the variety of possible approaches for their solution, the purpose of this thesis is made clear. It contains a non-exhaustive overview of modelling techniques, and briefly sketches some aspects of their application. Here, also a short overview of the system architecture is given, and some issues which could not be covered within the scope of this work are listed.

Basic Concepts The basic concepts of the underlying domain of geospatial information processing, which are reflected by the basic types and functions of MPLL, are laid out in chapter 2. Here, an introductory example establishes the most important concepts. Basic data types and spatial relations are discussed at length. The key concept of landmarks is laid out in detail, along with other concepts, such as reference systems and fuzziness.

System Architecture Chapter 3 contains an overview of the system architecture and describes a number of modules which provide functionality around the central component MPLL. It illustrates the framework in which the individual modules and components reside. A short overview of related projects, which are potential candidates for modules or services, can also be found here.

MPLL Chapter 4 introduces fundamental design aspects of MPLL and shows the relation to its temporal counterpart, the Specification Language for Geo-Temporal Notions (GeTS). This chapter further contains the complete MPLL specification, including basic types and functions, and the complete MPLL Standard Library.

Application A short evaluation of the application of MPLL by means of a detailed discussion of suitable and practical examples is the content of chapter 5. Here, typical MPLL constructs and the underlying processing is shown.

Related Work Related work, primarily pertaining to qualitative direction and distance, is described in chapter 6, along with a summary of related projects, diploma theses, and project theses, which have been carried out in connection with this thesis. A short introduction to related standards can also be found here.

Conclusion A subsumption of results and a discussion of future work, along with concluding remarks, is contained in chapter 7.

The appendix, consisting of three chapters, contains a complete language reference (A), an Application Programming Interface (API) reference (B), and some selected source code samples (C). The appendix is further followed by a list of acronyms, the bibliography, and a keyword index.

2. Basic Concepts

This chapter introduces common basic concepts of spatial cognition and information processing, which are used in the description of the system architecture and, subsequently, in the language reference.

The necessary basic and compound spatial types, notions, and concepts are tied together in the first section by the example of a key application for MPLL, the generation of route descriptions. This section illustrates the interworkings of the concepts and shows their respective use, connection and relevance. The second section further introduces the concepts, which represent the counterparts of MPLL to the essentials of human spatial reasoning, for example the notions of *configuration* and *direction*. Subsequently, some qualitative geospatial relations are discussed, which occur in natural language (see examples in section 2.1) and have to be handled by MPLL. Furthermore, a detailed definition and discussion of the term *landmark* is given, for this notion is a fundamental concept in MPLL. Then, the possible occurrences of fuzziness and its variants are laid out, along with practical illustrations and scenarios. Finally, the relevance of user and context modelling is introduced, albeit that a comprehensive discussion of this broad field is out of the scope of this work.

2.1. Introductory Example

Car Navigation

Generating route descriptions is, in general, not a trivial issue. Common car navigation systems rely on the underlying domain of car travel with the implicit rules and regulations to provide a very restricted environment regarding vocabulary and action alternatives.

The domain of car travel is rather confined, for example by traffic rules and regulations, driving physics, and modes of transport. Car drivers cannot just turn anywhere or reverse the direction of their vehicle, they are restricted to certain paths. In contrast to this, the technology can be deployed in a rather unrestricted manner. Components do not have to be particularly small or very efficient regarding power consumption. Using a car's available instrumentation can counteract for bad reception or even signal loss (dead reckoning). Additionally, a large data set of maps of transport networks can be used to find the most probable positional fix, if positioning is ambiguous.

Providing route descriptions under these circumstances is comparably straightforward and unambiguous.

Pedestrian Navigation

The navigation of human users in indoor and outdoor environments is much more complex due to the lack of rules and regulations, and, subsequently, too many and very diverse action alternatives.

Pedestrian locomotion can be comparably erratic, and there are no means for coping with unreliable positioning – there is no way to do dead reckoning in a pedestrian scenario. However, considering the smaller scale of locomotion – pedestrians travel much slower and less far than cars – there is a specific need for reliable and precise positioning. Furthermore, mobile devices must be smaller, lighter, consume less power, offer similar capabilities regarding computation power and storage space, and be equally reliable as their in-car counterparts.

Also, pedestrian locomotion suffers from the complexity of multimodality and the inherently frequent context switches. The just mentioned example of car navigation would constitute only one of several modes of transport, each mode individually adding their own specificities and complexity to the overall navigation task.

Especially in the domain of pedestrian navigation, route descriptions have to be adjusted to the individual user profile and context, because pedestrians do not have such a strong common code (traffic laws) as is required for car traffic. Furthermore, due to the lack of network-specific landmarks (such as changes in road structure and so-called road furniture, e.g. traffic lights and signs), external landmarks are much more important. However, integrating external landmarks into a route is very complex (see section 2.6.3), and the definition of landmarks itself is a nondeterministic and highly subjective process (see section 2.6.4).

Producing route descriptions such as those presented later in this section (see examples 1 and 2) is very different from, and much more complex than, displaying routes within street networks containing simple messages such as "turn right" or "go straight". In pedestrian navigation action alternatives are much more diverse and there exists no predefined and strictly regulated set of artificial[1] landmarks – car traffic would not be possible without such a set.

[1] These are, for example, traffic lights, signs, or markings on the road. See introduction to landmarks in section 2.6 on page 61.

MPLL provides means for identifying and integrating landmarks into route descriptions, for example by examining the spatial relations between the user's position and the landmarks. Furthermore, MPLL aides in expressing spatial relation between different features in order to specify locations which in the final route descriptions constitute so-called *trigger cues* and *reassuring cues* (see section 2.6.3, p. 65).

2.1.1. Elements of Route Descriptions

The following examples of route descriptions have been gathered from some German web sites in January 2006 by searching Google for the word "Wegbeschreibung" (Engl. directions). For reasons of authenticity and since the actual wording is not relevant here, we have abstained from complete translations. The essential information is conveyed using the categorisation of geospatially relevant phrases and expressions. These have been translated, whenever necessary.

Landmarks are an essential part of natural language directions. As described in section 2.6, they serve to provide *trigger cues* and *reassuring cues*. Since almost every action in a list of instructions for following directions is in some way associated with at least one landmark, directions usually contain a significant number of landmarks.

Angular Expressions are the second essential factor commonly found in directions. Spatial relations between two or more positions (e.g. the positions of the user, one or more landmarks, and the destination) are based on expressions of direction and distance. While the latter are more prone to fuzziness and less important for the semantics of a route, the former are more important and therefore exhibit a stronger presence.

Distal Expressions are used for expressing quantifiable information, such as *"100 m"* or *"at the third station"*. They can be used in a fuzzy or non fuzzy sense. While *"100 m"* can mean *"around 100 m"* and therefore should be interpreted as *"100 m* $\pm \varepsilon$*"*[2], the instruction *"leave the subway at the third station"* is unambiguous.

Example 1

The following description contains the directions from the airport *Berlin-Schönefeld* to the *Verbraucherzentrale Bundesverband*, located in the city of Berlin, Kochstraße 22, using different means of transport, such as the Airport Express regional train and subway.

> "**Anreise vom Flughafen Schönefeld** – Fahren Sie ab dem Bahnhof Berlin-Schönefeld mit dem Airport Express (Regionalbahn) bis zum Bahnhof Friedrichstraße, von dort weiter mit der U-Bahn Linie U 6 in Richtung Alt-Mariendorf bis zum Bahnhof Kochstraße (3 Stationen).

[2] The choice of a suitable value for ε depends on the context. Here, it could amount to, for example, 10 m.

> Benutzen Sie den U-Bahn-Ausgang in Richtung Kochstraße und biegen Sie nach rechts in die Kochstraße ein. Nach ca. 100 m erreichen Sie die Hausnummer 22."[3] ■

This example shows how different elements can have multiple, but definite, meaning. For example the item *Hausnummer 22* (Engl. *house number 22*) conveys all three categories of meaning: (1) due to its singularity (there is only one house with the number 22 in this street) it can be seen as a landmark, (2) because of the numbering scheme it can be used to determine direction, both longitudinal and lateral[4], and (3) the order of the house numbers provides a suitable means to express distance (e.g. "continue along the street until you reach house number 22" instead of "continue for approximately 180 m").

Landmarks	Bahnhof Berlin-Schönefeld, Airport Express (Regionalbahn), Bahnhof Friedrichstraße, U-Bahn Linie U 6, Alt-Mariendorf, Bahnhof Kochstraße, Kochstraße, Hausnummer 22	*translation n.a.*
Angular Expr.	in Richtung Alt-Mariendorf, in Richtung Kochstraße, biegen Sie nach rechts, Hausnummer 22	in direction Alt-Mariendorf, in direction Kochstraße, turn right, (house) number 22
Distal Expr.	3 Stationen, nach ca. 100 m, Hausnummer 22.	(at the) 3rd stop, after approx. 100 m, (house) number 22

Table 2.1.: Elements of Directions (Example 1)

Example 2

The following description contains the directions for pedestrians from the central train station Freiburg to the computing centre of the University of Freiburg.

> "**Ankunft mit der Bahn** – Wenn Sie den Hauptbahnhof Richtung Stadtmitte verlassen, befinden Sie sich auf der Bismarckallee. Sie müssen sich

[3] `http://www.vzbv.de/start/index.php?page=kontakt&pagelink=wegbeschreibung`

[4] All even numbers are located on one side, all odd numbers on the other, and numbers are ordered. Therefore, if the house number 4 is on the left, then house 22 must also be on the left. Likewise, if houses 4 and 6 are located in this order on the left hand side, it can easily be deduced that house 22 must be located further along the street – exceptions notwithstanding.

nach links wenden und immer geradeaus gehen. Sie überqueren die Friedrichstraße; die Bismarckallee geht hier in die Stefan-Meier-Straße über. Die dritte Querstraße rechts ist dann die Hermann-Herder-Straße. Das Eckhaus auf der linken Straßenseite ist das Rechenzentrum (Nr.10)." [5] ■

Yet again, these directions contain several expressions (as listed in table 2.2) which have multiple meaning. Some landmarks fulfil the purpose of positioning and reassuring the user, and, at the same time, providing directional cues. This is the case, for example, with the expressions [if you leave the train station in direction of the city centre, then] *you are on Bismarkallee*, or [if you continue straight on, then] *you will cross Friedrichstraße* [and Bismarkallee will be renamed Stefan-Meier-Straße]. Note that the double occurrence of the landmark *"Friedrichstraße"* in this and the previous example is purely coincidental.

Landmarks	Hauptbahnhof, Stadtmitte, Bismarckallee, Friedrichstraße, Stefan-Meier-Straße, Querstraße, Hermann-Herder-Straße, Eckhaus auf der linken Straßenseite, Rechenzentrum	*translation n.a.*
Angular Expr.	Richtung Stadtmitte, nach links wenden, immer geradeaus gehen, überqueren, auf der linken Straßenseite	in direction of the city centre, turn to the left, continue straight, cross [sth.] (implicates sth. in lateral position), on the left hand side [of the street]
Distal Expr.	überqueren die Friedrich- straße, dritte Querstraße rechts, (Nr. 10)	[until] you cross Friedrich- straße, third crossroad to the right, (house) number 10

Table 2.2.: Elements of Directions (Example 2)

[5] http://portal.uni-freiburg.de/rz/organisation/wegbeschreibung/

2.1.2. Summary

These concise examples represent only a fraction of the broad spectrum of concepts in human communication of spatial issues. However, the most important elements can immediately be found: landmarks, angular expressions, and distal expressions. To express direction or distance, humans almost always use spatial relation to one or more landmarks. Therefore, the basic types and many constructs in MPLL, as they are laid out in the next section, contain these basic elements and reflect the underlying principles.

2.2. Reference Systems

The notion of a *Reference System* (RS) is central to the study of spatial cognition across all levels of cognition (sensory and motor systems, conceptual thinking, language) and all the disciplines that study them (psychology of perception and attention, developmental psychology, neuropsychology, neuroscience, computational modeling, robotics, linguistics, psycholinguistics, philosophy, aesthetics, architecture).

As sketched in this section, a number of distinctions have been proposed between frames of references and each of these distinctions has been variously construed both within and across disciplines. There has been considerable discussion in the literature concerning the merits of various competing taxonomies of frames of reference. See for example the work of Roberta Klatzky [90], Barbara Tversky and Paul Lee [158, 159], and Steffen Werner et al. [168] for more details on these issues.

There has also been extensive discussion concerning whether different representational systems natively and necessarily employ proprietary frames of reference and the degree to which frames of reference are compatible across representational systems.

Within the scope of this work we need to briefly clarify the differences and commonalities between these distinctions. However, we must also clarify necessary nomenclature.

When nonpropositional, especially spatial, information is to be expressed, entities need to be included which are of relevance to the communicative task at hand. In this step of abstraction, which is referred to as *thinking for speaking* by Dan I. Slobin [149] or *microplanning* by Willem J. M. Levelt [98], we need to generate predications that accurately capture the entities' spatial relations within the scene. This process typically involves three components:

1. **Referent:** The object or portion of the scene, whose spatial disposition (i.e. position, orientation, path, etc.) is to be expressed [152], is called *referent*.

2. **Relatum:** The referent's spatial disposition is expressed using an the reference to an object or portion of the scene. This object or portion is called *relatum*.

3. **Perspective System:** The frame of reference in terms of which the referent is related to the relatum, or in which the referent's orientation or path is expressed, is called *prespective system*.

2.2.1. Types of Reference Systems

Depending on the way in which the above given components work together, a number of different types of spatial reference can be distinguished:

Allocentric

Allocentric[6] Reference Systems (RSs) (also called *exocentric* or *geocentric* [90]) assume a fixed coordinate system, whose direction and origin are defined by external factors such as geographic directions (e.g. *north*, *south*, etc.), and are independent of the observer's current position and/or orientation [158, 159]. Allocentric RSs facilitate expressions such as *north of the church* or *in the southern part of the city*. This kind of spatial reference also exists in smaller scale scenarios, when spatial disposition is described in, for example, oceanic languages by using expressions such as *inland* and *seaward*. Allocentric reference is very much depending on locality in scenarios such as these, described in more detail in section 2.5.1.

Egocentric

In an egocentric RS, the origin of the coordinate system is defined by the location and orientation of some object, mostly a human person. The orientation is defined with respect to the intrinsic body axes [90].

Object-Centred

According to an object-centred RS, the axes are oriented with respect to the intrinsic sides of the reference object, i.e. along its intrinsic longitudinal axis. Sometimes the determination can be subjective or difficult. Buildings, for example, do not always have a clear layout and/or orientation (e.g. TV tower vs. church). They might for instance have multiple "main" entries, (axis-) symmetrical layout and/or irregular shapes which do not serve well for determining an intrinsic *front* or *back*.

World-Centred

Here, the axes are oriented with respect to salient aspects of the world, such as gravity or cardinal directions. World-centred RSs are also called *environment-centred*.

Viewer-Centred

According to a viewer-centred RS the axes are oriented with respect to the head/feet, front/back and left/right sides of the viewer. In contrast to object-centred RSs, it is in this case rarely difficult to determine orientation.

[6] The origins of this expression are the Greek syllables "allo-" or "all-": *different*, *other*, *another*; *divergence*.

Absolute

If an RS does not depend on the bearing or orientation of an object, then it is absolute. *World-centred* and *allocentric* RSs are absolute. *Orientation-free* RSs that depend on the bearing between objects are absolute only if the objects are static, e.g. buildings, etc.

Relative

By contrast, if an RS **depends** on the bearing or orientation of an object, then it is relative. *Egocentric*, *object-* or *viewer-centred*, and *orientation-bound* RS are relative.

Intrinsic

The orientation of an involved object determines the frame of reference, e.g. the intrinsic front of a building, which facilitates statements such as *"in front of the church"* or *"behind the supermarket"*. An intrinsic RS is always *orientation-bound* and *object-centred*.

Extrinsic

The frame of reference used is external to (all of) the involved objects, such as the cardinal directions. Hence, orientation is global.

Deictic

The bearing of an external observer to an involved object, defines the frame of reference[7]. Statements such as *"left of the TV tower"* (concerning objects which may have no intrinsic front) or *"across the pond"* fall into this category. A deictic RS is always *orientation-free* because the orientation of the RS is based on a bearing and it is *viewer-centred* (viewer meaning the same as observer, speaker, listener, etc.).

Orientation-Free

The orientation of the frame of reference is either *global* or it depends on the *bearing* between two involved objects.

Orientation-Bound

The orientation of an involved object determines the frame of reference, e.g. the intrinsic front of a building, which facilitates statements such as *"in front of the church"* or *"behind of the supermarket"*. An orientation-bound RS is always *intrinsic* to a certain involved object.

[7] The origin of this expression is the Greek word "deixis": *display*, *demonstration*, *reference*.

2.2.2. Anchoring

The question of anchoring arises as soon as more than one reference system occurs in the same context. Anchoring means putting one reference system in relation to another by defining parameters which spatially fix its position in terms of the other reference system. Usually, this involves origin and orientation, as well as scale of a reference system.

This process is necessary, for example, if data from two different sources are used, which (at least partly) pertain to the same area, each providing different features, e.g. one source holds residential zones, the other industrial zones. If objects from these two sources do not adhere to the same reference system, objects from one source cannot be related to objects from the other source – unless there is a possibility to translate the coordinates from one reference system to the other. Even in cases where the underlying coordinate systems are identical, if the reference system uses for example a different prime meridian, then the coordinates are not compatible and have to be converted.

First, the orientation and origin of one reference system has to be anchored in terms of another reference system by giving the necessary translation and rotation matrices. This way, the origins of both reference systems are identical. Second, the scale of the reference systems must be equalised by providing a scale factor (or several, one for each axis, in case of anamorph scale) for the conversion of coordinates. Hence, moving in a certain direction for a certain distance renders the same position in both reference systems

2.3. Coordinate Systems

This section also covers some aspects of three dimensional computer graphics. For more details on these issues and some introductory tutorials see, for example, the Microsoft DirectX SDK [45]. Julier and Uhlmann [87] focus on the conversion between polar and cartesian coordinates.

2.3.1. Cartesian Coordinate Systems

Left-handed and Right-handed Coordinate Systems

Typically, three dimensional graphics applications use two types of Cartesian coordinate systems: *left-handed* and *right-handed*. In both coordinate systems, the positive x-axis points to the right, and the positive y-axis points up. An easy way to picture the axes is to point the fingers of either one's left or right hand in the positive x-direction and curling them into the positive y-direction. The direction one's thumb points, either toward or away from one's body, is the direction of the positive z-axis.

To translate data from a right-handed coordinate system to a left-handed one is not trivial. Two changes are required:

- The order of triangle vertices must be flipped so that the system traverses them

clockwise from the front. For vertices $v0$, $v1$, and $v2$ have to be reordered as $v0$, $v2$, and $v1$.

- Additionally, world space must be scaled by -1 in the z-direction in order to get the correct coordinates.

Although left-handed and right-handed coordinates are the most common systems in three dimensional graphics, a number of other coordinate systems exist. It is not unusual for 3D modelling applications to use a coordinate system in which the y-axis points toward or away from the viewer, and the z-axis points up.

Formally, the orientation of a set of basis vectors (i.e. a coordinate system) can be found by computing the determinant of the matrix defined by the particular set of basis vectors. If the determinant is positive, the basis is said to be "positively" oriented (or right-handed). If the determinant is negative, the basis is said to be "negatively" oriented (or left-handed).

The essential operations performed on objects defined in a 3D coordinate system are translation, rotation, and scaling. You can combine these basic transformations to create a transform matrix. These operations are not commutative, the order in which matrices are multiplied is important.

2.3.2. Transformations

Transformations are used to convert object geometry from one coordinate space to another. This is done using several kinds of transformation matrices which are described in the following sections.

Matrix Transformations

Geometrical transformation using matrices can be used, among others, for the following purposes:

- expressing the location of an object relative to another object
- translation, rotation and scaling of objects
- changing viewing parameters, i.e. panning, zooming, tilting, etc.

The transformation of a point (x,y,z) into another point (x',y',z') is done using a 4 by 4 matrix:

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

This results in the following operations on x, y, and z to produce x', y', and z':

$$\begin{aligned} x' &= (x*M_{11})+(y*M_{21})+(z*M_{31})+(1*M_{41}) \\ y' &= (x*M_{12})+(y*M_{22})+(z*M_{32})+(1*M_{42}) \\ z' &= (x*M_{13})+(y*M_{23})+(z*M_{33})+(1*M_{43}) \end{aligned}$$

The most common transformations are *translation*, *rotation*, and *scaling*. Matrices can be combined into a single matrix to calculate several transformations at once. For example, a single matrix can be used to translate and rotate a series of points in one step. Matrices are written in row-column order.

Translation

The following matrix translates a point (x,y,z) into another point (x',y',z'):

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Scaling

The following matrix scales a point (x,y,z) by arbitrary values in the x, y, and z directions to a new point (x',y',z'):

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

The following[8] matrix rotates the point (x,y,z) around the x axis, producing a new point (x',y',z'):

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) & 0 \\ 0 & -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following matrix rotates the point around the y axis:

[8] These transformations are specific for left-handed coordinate systems.

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following matrix rotates the point around the z axis:

$$[x',y',z',1] = [x,y,z,1] \begin{bmatrix} cos(\theta) & sin(\theta) & 0 & 0 \\ -sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In these example matrices, θ stands for the angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Matrix Concatenation

The effects of two or more matrices can be combined by multiplying the individual matrices. In order to rotate and scale an object's vertices it is not necessary to apply two matrices consecutively. Instead, rotation and scaling matrices are multiplied and the resulting composite matrix is used to produce both effects.

$$C = M_1 \cdot M_2 \cdot \ldots \cdot M_{n-1} \cdot M_n$$

Here, C is the composite matrix to be created, and M_1 through M_n are the individual matrices. Usually only a small number of matrices is concatenated, although there is no technical limit.

The order of matrix multiplication is crucial. The preceding formula reflects the left-to-right rule of matrix concatenation. That is, the visible effects of the matrices that are useed to create a composite matrix occur in left-to-right order.

A typical matrix is shown in the following example. Imagine that you are creating the matrix for a flying saucer. Usually the flying saucer would spin around its centre - the y-axis of its model space - and then translated to some location in (world) space. This effect is done by creating a rotation matrix, and then multiplying it by a translation matrix:

$$C = R_y \cdot T_w$$

Here, R_y is the rotation matrix (around the y axis), and T_w is the translation matrix which produces translation to some position in world coordinates.

Unlike multiplying scalar values, matrix multiplication is not commutative. Therefore the order of multiplication is important. Multiplying the matrices in the opposite order would result in translation of the saucer to its destination and rotation along the y axis around the world origin. Needless to say that this is a completely different effect.

2.3.3. Longitude and Latitude

Longitude

Longitude (λ) denotes the location of a position on Earth east or west of a prime meridian. Longitude is given as an angular measurement ranging from 0° at the prime meridian to $+180^\circ$ eastward and -180° westward. Unlike latitude, which has the equator as an origin, there exists no natural origin for longitude. In order to anchor longitude on the planet, a reference meridian is needed. British cartographers have long used the Greenwich Prime Meridian in London. Other references used elsewhere include: El Hierro, Rome, Copenhagen, Jerusalem, Saint Petersburg, Pisa, Paris, Philadelphia and Washington. In 1884, the International Meridian Conference adopted the Greenwich meridian as the universal prime meridian or zero point of longitude. If not stated otherwise, we use the Greenwich Prime Meridian by default.

Latitude

Latitude (φ) is used to express the location of a position on Earth north or south of the Equator. Latitude is given as an angular measurement ranging from 0° at the Equator to 90° at the poles (90° N or 90° S). Certain lines of latitude are important such as the Equator, Arctic Circle, Tropic of Cancer, Tropic of Capricorn or Antarctic circle.

Each degree of latitude is sub-divided into 60 minutes, whereas one minute of latitude amounts to one nautical mile (1852 metres) at sea level. This varies slightly with latitude because of the Earth's non-spherical shape. Each minute further divides into 60 seconds (one second equals about 31 metres), which can be given with a decimal fraction for higher accuracy. Sometimes, the *south* suffix *S* is instead indicated by a negative sign.

Like latitude, each degree of longitude is sub-divided into minutes and seconds. However, one minute of longitude amounts to $\cos(latitude) * 1852\text{m}$ (a value within the interval $[0, 1852\text{m}]$), depending on the latitude, since all longitudes are joined together at the poles. Sometimes, the *west* suffix *W* is instead indicated by a negative sign.

2.3.4. WGS-84

The World Geodetic System (WGS) defines a fixed global reference frame for the Earth, for use in geodesy and navigation [172]. The latest revision is WGS-84 dating from 1984, which will be valid up to about 2010.

In the early 1980s, the need for a new world geodetic system was generally recognised by the geodetic community, also within the Department of Defense. WGS-72 no longer provided sufficient data, information, geographic coverage, or product accuracy for all then current and anticipated applications. The means for producing a new WGS were available in the form of improved data, increased data coverage, new data types and improved techniques. Geodetic Reference System (GRS) (currently GRS-80) parameters together with available Doppler, satellite laser ranging and Very Long Baseline Interferometry (VLBI) observations constituted significant new information. Also, an outstanding new source of data had become available from satellite radar altimetry. Also available was an advanced least squares method called collocation which allowed for a

consistent combination solution from different types of measurements all relative to the Earth's gravity field, i.e. geoid, gravity anomalies, deflections, dynamic Doppler, etc.

The new World Geodetic System was called WGS-84. It is currently the reference system being used by the Global Positioning System (GPS). It is geocentric and globally consistent within ± 1 m. Current geodetic realisations of the geocentric reference system family International Terrestrial Reference System (ITRS) maintained by the International Earth Rotation and Reference Systems Service (IERS) are geocentric, and internally consistent, at the few-cm level, while still being metre-level consistent with WGS-84.

The longitude zero is located approximately 100 metres east of the traditional prime meridian in Greenwich, U.K.

2.3.5. Universal Transverse Mercator Coordinate System

The Universal Transverse Mercator coordinate system (UTM) [171] is a grid-based method for specifying locations on the surface of the Earth. It is different from the traditional method of latitude and longitude (see WGS-84 above) in several aspects. UTM is not a map projection, but rather consists of a series of zones based on specifically defined Transverse Mercator projections. Originally developed by the U.S. Army in 1947, UTM now relys on the WGS-84 ellipsoid as the underlying model of the Earth.

The UTM system divides the surface of the Earth between 80° S latitude and 84° N latitude into 60 zones, each 6° of longitude in width and centred over a meridian of longitude. Zones are numbered from 1 to 60. Zone 1 is bounded by longitude 180° to 174° W and is centred on the 177th West meridian. Zone numbering increases towards the east.

Each of the 60 longitude zones in the UTM system are the basis of a Transverse Mercator projection, which is capable of mapping a region of large north-south extent with a low amount of distortion. By using narrow zones of 6° in width, and reducing the scale factor along the central meridian to 0.9996, (a reduction of 1:2500) the amount of distortion is held below 1 part in 1,000 inside each zone. Distortion of scale increases to 1.0010 at the outer zone boundaries along the equator.

The reduction in the scale factor along the central meridian creates two lines of true scale located approximately 180 *km* on either side of, and approximately parallel to, the central meridian. The scale factor is too small inside these lines and too large outside of these lines, but the overall distortion scale inside the entire zone is minimized.

2.3.6. Gauß-Krüger Coordinate System

Like UTM, the Gauß-Krüger coordinate system is a grid-based method for specifying locations. It uses the same algorithms as UTM for the projection of the respective ellipsoid

onto the plane. The main difference is that Gauß-Krüger coordinates correspond to the Bessel- or Krassowskie ellipsoid in zones of 3° in width, while UTM coordinates correspond to the WGS-84 ellipsoid in zones of 6° in width. Austria uses the Datum Austria with a slightly shifted ellipsoid and reference meridian. If Austria were to use the same datum and reference as Germany, the country would be divided into four instead of three zones. The Gauß-Krüger coordinate system is also used in Russia in connection with the Krassowskie ellipsoid. This was also the case in former Eastern Germany, where it is still used in some areas today.

The system has been developed by Johann Heinrich Louis Krüger[9] based on previous work by the German mathematician Carl Friedrich Gauß[10] and is in use in German-speaking Central Europe since 1923. Many official topographic maps, especially large and medium scale, make use of the Gauß-Krüger coordinate system.

2.4. Basic Data Types

MPLL uses classic numeric data types to represent quantitative data, such as binary, integer, and floating point values. The implementation details (e.g. size of data types) depend on the respective platform where MPLL is used. The basic concepts of MPLL consist of these basic types. Points in (planar) space, for example, are defined by a pair of coordinates, which are integer or floating point values. Likewise, lines are defined by a pair of points, and polylines and polygons by three or more points. A formal definition of these basic types can be found in section 4.4.1.

In the application domain of MPLL, an essential component in modelling the real world are point entities, for example representing users, points of interest, or landmarks. Data structures for these point entities need to hold more information than just a position in space. The next section introduces the concept of configurations and configuration space, which, in MPLL, is used to model point entities. In the subsequent sections, the mechanisms for representing direction are laid out, which constitute an equally important component in the underlying model.

2.4.1. Configurations and Configuration Space

Configuration Space

In the field of classical mechanics, the space of all possible positions that a physical system may attain is called *configuration space*, or *C-Space*. The number of possible positions is also subject to external or internal constraints. The C-Space of a typical system has the structure of a manifold, therefore it is also called the *configuration manifold*.

The C-Space of a single point in Euclidean 3-space is R^3. For N points the C-Space is R^{3N}, i.e. the subspace where no two points are equal. More generally, the configuration space of N points moving in a manifold M can be regarded as the function space M^N.

[9] Johannes Heinrich Louis Krüger (∗ 21. September 1857, Elze, – † 1. June 1923)

[10] Johann Carl Friedrich Gauß (∗ 30. April 1777, Braunschweig, – † 23. February 1855, Göttingen)

A concrete example is a mechanical robot arm with one joint. Suppose the joint could do a full 360° turn, the configuration space would be $[0, 360°[$. In case the robot arm had two joints of this kind, the configuration space would be $[0, 360°[\times [0, 360°[$.

Configuration Manifold

If both position and momenta need to be taken into account, one moves to the cotangent bundle of the *configuration manifold* (see e.g. Ritter [140] or Tilbury et al. [154] for examples), which is called the *phase space* of the system.

Within the scope of this work, we do not focus on momenta or any implied dynamic changes, since we concentrate on spatial relations and their composition, description and communication. Nevertheless, we use the term *configuration* in the original sense, i.e. in describing an object's physical state. For point entities, this state contains two properties: position and orientation. If the object represented by the configuration has no intrinsic front, then the orientation is not used.

2.4.2. Angular Expressions

There are several parameters of a configuration which can be expressed using angular values. Regarding computation, these can be treated in much the same way, while their semantics partly differ considerably.

Direction

Direction is denoted by the *angles* which a vector (defined by two positions *a* and *b*) spans to the axes of a reference coordinate system. Usually, the base of such a vector (the user's position) is located at the origin of the coordinate system, while the tip of the vector is located somewhere in the plane or space spanned by the coordinate system's axes. This way, a planar (2D) direction can be expressed by an angular value $\alpha = [0°, 360°[$ between the vector and the x-axis, and a spatial (3D) direction can be expressed by a pair of angular values $\alpha = [0°, 360°[$ and $\beta = [0°, 360°[$ between the vector and the x-z-plane and the x-y-plane respectively. The horizontal component of a direction is called *azimuth*, the vertical component is called *elevation*. The true direction of travel, which is not necessarily identical to the heading (see below) is referred to as *track*, θ_t. When a direction denotes movement, it is also called *heading*, when it is used in a static context, i.e. without movement, it is called *orientation* (see Fig. 2.1) to denote the intrinsic front of an object.

Formal Definition

Direction is the information contained in the relative position of one point to another point, without distance information. Direction can be specified by a unit vector whose components are direction cosines. Together with magnitude, direction can be specified by a vector in general. A unit vector in a normed vector space is a vector whose length is 1. In Euclidean space, the dot product of two unit vectors is the cosine of the angle between them. This follows from the formula for the dot product, since the lengths are

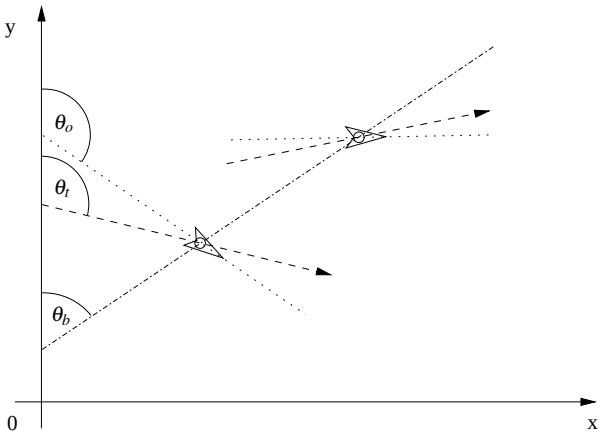


Figure 2.1.: Orientation, Track, and Bearing

both 1. The normalized vector $\hat{u}$ of a non-zero vector u is the unit vector codirectional with u, i.e.,

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|} \tag{2.1}$$

where $\|u\|$ is the norm (or length) of u. As opposed to a general vector, which represents direction and magnitude, a unit vector just represents direction. The components are called direction cosines, because each is the cosine of the angle between the vector and one coordinate axis. The elements of a basis are often chosen to be unit vectors. In the three-Dimensional Cartesian coordinate system, these are usually i, j, and k–unit vectors along the x, y, and z axes, respectively:

$$\hat{\mathbf{i}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \hat{\mathbf{j}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \hat{\mathbf{k}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.2}$$

Orientation / Heading

Orientation, θ_o, defined in the same way as direction, denotes the *alignment* or *arrangement* of an object to face into a certain direction. The base of the vector is in this case situated at the origin (or central reference point) of the object itself. For example the orientation and the direction of movement of a non-holonomic vehicle, such as a conventional automobile, while going forward are – due to the way that it is constructed

– identical[11]. In the case of a holonomic vehicle, orientation and direction of movement are not necessarily identical. An orientation always relates some object to a certain Reference System (RS).

In certain domains, for example navigation or aviation, the true direction of travel (θ_t) is often different from the heading of the craft. Due to currents and/or cross winds one must almost always consider a certain deviation between the direction of travel and the orientation of the craft. In order to reach a particular location, such as a waypoint, the heading of the craft must be adjusted so that the bearing to the waypoint and the true course match. Then, the heading and the bearing often do not match.

In three-dimensional models, orientation, i.e. rotation around the vertical axis, is called *yaw*. Rotation around the longitudinal axis is called *roll* or *bank*. This is the movement, for example, a plane undertakes, when turning to a new course. Rotation around the lateral axis is called *pitch*, which is, for example, performed by a plane at takeoff.

Bearing

Bearing (θ_b, see Fig. 2.1), in contrast to orientation, relates an object to *another* object. It is defined by the angle between the direction to an object and a reference direction. This reference direction is usually *magnetic north* [12], in which case also the term *compass bearing* is used. Bearing is the only property which necessarily relates two *objects* to each other, and it is also referred to as **course**.

If $\triangle x = x_B - x_A$ and $\triangle y = y_B - y_A$ with $A = (x_A, y_A)$ and $B = (x_B, y_B)$, then

$$\theta_b(A,B) = \begin{cases} cos^{-1}(\frac{\triangle y}{\sqrt{\triangle x^2 + \triangle y^2}}) & \text{if } \triangle x \geq 0 \\ cos^{-1}(\frac{\triangle y}{\sqrt{\triangle x^2 + \triangle y^2}}) + \pi & \text{otherwise} \end{cases}$$

2.4.3. Shape and Size

The shape and size of spatial entities is a very interesting part of spatial information processing. Especially in the field of qualitative reasoning techniques, there still exist a number of unsolved issues, mainly because of the difficulties in finding a suitable vocabulary and set of spatial relations, as well as a common understanding of the issues involved with irregular shapes. The notion of size and scale is also of great importance. A typical statement would be, for example, *"the bicycle is in front of the church"*. The reason for this statement not to be formulated in reverse (*"...church behind bicycle..."* is

[11] Arguably, this is not entirely true in some driving situations, for example when there exists moving friction between the tyres and the road surface (i.e. the vehicle is sliding). However, under normal conditions the above mentioned statement is sufficiently correct for the purpose of illustration.

[12] When a gyrocompass is used, the reference direction is *true north*. In stellar navigation, the reference direction is that of the North Star, *Polaris*.

primarily due to the comparison of the sizes of the two spatial entities. However, more complicated factors influence these issues, such as whether entities are mobile or immobile, the symbolic significance of entities, or issues pertaining to different languages and cultures.

Within the scope of this work, however, shape and size are not of primary importance. As already laid out, human spatial reasoning deals primarily with entities that can easily be modelled as points (e.g. the user's position, other entities' positions). Furthermore, most occurences of linear features or regions more often involve topological relations (e.g. whether something is inside or outside of a region, or on which side of a river something is located), rather than pertaining to the exact shape or size of an entity.

In any case, suitable expressions dealing with shape and size can, and will, be added to the language MPLL at a later point in time. However, these issues are not covered in the first version of the language specification.

2.5. Basic Spatial Relations

Relations involving at least one point entity are generally of more interest within the scope of this work due to the underlying problem domain of navigation and wayfinding. In this domain, users (or mobile entities in general) are preferably modelled as entities which are located at a single point in (Euclidean) space. Moving through their model space, these point entities can have a number of properties, such as a position (geometric or symbolic), orientation, and others. A collection of these properties is referred to as *configuration*, as already discussed in section 2.4.1.

2.5.1. Direction

Cardinal Direction

The two classic examples for cardinal direction in user modelling are the cardinal points (i.e. cardinal compass points) and the generic egocentric cardinal directions. The former consist of the terms *north*, *south*, *east*, and *west*, and combinations thereof, for example *southeast*, as shown in Fig. 2.2 a. The latter are composed from the terms *front*, *back*, *left*, and *right*, including combinations such as *left front* (see Fig. 2.2 c and d). Due to usability reasons, the number of directions is usually limited to four or eight, albeit in certain domains smaller partitionings might be used for the purpose of increased accuracy. This is, for example, the case in navigation, where there exists a third level (*north-northeast*) of cardinal directions combining the first (*north*) and second level (*northeast*), resulting in a 16-fold partitioning shown in Fig. 2.2 b.
In addition to regular partitionings, there exist irregular partitionings (see Fig. 2.2 d), which originate from cognitive research. The reason for irregular partitionings is, for example, the uneven perception of lateral forward and backward space, which occurs with human spatial cognition [141, 167].

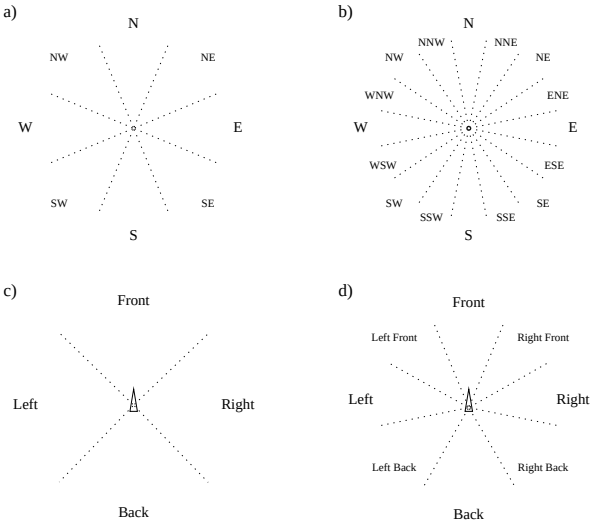


Figure 2.2.: Allocentric and Egocentric Cardinal Direction

A special case of allocentric cardinal direction manifests itself in spatial expressions found in oceanic languages. Like with generic allocentric direction, the cardinal directions depend on an allocentric reference system. In this special case, however, there is a *small scale dependence* between the reference direction and the orientation of cardinal directions (see Fig. 2.3). The cardinal directions follow a small scale dependence between the location of the referent and the locations of the centre of the island and the nearest shore. The same effect occurs close to the north or south pole with the cardinal direction shown in Fig. 2.2 a. However, this effect is usually negligible due to the distance of the application scenarios to the polar regions. On a technical level, scale notwithstanding, the effect is identical.

Point-to-Point Direction

Geospatial relation is almost always expressed in two different ways: (1) in relation to the observer's position, the current viewpoint, or (2) in relation to a tertiary object which serves as a reference point. The former we refer to as *absolute* or *egocentric direction* (as already discussed in section 2.2), since it relates directly to the observer's position, the latter we call *relative* or *global direction* because the target object relates to the observer via a tertiary object, which is part of the environment.

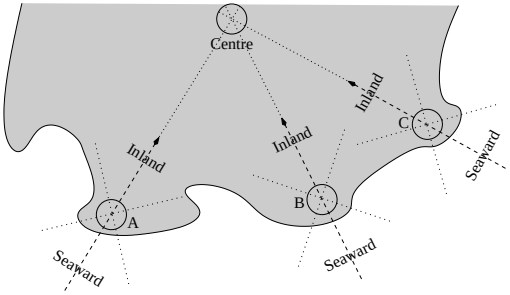


Figure 2.3.: Allocentric Cardinal Direction

	reference system	relation	example	figure
extrinsic	allocentric	absolute	*"X is located in southwest direction of A"*	2.4
intrinsic	egocentric (orientation)	absolute	*"X is located left of A"*	2.5
deictic	allocentric (bearing)	relative	*"X is located left of A (as seen from B)"*	2.6

Table 2.3.: Directional Spatial Relations

This section deals with directional spatial relations between two or more points. Directional spatial relations, such as *"left of"*, *"east"*, or *"behind"* require a frame of reference. Depending on the specific application, the frame of reference may be either *extrinsic*, *intrinsic*, or *deictic* [99, 41]. An overview over the different categories and properties is shown in table 2.3.

Extrinsic: The frame of reference used is external to the involved objects, and employs cardinal directions such as *north*, *west*, or *southeast*.

Fig. 2.4 shows the relation between objects A and X. The global (i.e. extrinsic to A and X) reference system renders *"X in southwest direction from A"*. The hatched area in this figure denotes the locations which lie *"west and east of A"*.

In this case, as well as the two subsequent cases, the MPLL function

$$\texttt{bearing}(Dir, C, D)$$

is used to determine spatial relation, i.e. it tests for the object denoted by C, whether D is located in the direction denoted by Dir. This function is overloaded

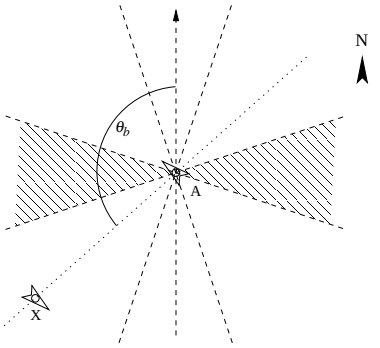


Figure 2.4.: Point-to-Point Directional Relation with Extrinsic Reference System; the hatched triangles denote the area *east* and *west* of A, X is located *south east of* A.

to handle the different cases depending on the type of *Dir*. See Def. 4.94 on page 166 in the MPLL Standard Library for more details. The composite function `bearing()` is an overloaded version of the basic MPLL function `bearing()` (see Def. 4.66 on page 143).

Intrinsic: The orientation of an involved object determines the frame of reference, for example the intrinsic front of a building, which facilitates statements such as *"in front of the church"* or *"behind the supermarket"*.

In Fig. 2.5 on page 49, which depicts the same setup as Fig. 2.4, *"X is located left of A"* due to the use of the orientation of object *A* as the reference system. The hatched area in this figure denotes the locations which lie *"left and right of A"*.

See Def. 4.95 on page 167 in the MPLL Standard Library for more details on the MPLL constructs dealing with this particular setup.

Deictic: In this case the bearing between an external observer, for example a speaker or listener, and the *relatum*[13] defines the frame of reference. Statements such as *"left of the TV tower"* (which may have no intrinsic front) or *"across the pond"* fall into this category. The role of the observer has also been referred to as *Relative* [148], since it is used to express relations between two independent objects via another object that *relates* to both.

A setup which incorporates an observer as object *B* is shown in Fig. 2.6 on page 50. The relation between the relatum *A* and the referent *X* as seen from *B* can be de-

[13] See section 2.2 on page 32 for a definition of *referent* and *relatum*.

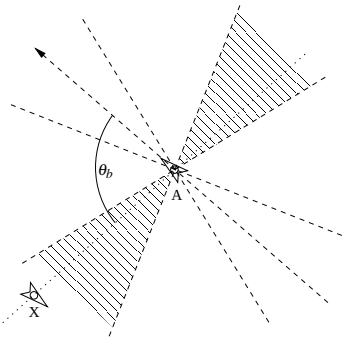


Figure 2.5.: Point-to-Point Directional Relation with Intrinsic Reference System; the hatched triangles denote the area *left* and *right* of A, X is located *left of* A.

scribed as *"X is located left of A"*. It must be noted that the orientation of the objects *A*, *B*, and *X* have no impact whatsoever on the spatial relation. Furthermore, for this (static) situation this statement is only true for an observer at position *B*. If the position of *B* changes (even while *A* and *X* remain static), this statement could cease to be true.

See Def. 4.96 on page 167 in the MPLL Standard Library for more details on the MPLL constructs dealing with this particular setup.

In general, qualitative direction can be expressed by using the models described in sections 6.1.1, 6.1.2, and 6.1.3. Escrig and Toledo [51] offer a more detailed discussion of these issues.

Absolute Direction

Whenever an extrinsic or intrinsic reference system is used, directional spatial relations have an *absolute* character, since directional expression does not depend on a tertiary relation with, for example, an observer. Therefore, statements such as *"the shopping mall is located east of the Olympic stadium"* (extrinsic RS) or *"the post office is in front of the church"* (intrinsic RS) are valid independently from the place of utterance or the location of an observer, speaker, or listener.

Relative Direction

The use of a deictic reference system always indicates *relative* directional spatial relation, for example when the observer holds a specified position or spatial expressions include a relative object: *"the news stand is located left of the main ticket booth, which you see over there"*.

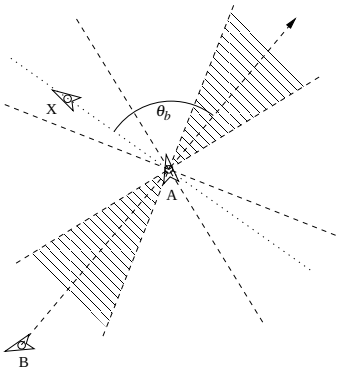


Figure 2.6.: Point-to-Point Directional Relation with Deictic Reference System; the hatched triangles denote the area *in front of* and *behind* A, X is located *left of* A.

Point-to-Line Direction

Qualitative direction between points and linear structures cannot be handled in the same straightforward way as between points. This applies also to point-to-region relations (see section 2.5.1) and is due to the fact that lines, polylines, and regions have, in contrast to points, a spatial extent. There are several simple and intuitive examples for point to line relations: *"the river runs in north/south direction east of the city centre"*, *"the border is located south of the airport"*.

The MPLL function

$$\texttt{bearing}(C, L)$$

of type $\texttt{Configuration} * \texttt{Line} \mapsto \texttt{CircularInterval}$ is used to determine the spatial relation between a point and a line entity. This function returns a crisp interval holding the bearings from C towards all line segments (i.e. all points and all segments between points) of L (see Def. 4.66 on page 143).

Alternatively, the line L can be reduced to one of its segments, namely the one closest to configuration C. This interpretation could be more suitable in some scenarios. The MPLL function

$$\texttt{closestSegment}(L, C)$$

of type $\texttt{Line} * \texttt{Configuration} \mapsto \texttt{Line}$ can be used to define a suitable function (see Def. 4.71 on page 148). Note that the `bearing` function above also accepts lines which consist of only one segment. Therefore, the same function can be called with the respective parameter.

Cardinal Point-to-Line Direction

Fig. 2.7 shows a setup in which the river is located south as well as north as seen from location A. In fact the only direction as seen from A which is not (at least partly) occupied by the river is northwest. Even in cases where the linear feature shows less distinct meandering (e.g. railway lines, highways, borders, etc.) this problem with the applicability of cardinal direction persists. One solution, which can also be found in natural language description, is to use cardinal direction only in unambiguous situations, for example when the linear structure is sufficiently straight, at least regarding a certain segment, and its orientation favours cardinal direction (i.e. it matches a distinct cardinal value, such as *east–west*).

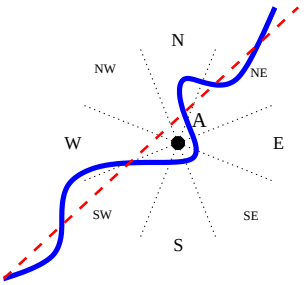


Figure 2.7.

Artificial reduction of meandering by interpolation of linear structures can reduce ambiguities with cardinal direction. However, this technique can also lead to errors. If the river in Fig. 2.7 were to be reduced to a simplified structure represented by the straight dashed line, while for location A the ambiguity in cardinal direction would be reduced, it would not anymore be situated on the correct side of the river.

Point-to-Region Direction

Directional ambiguities in point to region relations are very similar to those described in the previous paragraph dealing with point to line relations. Relations of this kind show analogous phenomena as those depicted in the scenario in Fig. 2.7. However, the likelihood of such problems to arise is smaller, since regions, if not defined by natural occurrences (shorelines, rivers, etc.), are often defined by cadastral partitioning which is usually done in a deliberately categorical manner. This results in clear and mostly straight lot borders which are often convex and therefore are not prone to the problems described above. The satellite image taken from an area near the Munich airport shows the different parcels alloted to different purposes (agricultural, infrastructure, residential) and samples some common shapes (see Fig. 2.8).

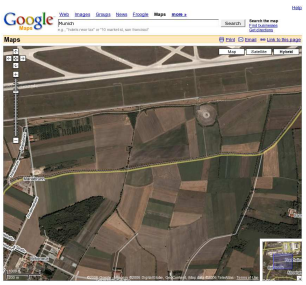


Figure 2.8.

The MPLL function

$$\texttt{bearing}(C,R)$$

of type $\texttt{Configuration}\star\texttt{Polygon} \mapsto \texttt{CircularInterval}$ is used to determine the spatial relation between a point and a region entity. This function returns a crisp

interval holding the bearings from C towards all polygon (region boundary) segments (i.e. all points and all segments between points) of R (see Def. 4.66 on page 143).

Line-to-Point Direction

Intrinsic Cardinal Direction

In contrast to point-to-line direction, line-to-point direction can be handled in a much more straightforward and unambiguous way by incorporating the direction of the linear structure, such as the flow of a river or the driving direction on a highway. However, the value of this kind of information is somewhat limited. In theory, the information *"the hospital is located on the right hand side of the river"*[14] leaves an almost infinitely large area in which the hospital is located and lacks any further means of specifying the exact location. Nonetheless, in practical situations which cover a limited area such as a portion of a map or the vicinity around a moving car, this type of information can be very important. The point of linear intersection, e.g. the crossing of a river, is ideally suited to represent a *landmark*. Sophisticated routing instructions would make use of such a landmark, either as a *trigger cue* or a *reassuring cue*. See section 2.6.3 on page 65 for a discussion of these terms.

The MPLL function

$$\texttt{side}(L,C)$$

of type $\texttt{Line} * \texttt{Configuration} \mapsto \texttt{Side}$ is used to determine the spatial relation between a line and a point entity (see Def. 4.66 on page 143). This function returns the side as the enumeration type `Side` (see Def. 4.17 on page 110 for the definition of enumeration types).

Additionally, in many cases this type of information can be very valuable in directing the user's attention to more important tasks. This is a crucial issue, for example, for car navigation systems, since devices need to adhere to special guidelines which make sure that the driver's attention is diverted from the ongoing traffic situation only to the least possible extent.

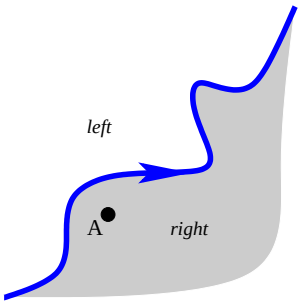


Figure 2.9.

As shown in Fig. 2.9, where the area located on the right hand side of the river is marked by the hatched area, expressions such as *"the hospital is located on the right hand side of the river"* can be determined unambiguously and expressed by quantitative means as well as be used in the qualitative sense. The reference to the river's intrinsic linear direction eliminates possible problems with cardinal directions as discussed in the next paragraph.

[14] Looking downstream determines the orientation and defines what is *left* and *right*.

Specifying extrinsic cardinal direction, such as *"north of"* in line to point relations is less straightforward than the previous case of intrinsic cardinal direction. Essentially, this is the inverted case of the setup depicted in Fig. 2.7, resulting in similar problems. MPLL provides the means to specify a function expressing the spatial relation according to the user's individual interpretation.

Extrinsic Cardinal Direction

One interpretation, for example, could be to determine the nearest line segment of L to C and to construct a circular interval with the bearings of the two endpoints of the segment. However, to determine the closest segment could be ambiguous (if there are two segments which have the same distance measure from C) and the range of interpretations of cardinal direction from a line segment to a point is very broad.

Line-to-Line Direction

The seemingly trivial directional relation between linear features serves a specific purpose, which has no immediate connection with the term "direction". With respect to a linear feature A, another linear feature B can be located either on the *left* side of A or on the *right* side of A, as depicted in Fig. 2.10. Whenever this relation changes, that is whenever both feature cross each other, a possibility for a landmark presents itself. Such a landmark can be, for example, a bridge or an underpass or tunnel. Features such as these serve perfectly well as landmarks as they can be determined rather easily.

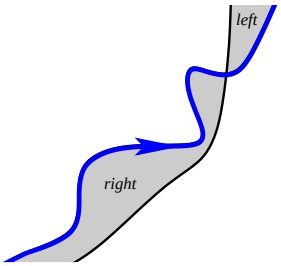


Figure 2.10.

Line-to-Region Direction

Often, the boundary of a region feature is perceived as a linear feature, which creates a similar scenario to the one described in the previous section. Therefore, this case can be treated in essentially the same way.

The broad range of possible interpretations leads to a number of different possibilities to model this relation. MPLL provides the necessary means to specify a suitable definition depending on the respective preferences of the user or the application.

Region-to-Point Direction

Cardinal direction between a region and a point has several interpretations. The query *"Is the Munich Olympic Tower located in the northern part of the city?"*, reflects one possible interpretation. Note that the use of the expression *"located in the north of"* implicates that the tower is located within city limits. If the expression *"located north of"* had been used, the implication would be that the tower is located *outside* city limits.

The query given above could be solved by the construction of a polygon representing *the north of* the city of Munich (i.e. a part of the original polygon), which could then be used for a test for inclusion of the point entity representing the tower.

Determining whether the tower is located *"north of"* the city of Munich would require a different approach. One possibility would be to construct a polygon representing the area located north of the city and to conduct yet again a test for inclusion. Note that for each individual interpretation of such a spatial relation, suitable processing mechanisms must be combined.

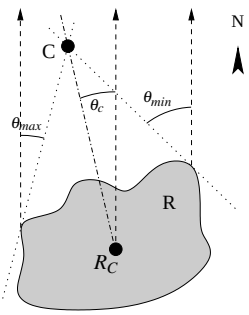


Figure 2.11.

Fig. 2.11 illustrates yet another interpretation of this relation. The bearing from the region R to a point C located outside of R can be defined, for example, in the following two ways. Either, the bearing should be returned as an interval, or it should be a single numeric value. The former version could be computed by an iteration over the polygon's points, the bearing of each point to A contributing to the construction of the interval (see Def. 4.61 for details). The latter version would be defined by the bearing from the centre of mass R_C of R to the point C.

The MPLL function

$$\texttt{bearing}(R,C)$$

of type $\texttt{Polygon} * \texttt{Configuration} \mapsto \texttt{CircularInterval}$ returns the bearing in form of a crisp interval holding the bearings from all polygon (region boundary) segments (i.e. all points of R) of R towards a configuration C (see Def. 4.66 on page 143). Effectively, this produces a range of bearings, constructed of all possible bearings from a point in R to C. Analogous, the overloaded version

$$\texttt{bearing}(\texttt{centerOfMass}(R),C)$$

of type $(\texttt{Polygon} \mapsto \texttt{Point}) * \texttt{Configuration} \mapsto \texttt{Angle}$ returns the bearing in form of an angle value.

Region-to-Line Direction

Following the examples above, there are several different interpretations for this type of relation. The result of these variants is either an interval, or a single numeric value. Polylines can be reduced to a single segment (or a single point) by the MPLL function

$$\texttt{closestSegment}(L,C)$$

of type `Line*Configuration` $\mapsto$ `Line`, see Def. 4.71. Similarly, polygons can be reduced to a single point (or a single segment) by the MPLL function

$$\texttt{closestPoint}(R,C)$$

of type `Polygon*Configuration` $\mapsto$ `Point`, see Def. 4.69.

The broad range of possible interpretations leads to a number of different possibilities to model this relation. MPLL provides the necessary means to specify a suitable definition depending on the respective preferences of the user or the application.

Region-to-Region Direction

Analogous to the previous relation, there are also several different interpretations for region to region direction. The result of these variants is, once again, either an interval, or a single numeric value. Polygons can be reduced to a single point by the MPLL function `closestPoint`(R,C) of type `Polygon*Configuration` $\mapsto$ `Point` (see Def. 4.69), or to a single segment by the MPLL function `closestSegment`(R,C) of type `Polygon*Configuration` $\mapsto$ `Line` (see Def. 4.71)

Also in this case, the broad range of possible interpretations leads to a number of different possibilities for modelling. MPLL provides the necessary means to specify a suitable definition depending on the respective preferences of the user or the application.

2.5.2. Distance

Distance is, unlike topology and direction, a scalar entity [137]. When humans communicate distance, they do so using qualitative categories (*"close to"*, *"far away"*), qualitative comparatives (*"closer"*, *"farther away"*), or metric distances (*"100 metres from here"*). Generally, there are two categories of distance relations: *absolute* (the distance between two entities) and *relative* (the distance between two entities compared to the distance to a third entity). While the former can be represented either qualitatively or quantitatively, the latter is purely qualitative.

Qualitative Distance

In contrast to qualitative direction, where an inherent zero-point (intrinsic front, north) and cardinal systems (e.g. *"left"*, *"right"*, *"east"*, *"west"*) exist, qualitative distances depend on a context made up of complex and subjective factors, such as scale (Montello [113] proposes four different kinds of scale of space), order of magnitude, and foundational modelling assumptions. To model qualitative distance one can, for example, take into account the mobility of the navigation entity (microbes, humans, birds) which in turn depends on the modes of transport available. Other factors could be time, nutrition, environment, weather, and many more.

In general, Euclidean distance can be defined in MPLL in the same way as cardinal direction. The standard library, therefore, contains a sample definition of distances (see

Def. 4.78). However, since the actual scale for any task will certainly vary, the predefined set has to be carefully adjusted to reflect the user preferences and context.

Section 6.2 provides an overview of related work on qualitative distance. For more details on this issue see the work of Mukerjee and Joe [114], Jungert [89], Zimmermann [173, 174, 175], Frank [57], and Clementini, Di Felice, and Hernández [77, 26].

As became clear during the course of this work, representing qualitative distance is more complex than representing qualitative direction. However, in the domain of navigation and wayfinding – one of the primary fields for the application of MPLL – (multimodal) route planning techniques provide a suitable means of representing distance. Usable distance measures can be obtained in connection with a specific cost function, which very much depends on the user and context. The next section further discusses this approach.

Point-to-Point Distance

Route Planning

Due to the fact that humans often measure distances using temporal means (5-minute walk, 1 hour by car, 4 hours by plane, etc.) one possibility of modelling qualitative distances would be to use available and proven routing algorithms in connection with context and user modelling to produce means of determining the correct frame of reference for a certain domain, query, etc. In this approach, one could make use of the possibility to compare different frames of reference, since people would normally not drastically distinguish between a two hour ferry ride or a two hour train ride – albeit the distances would greatly differ. In other aspects, one could incorporate the difference between a two hour train ride and a two hour walk, because of the inherent implications. To some extent, a scale-less qualitative template of discrete distance labels (e.g. close, near, far, very far, unreachable) could be applied to different scales in order to provide generic means of reasoning for a diverse number of domains and scales.

The outcome should be an extensible table of frames of reference and their adjacency in order to answer the following questions: What modes/distances translate well into others at what scale? What scale underlies certain modes? What limits exist for different modes in what respect (e.g. cost, exhaustion)?

We believe that routing algorithms combined with adequate cost functions provide a powerful means of answering these kinds of questions under the aspect of working on a temporally oriented model of qualitative distance. This model could also be used to facilitate processing and communication of (natural language) distal expression.

Euclidean Distance

In some cases, however, there is no need for sophisticated modelling of distances. In preprocessing spatial information, for example, many operations only pertain to entities in the vicinity of the user. The generation of routing instructions at a certain point along the route does not involve each and every landmark in all available databases at hand. To the contrary, processing becomes cumbersome if there is no quick way to determine the k-nearest landmarks which are relevant for the individual task. In these cases, well

established algorithms, e.g. geometric k-nearest neighbour search, are used to filter out likely candidates for the current processing task. The k-nearest neighbour search is a prominent example, because in generating routing instructions one of the key issues is to determine suitable landmarks and to test the spatial relations between landmarks and the user's position for clear and unambiguous constellations.

Point-to-Line Distance

This setup also allows for a number of different interpretations. We only briefly sketch some ideas here, to exemplify the main issues. As mentioned before, MPLL offers a function to derive the closest segment of a linear feature: $\texttt{closestSegment}(L,C)$ of type $\texttt{Line}*\texttt{Configuration} \mapsto \texttt{Line}$ (see Def. 4.71). This function can be used in order to examine the shortest Euclidean distance to a possibly very large linear feature.

In other cases, the semantics of the spatial entities involved could necessitate different interpretations. From the point of noise emissions, for example generated by a highway, the spatial semantics are relatively simple: notwithstanding the influence of building development and vegetation, the noise emissions primarily depend on the geometric distance to the highway. From the point of transport networks, however, the semantics are different. A highway cannot be entered anywhere, but only at designated points, namely on- and off-ramps. The distance from any location to the highway can now be transformed into a proximity relation between points, whereas these points represent nodes in transport networks. Yet again, this particular problem involves route planning.

Line-to-Line Distance

Determining the distance between two linear features (see Fig. 2.12) is not a trivial issue, since there exist different notions of distance. The statement *"the river runs along the road"*, for example, is vague in the sense that it completely lacks any quantitative criteria. Once again, the context determines what *along* means, in the form of an upper bound of an interval (e.g. $[0,200\ m]$). A suitable distance metric would be the shortest geometric distance between a point on the first linear feature to any point on the other linear feature.

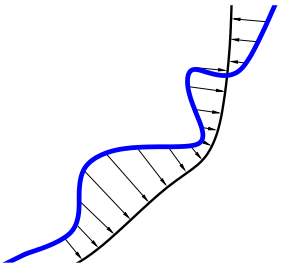


Figure 2.12.

Region-to-Region Distance

Topological Distance

Topological distance can be represented by an order of topological relations, for example, the RCC-8 relations. The order of the relations determines a qualitative distance in the sense that transitions between relations always imply a movement closer to (or farther

away from) a region. In this concrete example, the relations are ordered as follows, from *near* to *far*: $EQ \leftrightarrow (TPP/NTPP) \leftrightarrow PO \leftrightarrow EC \leftrightarrow DC$. Transitions between these relations are unambiguous. For the relations TPP and $NTPP$, however, distal order cannot be clearly determined, since there exist equally qualified examples for both alternatives. Here, the inverse cases TPP^{-1} and $NTPP^{-1}$ need not be handled separately.

2.5.3. Topological Relations

Topological relations between spatial entities play a fundamental role in spatial knowledge. Due to the inherently qualitative nature of these relations, a strong connection to human spatial reasoning seems not far fetched. However, publications about research backing up this statement are very sparse, although topological relations undoubtedly represent an important aspect of qualitative spatial reasoning.

Topological approaches to qualitative spatial reasoning usually deal with regions (rather than points) which are subsets of topological space. Most formalisations are based on work from Whitehead [169] and Clarke [24, 25] who axiomatised mereotopologies using a single relation, the binary connectedness relation. For a general overview of mereotopology, see, for example, Cohn and Varzi [34, 32, 33].

One of the most well known approaches in this field is the Region Connection Calculus (RCC) by Randell, Cui, and Cohn [134] which defines eight relations known as the RCC-8 relations. Renz [137,138] provides a comprehensive discussion of RCC-8 and topological reasoning. Further reading includes, for example, Asher and Vieu [7], Cohn [28,27], and Cohn et al. [29].

A different approach was proposed by Egenhofer [48], some discussion of his work and a comparison to RCC can also be found in the work of Renz [137].

Topological reasoning is not within the primary scope of this work. However, the necessary infrastructure to include a topological reasoning mechanism into the MPLL system architecture as an external module or service is available. The use of such a reasoning mechanism is applicable to a variety of spatial tasks, especially complementing the available quantitative methods.

Region Connection Calculus

As shown in Fig. 2.13, there exist eight RCC-8 relations [134] based on the notion of *connection*. Two regions are either disconnected (DC), externally connected (EC), overlap partially (PO), overlap fully (TPP, NTPP), or are equal (EQ). Each case of full overlap allows for the inverted relation (i.e. *contains* vs. *contained by*), This relation is further distinguished by whether the borders of the two regions share a common point. Fig. 2.13 also shows the possible transitions between the different states.

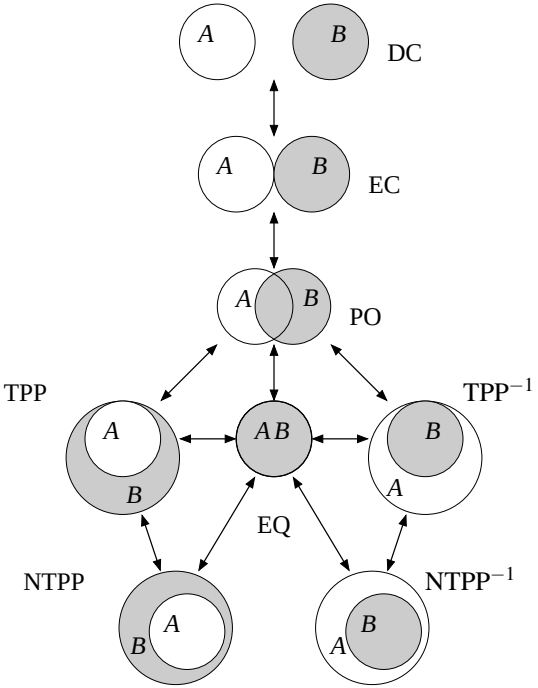


Figure 2.13.: RCC-8 Relations

9-Intersection Model

A different model of topological spatial relationship was developed by Egenhofer [48] independently from the Region Connection Calculus (RCC). The relations between spatial entities are, according to Egenhofer, defined by the possible intersections of the entities' *exterior*, *boundary*, and *interior*. The model is, therefore, called the *9-intersection-model*. Here, in contrast to RCC, the domain of spatial entities is restricted to simply connected planar regions, which cannot have holes, i.e. two-dimensional regions whose boundary is a closed Jordan curve.

The RCC and the 9-intersection-model result in the same set of topological relations, apart from the different restrictions on spatial entities. This fact substantiates that these two approaches define a reasonable level of granularity of topological relations. As already mentioned, Renz [137] gives a fine introduction to the 9-intersection-model, along with a detailed description of RCC-8 and a comparison of the two approaches.

2.5.4. Complex relations

There exist many spatial relations which are very specific to a certain scenario or a certain problem domain. Mostly, these relations are specifically designed to be applicable to a small number, and not a broad variety, of tasks. Some examples for such relations are listed in this section under the common notion of *complex relations*. This is done not because these are so much harder to compute or to define, but because there very often exist several specific preconditions and requirements for their application.

Generally, these relations do not only rely on angular and/or proximity relations, but on other properties, such as shape, size, or combinations thereof. The representation of the regions of acceptability associated with a given relation is also called *spatial template* [102]. A function definition should, therefore, define the spatial templates in light of the intended interpretation.

Order Relations

One good example are order relations of different kinds, such as *"the n-th"*, *"before/after"*, or *"next to"*. These rely on structures, which facilitate ordering of some sort, such as house numbers, room numbers, or the numbered streets in Manhattan. However, explicit numbering is not a requirement. The implicit order of stations along a railway line, albeit not numbered, serves the same purpose. This also applies to other, purely structurally ordered entities, for example a sequence of traffic lights along a road, blocks on a city grid, or side streets branching off a main road. Yet again, the interpretation of such notions is not unambiguous and should, therefore, be reflected by the individual definition.

Peculiarities of natural language also influence these notions. The notion *"next to"*, for example, has shown to be strictly interpreted as horizontal proximity (see Logan and Sadler [102]), while a number of other notions exhibit ambiguities. As an example, the notion *"between"* (as sketched in example 1.3 on page 5) serves to illustrate the variety of interpretation. The individual modelling of other notions, albeit highly specific, can be achieved in essentially similar manner.

Between

The notion of *between* can be defined in multiple ways. In a network based environment, it makes sense to define the notion of *between* based on distances within the network, since free traversal of space is usually not possible. Note the distinct difference of this approach to the following angular definition. It is also important to point out that the following definitions do not represent actual MPLL definitions, but mere pseudo-code formalisations in order to make the intended functionality clear.

Distal Definition

Let $\texttt{route}(B,C)$ denote the shortest route from B to C, and let $\texttt{route}(B,C)^{-1}$ denote its inverse, i.e. the reverse route $\texttt{route}(C,B)$. Furthermore, $\texttt{length}(\texttt{route}(B,C))$ shall yield the length of this route. Thus, $\texttt{between}(A,B,C)$ shall render a value between

0 and 1. The result is 1 if A is exactly on the route between B and C. The longer the detour through A is, the smaller is the value of $\texttt{between}(A,B,C)$.

$$\texttt{between}(A,B,C) = \frac{\texttt{length}(\texttt{route}(B,C))}{\texttt{length}(\texttt{route}(B,A)) + \texttt{length}(\texttt{route}(A,C))} \tag{2.3}$$

However, there are special cases in which this approach yields unintuitive results. If A is for example not at all located (in the classical, common sense) between B and C, but instead very close to B (C) on the opposite side of C (B), then the given definition would still render results close to 1, although this is against common sense.

Furthermore, this definition is not necessarily symmetric. This is the case, for example, when $\texttt{route}(X,Y)$ operates on a directed graph data structure, i.e. that pairs of X,Y exist, with $\texttt{route}(X,Y) \neq \texttt{route}(X,Y)^{-1}$. A street network consisting partly of one-way streets is a good example for an environment, where such routes can occur. The distal (route-based) definition of *between* is therefore not symmetric; $\texttt{between}(A,B,C) \neq \texttt{between}(A,C,B)$.

Angular Definition

The angular way to define *between* checks whether there can be a straight line drawn through the locations B, A, and C. The greater the deviation from the target value 180° is, the less A lies *between* B and C. The following definition is crisp. However, for practical purposes a definition involving fuzzy values would be preferable, returning values between 0 and 1 depending on the deviation of the target value. Here, we need to use the *bearing* (see section 2.4.2) between locations, since the orientation of A, B, or C is not relevant.

$$\texttt{between}(A,B,C) == (180^\circ = |(\theta_b(A,C) + \theta_b(B,A))|) \tag{2.4}$$

2.6. Landmarks

Because of their fundamental role in human spatial cognition and wayfinding, *landmarks* are of key importance also within MPLL. Intuitively, a landmark is any spatial entity which the user can relate to, in order to solve a given spatial task. The locations where the user is supposed to conduct an action or where the user is supposed to verify his or her position, can be marked by their spatial relation to, or the presence of, one or more landmarks. This spatial relation can, for example, be distal (the landmark denotes a certain distance, e.g. *"go straight until you reach the bridge"*) or angular (the landmark provides a direction, e.g. *"go towards the church"*).

Landmarks in MPLL

As MPLL already provides the necessary data types to model landmarks, there exists no special "landmark" data type. Landmarks are often modelled as point entities which optionally feature an orientation. Therefore, they can be represented as points (see Def. 4.21, pp. 4.21) or configurations (see Def. 4.23, pp. 4.23). Landmarks without

an intrinsic front can be represented without an orientation property. They can be represented either by a point entity or by a configuration, whereas the orientation property of the latter is not used. For example a tree, a TV tower, or a smokestack fall into this category. Those entities with an intrinsic front can only be represented by a configuration. This concept applies, for example, to buildings with a clearly perceivable main entrance (e.g. a church).

Furthermore, landmarks can also occur as linear features (e.g. rivers, railway lines) or as regions (e.g. parks, lakes, parcels). MPLL also provides suitable data types to represent such landmarks, along with the necessary spatial relations. These relations have been introduced in section 2.4.2, the equivalent MPLL functions can be found in sections 4.5.2 and 4.7.2.

In this section, we give a definition of the term *landmark* and the closely related notion of *named entities*, followed by a discussion of the purpose of landmarks in wayfinding and other possible applications. A short discussion of remaining challenges, especially regarding landmark acquisition, concludes this section.

2.6.1. Definition

One of the most broadly accepted definitions of landmarks has been provided by Kevin Lynch [107] (pp. 78). This definition is also very suitable in the scope of this work, given the geospatial nature of the underlying domain:

> "Landmarks, the point references considered to be external to the observer, are simple physical elements, which may vary widely in scale. [...] Since the use of landmarks involves the singling out of one element from a host of possibilities, the key physical characteristic of this class is singularity, some aspect that is unique or memorable in the context. [...]"

Furthermore, other definitions confirm this statement or are merely of supplemental character: "[A landmark is a] monument or material mark or fixed object used to designate a land boundary on the ground: any prominent object on land that may be used to determine a location or a direction in navigation or surveying" [50], "[landmarks are] point references considered external to the user" [107]. Even in rather complementary fields (e.g. real estate, law, military), landmarks are used in a similar manner, for example by Axel Pinz [133]. Several Web resources, for example answers.com [4], point to a number of different uses and definitions of the term "landmark" in different domains.

As an extension to these definitions, one might add that landmarks can also be non-physical – at least in the sense of human perception. For a blind person, who cannot see a landmark, certain other features, such as aural or olfactory, might serve the purpose. Whereas a landmark's aural (and other) properties mostly depend on its physical features (e.g. the sound of cars crossing an intersection which features rails of a tramway, or the

acoustically shielding effect of a big building), olfactory properties pertain more to the use of a landmark itself (e.g. the smell of a brewery or a bakery), and not its physical features. Exceptions notwithstanding, the above given definition is sufficiently correct for the given problem domain.

Landmark Integration

Incorporating landmarks into route data structures poses an interesting problem, as they are only very rarely part of the route itself.

While landmarks are used for a variety of purposes, in light of this work, they primarily serve as navigation and routing aids. They can denote a location where a certain action is to be performed: *"Turn right at the supermarket."* They also serve as directional aids: *"Go into the direction of the cathedral."*, *"Follow the river on your right hand side."* Used as *reassuring cues* they denote a route simply by their presence.

Key Properties

The key properties which characterise a landmark are the following (These attributes, however, are not all mandatory, since in some cases just one of these properties is sufficient for the object to be treated as a landmark.):

Contrast to Environment – One of the principal factors is the contrast to its environment which an object can generate. This contrast can manifest itself through a number of features, most importantly through form, colour, size and other primary features (see below), but also through several secondary features. Clean buildings in otherwise unclean surroundings (or vice versa) can perfectly serve as landmarks. The same applies to objects which only stand out by their orientation, their architecture (modern vs. antique), or similarly composite features.

Contrast to the environment is not confined to the immediate surroundings. Some objects have features which stand out in the wider area of a city or region, such as TV towers, waterfalls, or mountains.

Clear Form – This feature is often connected to size, although the two are not dependent. Clarity of form is stronger with objects which offer clear clues as to their form, size and the angle of the current view. This can be the result of certain (a)symmetrical shapes. Sometimes, the clarity comes from the fact that the object has *the same* shape, independent from the angle of the view. This is the case, for example, with axis-symmetrical buildings such as domes, TV towers, smokestacks, etc.

Spatial Prominence – Taller objects tend to be visible in farther distances and, therefore, they usually represent good candidates for landmarks. In addition to size, the location of an object can add to that effect. Churches and castles, for example, have often been built on hilltops or other prominent locations.

However, the more suitable an object is as a 'long-distance' landmark, sometimes its usability in smaller scale environments is decreased. The John Hancock Building or the Prudential Tower in Boston, for example, are huge structures also at

their base. They are too big to be in contrast with the surroundings since *they are the surroundings*. Therefore, depending on the scale, they can be very useful, or less so, to serve as landmarks.

Symbolic Significance – In some cases, contrast with the background is achieved by *precisely not* being significant. Some objects stand out by their special quality of being rather inconspicuous or unobtrusive. This can also occur in connection with the abstract importance of a structure, for example when there is a huge discrepancy between the importance of an institution and its premises.

Artificial vs. Natural Landmarks – Landmarks can be natural or artificial [15]. *Natural* landmarks are a normal part of the environment, while *artificial* ones have been specifically placed for some perceptual activity. Street signs or room numbers are artificial, while buildings and doors are natural.

This distinction is more important for other domains, such as robotics, for example. It is presented here purely for the sake of completeness since humans do not tend to treat artificial and natural landmarks very differently. Robots, however, due to their sensory limitations, might have to rely on a specific type of landmark and cannot utilise a broad variety of types.

2.6.2. Named Entities as Landmarks

Named entities are expressions which occur in natural language text and which denote, by name, certain spatial (e.g. *"Beacon Hill", "Boston Common"*), temporal (e.g. *"the nineties", "prehistoric"*), or spatio-temporal (e.g. *"post-war Europe", "14th century France"*) entities. These entities can be, for example, persons, organisations, locations, epochs or eras. Although named entities are also called *Proxy Place Names* [6], we stick with the more general term as used in the section header.

Within the scope of this work, we want to focus on named entities denoting *locations*, which, due to common knowledge about them and their often unambiguously clear form, can almost always serve as landmarks with their inherent properties and features. For completeness, however, we give a generic definition of named entities in this section.

Definition Named entities are expressions, i.e. natural language words or phrases that contain the names of entities, for example persons, organisations, locations, times, quantities, etc. The following example features three named entities which are marked by a `type: value` pair within square brackets[15]:

Chancellor [`PER: Merkel`] meets
[`ORG: U.N.`] representative in [`LOC: New York`].

[15] Types in this example are: `PER` (persons), `ORG` (organisations), and `LOC` (locations).

Named Entity Recognition (NER) is a subtask of Information Extraction which is highly language-specific. Due to this fact, different systems employing language-specific resources to accomplish the task show varying performance, and it is unknown how they perform on text from languages other than the specific target language [131].

History

Since 1995, NER systems have been developed for some European languages and few Asian languages. Palmer and Day [131] used statistical methods for finding named entities in news articles in Chinese, English, French, Japanese, Portuguese and Spanish. They found that the difficulty of the NER task was different for the six languages, but that a large part of the task could be performed with simple methods. Cucerzan and Yarowsky [37] used both morphological and contextual clues for identifying named entities in English, Greek, Hindi, Romanian and Turkish. We do not go into detail because further reading regarding individual techniques is readily available [11, 38, 54, 91, 109, 157, 170] and it is not of extraordinary relevance within the scope of this work.

Named Entities in MPLL

The subset of named entities which occur in MPLL, namely those denoting locations, are treated no different from landmarks. Therefore, there exist no special data types for their representation. Instead, they are modelled using standard data types, such as points, configurations, polylines, or polygons.

2.6.3. Landmarks in Wayfinding

The role of landmarks in wayfinding are manifold, although their primary function tends to be associated with providing cues to the user. As "*...trigger cues* [...] and *reassuring cues*..." [107] they provide two very important services.

Trigger Cues

Trigger cues are needed at every decision point along a route. Any decision regarding route traversal is usually bound to a location, and since landmarks are the preferred means of identifying locations, they are very often used in this respect.

Reassuring Cues

In a similar way, even when there is currently no decision point along the route segment, the user periodically needs to be reassured that he or she is still on the right track. In this context, we have to distinguish between *real* and *optional* decision points. The former are a necessary and substantial part of the route description, while the latter are optional. Optional decision points are, for example, *go straight at the traffic lights* or *continue along the main road*. Leaving out instructions like these still renders a traversable route, because the omission of the (optional) action at these points (i.e. doing nothing) leads to the same result. However, when real decision points are too far apart, exactly these optional actions serve well as reassuring cues: they tell the user that he or she is still on the right track. Reassuring cues can even be artificially inserted along the route when there is no decision point at all (not even an optional one). If the next intersection is still quite far away, the statement *"continue along the main road"*, while not bound to an immediate location, will still serve its purpose.

Single Landmarks

Situations in which spatial relations to single landmarks are relevant, are discussed at length in section 2.5. Such relations to landmarks are best usable, if they are clear and unambiguous, i.e. they locate the user with the highest possible accuracy. This can be achieved equally by using angular and distal relations. A landmark located at an intersection (distal relation), for example, makes the intersection unique in the vicinity, simply by its presence. A visible (angular relation) landmark unambiguously defines the cardinal directions *"towards"* and *"away from"*, even though, depending on the user's position, the absolute direction can vary.

More complex relations include, for example, *"alongside"*, *"into/out of"*, *"next to"*, *"around"*, *"on the outskirts"*, and so on. The formal definition of these and many more relations depends on the individual interpretation and semantics and, therefore, must be defined in a domain dependent manner.

Landmark Pair Boundaries

In case of the presence of multiple landmarks, the theory of the so-called Landmark Pair Boundary (LPB) facilitates reasoning [97, 12, 16]. On a planar map containing several (visible) landmarks, the Delaunay Triangulation produces a number of triangles which represent *orientation regions*. Within each of these orientation regions, any arbitrary position features a unique order of visible landmarks. This information can be used for the purpose of positioning, if the position of landmarks is available. In fact, for any two landmarks $l_i, l_j, i \neq j$, i.e. any landmark pair, the position p can be determined regarding the side of the line (l_i, l_j). The individual distances, e.g. $d(p, l_i)$, need not be used, but only the bearings from p to the l_n.

2.6.4. Challenges

There remain several issues which complicate the use of landmarks in spatial information processing, and which require further extensive research.

Landmark Definition

Landmarks are difficult to define due to a number of reasons, the first and foremost being that the perception of landmarks is highly subjective. Depending on the individual application and scale of spatial relations, the properties of an ideal landmark will differ substantially. Even within a certain domain, subjectivity prevails to a certain extent, mostly due to issues in connection with personal preferences, perception, and context of use. A suitable example are route descriptions for pedestrians. The ideal description for a person to traverse a certain route will be different, depending on the time of day of the travel. The applicability of landmarks varies greatly between daytime and nighttime. Also, for two individual persons (and their subjective perception of landmarks), the description of the same route will have to be adapted to their individual preferences and context. Therefore, a single route can be described by several different route descriptions – neither one of them being the only and ideal one.

Landmark Generation

As a consequence, there currently exists no universal automated process for landmark generation within a certain application domain, problem domain, or map. For some domains, a number of feature properties (location, size, shape, other geometric features)

are sufficient to determine the overall quality as a landmark. In other domains, specialised processes must further determine complex relations, such as visibility within a 3D model of a city. Certainly, the instruction *"go towards the TV tower"* is valuable only if the tower *is visible* from the user's current position. Similar aspects pertain to other senses, such as audiovisual, aural, olfactory, or tactile. The latter three are especially important, for example, for blind people.

The process of landmark generation, identification, ranking, etc. is not a part of the MPLL functionality. Therefore, MPLL relies on the services of another application (in form of an MPLL module or a (web) service) to provide these functionalities. Within the scope of this work, we presuppose the availability of suitable landmarks, whenever the individual task includes spatial relations to landmarks.

2.7. Fuzziness

The ability to process fuzzy information is eminently important in geospatial reasoning because of the way humans express their perception of space around them and because of their way to communicate this perception. They use mostly qualitative terms [13,18,56], such as *"in direction of the city centre"*, *"south of the park"* or *"close to the train station"*. While quantitative notions are also used, they often lack the necessary precision in order to be processed with purely quantitative methods: *"the pharmacy is located 300 m from the subway station"*. Of course, unambiguous quantitative expressions are also used: *"take the westbound train at North Greenwich and leave at* ***the fourth stop****"*, or *"turn left at* ***the third traffic lights****."*

Specification Mechanisms

Several issues must be addressed in connection with fuzziness. The underlying geospatial model must accommodate fuzzy data, and basic operations to manipulate it. Specification mechanisms must be provided in order to facilitate the declaration of fuzzy terms and notions for use in certain applications. This includes means for end users to specify their interpretation of *"close to"* or *"south of"*. In the same way as fuzziness can occur in a one-dimensional environment, such as a time line [122, 119], it can occur in multidimensional environments. Three or more dimensions are not discussed here, since we concentrate on one–, two– and 2.5–dimensional data. Compared to one-dimensional data, fuzzifying multidimensional data requires essentially similar, but refined, methods. There exist several different flavours of fuzziness [86], which are briefly outlined before we go into detail about the consequences of fuzziness for the underlying problem domain.

Vagueness pertains to the lack of definite criteria about the applicability of a concept. For example the statement "The chair is in the corner of the room" is not uncertain (the chair is definitely there) but vague as to the exact position of the chair.

Imprecision is an inherent feature of every GIS. Not a single data set can be defined in an infinitely accurate manner, and, therefore, every data set is inherently imprecise. Whether within the respective problem domain this means deviations of centimetres, millimetres, or nanometres does not change this fact, but only the order of magnitude of imprecision.

In the sense of imprecise geospatial features, sometimes the notion *diffuseness* is used, which in the context of fuzziness also means imprecision.

Uncertainty means the lack of exact knowledge about an object. A precise statement can be uncertain in the way "if it is correct, then it is precise", meaning there is a chance that the statement is not true altogether – regardless of its precision. If the person making the above mentioned statement about the chair is not sure whether the chair is there at all, the statement altogether is uncertain.

Ambiguity pertains to several contradictory conclusions which could be drawn from a statement. If within a route description an instruction says to go "into the direction of the church", this instruction might be ambiguous if there are two churches visible. Certainly one of the alternatives is the correct one, although it cannot be determined which one.

Generality is also a form of fuzziness as it describes statements of coarser quality. A statement such as "Let's meet at Piazza Castello." specifies a clearly defined region (the Piazza). Whether this region is sufficiently precise depends on the respective context. Generality is closely related to imprecision in the sense that both types depend on a context to define their fuzzy quality.

One-dimensional data are discussed in the next paragraph. Examples for two-dimensional data are basically all elements which are found on a standard map: streets, railway lines, rivers, lakes, districts, cities, and so on. Their geometric counterparts are points, lines, and polygons. The following sections deal with different aspects of modelling and using fuzzy logic with respect to geospatial reasoning tasks.

2.7.1. Fuzzy Intervals

Common fuzzy intervals, such as the ones for modelling fuzzy time intervals [125], can also be used in spatial information processing. Fuzzy intervals can be used in connection with one-dimensional spatial data. As laid out in this section, many cases of fuzziness – pertaining to one-, two- or n-dimensional data – can be modelled using fuzzy intervals. In two-dimensional space, for example, distance and direction can be treated as one-dimensional structures. The fuzzification of regions, however, produces a three-dimensional structure, which cannot simply be modelled using intervals. In this section, several forms of fuzziness are discussed, beginning with one-dimensional data.

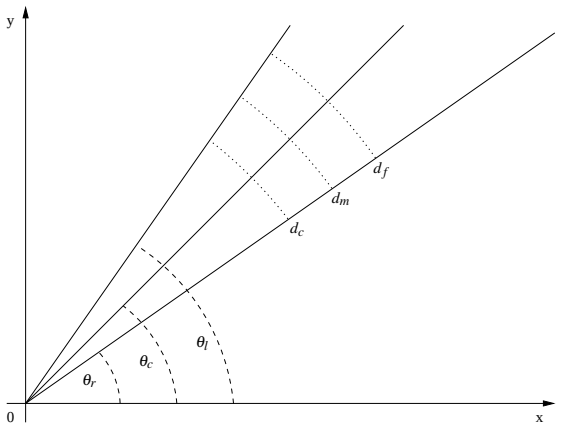


Figure 2.14.: Fuzziness in Distance and Angle

What is meant by one-dimensional spatial data? From the viewpoint of routing and wayfinding, the goal is a *route*, i.e. a linear structure which connects the start (`S`) with the destination (`D`) via a number of intermediate segments and junctions (`1-5`) as shown in figure 2.15 a).

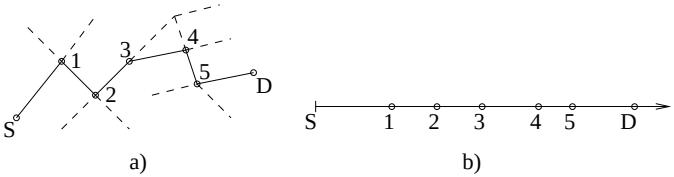


Figure 2.15.: Representations of a Route

This route – albeit a fully two-dimensional (or three-dimensional) entity – can be handled as a *flat* structure in order to reduce complexity in processing. This means that not the coordinates of the start or destination or intermediate nodes need to be regarded, but only the distance of a certain point along the route to the start (or destination). If there is an intersection at a certain point, its location is marked only as the distance to the start. If all intermediate nodes are marked this way, the result is a one-dimensional structure in which the origin denotes the start and the destination the highest value on the axis, as shown in Fig. 2.15 b).

The motivation for this type of modelling approach is that the resulting one-dimensional structure is the ideal basis for classic fuzzy calculations. Many fuzzy aspects which are relevant for routing can be incorporated this way. In a region with poor Global System for Mobile Communications (GSM) coverage, for example, the fuzzy value along the route could indicate the reception quality for mobile phones, ranging from 0 (none) to 1 (excellent). Converting distances to GSM base stations along the route into fuzzy values provides a simple way to ascertain GSM reception for any point along the route. Many other route characteristics can be modelled this way: inclines, friction coefficients, curve radii, terrain attributes such as flora or housing density, and more.

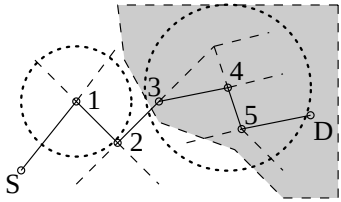
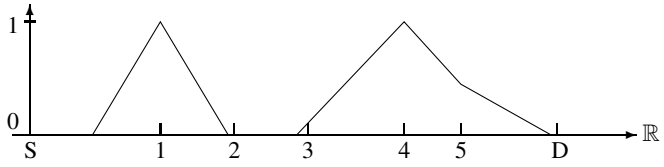


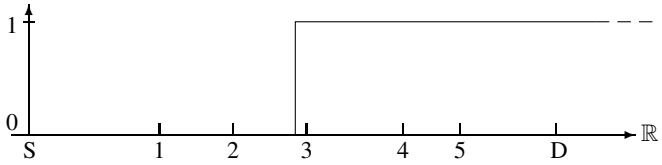
Figure 2.16.: Extended Route Features

Fig. 2.16 shows an enriched version of Fig. 2.15. The dotted circles denote the maximum range of GMS base stations, the grey polygon marks an urban area. The GSM signal strength for any location along the route can now be deduced by calculating the distances to the nearest GSM transmitter in the vicinity. Reception quality (i.e. signal strength, degradation, and attenuation) is, technically, not linear over distance and depends on several more complicated factors, but for demonstration purposes this approximation will suffice. The result is a fuzzy interval like the following:



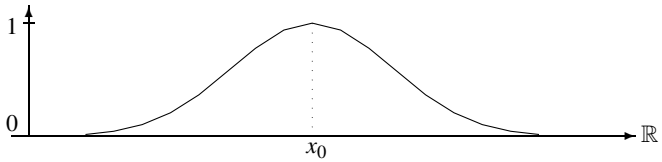
GSM Reception

An interval marking urban areas can be applied in a similar way. The following diagram shows the urban boundaries as a crisp interval along the route from S to D:



Affiliation to urban areas

Nonlinear functions can also be used to compute fuzzy intervals. If the coverage of GSM base stations degrades in a nonlinear way, the interval can be adapted to reflect this effect. Other, more complicated cases can be computed as long as a suitable formula is given.



Gaussian Shape

A clear advantage of handling spatial fuzzy intervals this way is the availability of standard fuzzy operations. If several different fuzzy (and/or crisp) intervals are to be combined, many standard operations can be used, such as the Hamacher Family (see, for example, Ohlbach [122, 119, 118] for a comprehensive overview):

Hamacher Intersection and Union with $\beta = \gamma = 0.5$

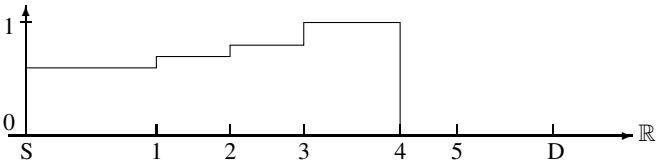
Using these methods, it is very easy to compute different fuzzy intervals or the union of different intervals for a given route. This way, different queries pertaining to fuzzy information can easily be answered: *"At what approximate time will we cross the border to France on the journey from Munich to Paris?"*, *"During this journey, are there any portions where a mobile phone cannot be used or incoming calls might not get through?"*, *"Does the journey lead through mostly rural or urban areas?"*, etc.

Landmark Integration

Fuzzy intervals such as the above can also be used to integrate *landmarks* into a route. Fuzzy representations are sometimes particularly well suited to handle references to landmarks.

Directions, for example, are rarely crisp. Therefore, a fuzzy interval which denotes the fulfilment of a typical spatial relation such as "go uphill" or "go towards the church" could be computed by crisp means only with difficulties, while fuzzy representations are quite elegant. The reason for this is that slight deviations are problematic. Certainly, there exists a metrically correct translation of "south of", for example 180°. Furthermore, regardless if either the crisp interval $[179°, 181°]$ or $[135°, 225°]$ should qualify in the same manner as "south of", there always remains the problem of the interval's crisp definition. A bearing just 1° outside the respective interval would not qualify as "south of", while another bearing just inside the interval would fully qualify. Considering the vague way of humans to deal with direction and distance, this representation is not satisfactory. Fuzzy representation can offer a much more suitable model, as is shown in the following example.

We assume that on a smaller scale copy of the scenario described in Fig. 2.16 a church is located at junction 4, and we regard the spatial relation between the user's current position and the one of the church. The fuzzy values for the different route segments regarding the relation "towards the church" could look like this:



A fuzzy spatial relation: "towards the church"

In general, a computation by crisp methods needs to substitute for these mechanisms and therefore somehow include special treatment of "near misses". If the instruction is "go towards the church", would a segment which deviates from the absolute direction to the church by only a few degrees be disregarded? What happens if there are no other alternatives, in particular not a single one satisfying the instruction (i.e. if we do not deal with navigation in free space, but with planning on a network of predefined routes)? What happens if there are several equally suitable alternatives? These questions also arise when using fuzzy logic, but solving them is much easier.

Other Applications

Directional conditions can also be time-dependent. Two different routes are, for example, rather equal regarding length and travel time. One starts off to the south and then turns to the east while the other starts off to the east and then turns south. If a constraint is defined as "not driving towards sunshine" (it might be winter time and visibility is affected due to the low position of the sun[16]) there is a significant difference between the first and the second route *depending on time*. If the journey starts in the morning and ends around noon, the second route would violate the constraint most of the time, while

[16] Also, for this example to make sense, the journey must take place on the northern hemisphere.

the first would not. At other times during the day, both routes could still be quite equal or the situation could be reverse. There certainly are many other examples which could make use of one-dimensional fuzzy intervals in this way.

2.7.2. 1.5-Dimensional Distributions

Different notions of fuzziness lead to two-dimensional fuzzy distributions which are axially symmetric to the z-axis. These distributions can be grouped in a class which is neither one-dimensional anymore, nor fully two-dimensional, and that we refer to as 1.5-dimensional distributions.

The reason for distributions in this class being axially symmetric is that they represent *omnidirectional* fuzziness which occurs in reference to a single location in space (a point) and does not take additional parameters, for example *direction*, into account. The same applies to *directional* fuzziness which does not take distance into account. The latter form, inherently, leads to distributions of similar shape with different orientation regarding the reference system. More detail on this issue can be found in section 2.4.2.

For the subsequent examples, the point of reference p is located at the origin of the coordinate plane, defined at $(0,0,0)$. The fuzzy value of any point p' with the planar coordinates (x,y) in relation to p is given as the value of z in $p'(x,y,z)$.

Linear Proximity

Linear proximity results in a linear decrease of the fuzzy value denoted by z with increasing horizontal distance to the point of reference p:

$$z = \begin{cases} 1 & \text{if } d \leq r_1 \\ max(0, min(1, 1-((d-r_1)*l))) & \text{otherwise} \end{cases}$$

with $0 \leq l \leq 1$ being the factor of linear decrease. See Fig. 2.17 and Fig. 2.18 for illustration.

Logarithmic Proximity

In case of logarithmic proximity, with increasing horizontal distance to the point of reference p, there is a logarithmic decrease of the fuzzy value denoted by z:

$$z = \begin{cases} 1 & \text{if } d \leq r_1 \\ max(0, min(1, 1-((ln(d)-r_1+c)*l))) & \text{otherwise} \end{cases}$$

with c as a constant shift of $ln(d)$ and $0 \leq l \leq 1$ being an additional linear factor. See Fig. 2.19 and Fig. 2.20 for illustration.

Exponential Proximity

In case of exponential proximity, increasing horizontal distance to the point of reference p results in an exponential decrease of the fuzzy value denoted by z:

$$z = \begin{cases} 1 & \text{if } d \leq r_1 \\ max(0, min(1, 1-((e^{d-c}-r_1)*l))) & \text{otherwise} \end{cases}$$

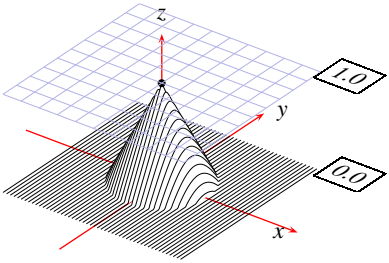


Figure 2.17.: Lin. Proximity ($r_1 = 0$)

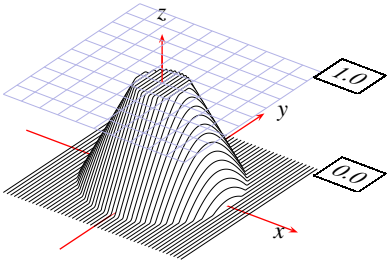


Figure 2.18.: Lin. Proximity ($r_1 > 0$)

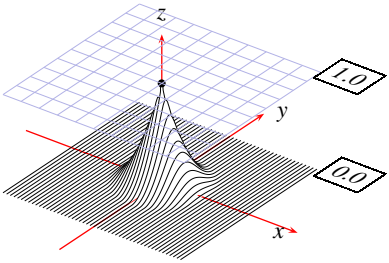


Figure 2.19.: Log. Proximity ($r_1 = 0$)

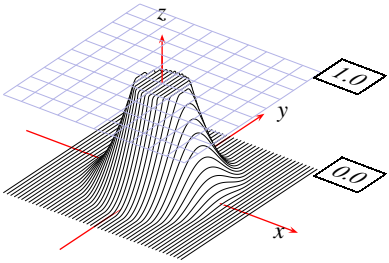


Figure 2.20.: Log. Proximity ($r_1 > 0$)

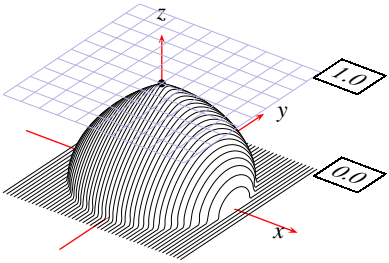


Figure 2.21.: Exp. Proximity ($r_1 = 0$)

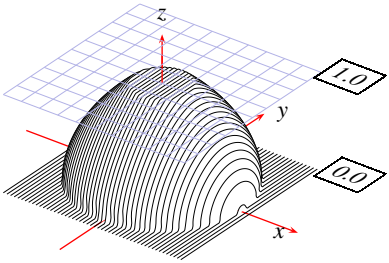


Figure 2.22.: Exp. Proximity ($r_1 > 0$)

with c as a constant shift of e^d and $0 \leq l \leq 1$ being an additional linear factor. See Fig. 2.21 and Fig. 2.22 for illustration.

Axially symmetric fuzzy distributions can easily be modelled using fuzzy intervals, because the fuzzy value is solely depending on the distance to the reference point p. The MPLL type `Interval`, along with the functions provided, is well suited to cover 1.5 dimensional fuzziness.

Proximity and Distance

The previously presented modelling of fuzziness required the fuzzy value at the origin to be always 1, hence we call these cases *proximity fuzziness*. However, some scenarios might require a different notion of distance, as, for example, depicted in Fig. 2.23.

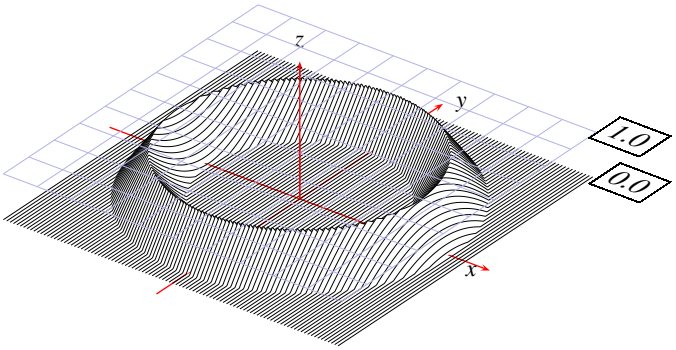


Figure 2.23.: Fuzzy Notion of Distance

This case, as well as many other scenarios, can also be modelled using regular fuzzy intervals. As long as the calculation of a position's fuzzy value depends only on the distance (independent of the distance metric used) to a single reference location, regular fuzzy intervals are sufficient. More complex scenarios might, however, require a different modelling, for example if many factors are used in the process of determining a two dimensional fuzzy distribution in which each position's fuzzy value depends on a number of factors and spatial functions.

The key issue in this discussion is that, within a reasonable range of likely spatial scenarios, 1.5-dimensional fuzziness can be modelled, processed, and generally treated as regular fuzzy intervals. There is usually no need to employ special techniques and mechanisms. All the different variants of fuzziness presented above can be solved with the regular set of tools which are provided for fuzzy intervals.

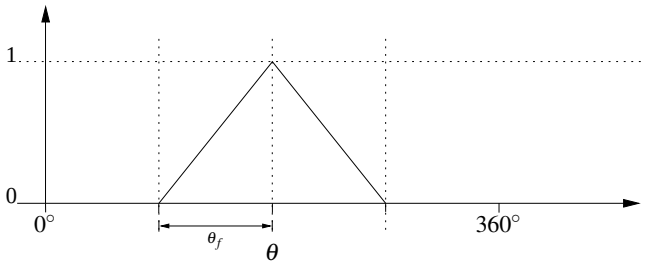


Figure 2.24.: Fuzzy Notion of Direction ($\theta_c = 0$)

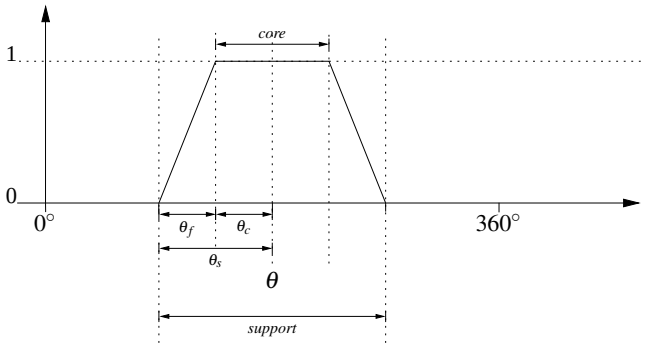


Figure 2.25.: Fuzzy Notion of Direction ($\theta_c > 0$)

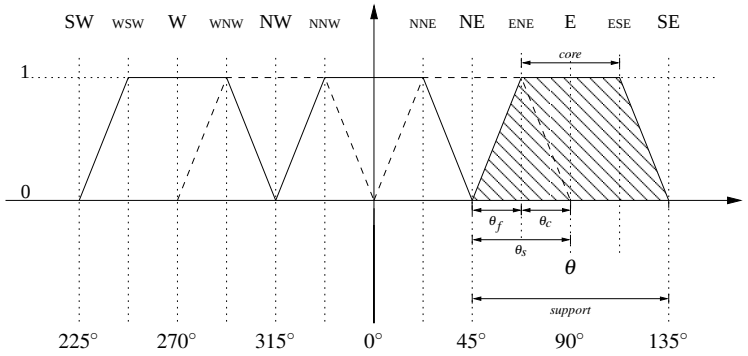


Figure 2.26.: Fuzzy Representation of Cardinal Direction

2.7.3. Directional Fuzziness

Direction

If *quantitative* means are employed to perform spatial information processing of *qualitative* data, suitable models are needed to cater for the inherently different properties of the two domains.

Qualitative notions are not precise numeric values, but more often a range of values, which fulfil a certain quality. Fuzzy intervals constitute an accepted means for modelling such data. This also applies to angular expressions.

Fig. 2.24 and Fig. 2.25 illustrate intuitive forms of modelling direction as a fuzzy interval. Provided that an angular value θ determines an absolute angle, any other angle can be related to θ by the fuzzy value it marks in the interval. Angles equal to θ result in a fuzzy value of 1. Angles within the interval of $\pm\theta_f$ around θ result in a fuzzy value in the interval $[0,1]$. All other angles result in the fuzzy value 0.

If necessary, a core interval $\pm\theta_c$, $\theta_c > 0$ can be defined to create an interval resulting in a fuzzy value of 1 (see Fig. 2.25). Parameters θ_f and θ_c can be modified to reflect the semantics of the angular expression.

A three-dimensional illustration of the directional fuzzy notions depicted in Fig. 2.24 and Fig. 2.25 is shown in Fig. 2.28 and Fig. 2.29 (see page 79) respectively.

Cardinal Direction

Fig. 2.26 shows a possible fuzzy representation of equally distributed cardinal directions. Eight partitions denote the fuzzy value of a certain direction with a core of 45° ($\frac{\pi}{4}$) and a support of 90° ($\frac{\pi}{2}$). Illustrated by the hatched area, the extent of the directional notion *east* (E) spans 45° (from 67.5° to 112.5°) at the core value of 1. Outside of this interval, it spans 90° (from 45° to 135°) at values between 0 and 1. This ensures that values close to 90° fully qualify as "east" and values close to (but outside) the core interval qualify to a certain extent as "east". This distribution shows no overlapping between contradictory assignments; i.e. no angle can both qualify, for example, as "north" *and* "east", whereas there is intentional overlap between, for example, "northeast" and "east".

2.7.4. Two-Dimensional Fuzzification

In the same way a one-dimensional structure can be fuzzified by transforming it into a two-dimensional structure, a fuzzified two-dimensional structure can be represented by a three-dimensional structure. Two-dimensional shapes represent many different elements commonly found in a map. A series of line segments can represent a highway, railway line or a river, but also abstract elements like border lines or flight connections. Polygons denote all elements which occupy an area of some sort, either real or abstract: cities, lakes, woods, farm land, districts, and so on.

Fuzzification of these shapes is especially significant because of their importance for navigation and their common use in interpreting and communicating geospatial data. Real-life expressions like "near the river" or "in the south of Munich" can only be processed correctly, if there exists an equivalent counterpart in the model representation.

The first expression means that a linear shape (the river) has to be expanded by fuzzyfication into a polygon which encompasses the area denoted by the term “near” in a fuzzy way. To be more exact, the two-dimensional shape becomes a three-dimensional shape which looks like a ridge (see Fig. 2.30). The second expression implicates the transformation of a polygon (the city of Munich, simplified in Fig. 2.27 a) into a trapezoid-like shape. In two separate steps, the polygon has to be clipped (2.27 b, c) to satisfy the notion of “the south of...” and the resulting polygon must be fuzzified in order to represent the notion of “within” (2.27 d).

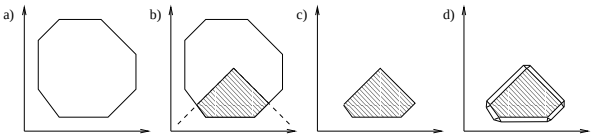


Figure 2.27.: Fuzzification of a Polygon

Lines and Polylines

The fuzzy expression *“near the river”* can be modelled as a fuzzy region around a linear shape as shown in Fig. 2.30. The closer a point is to the polyline object representing the river, the higher its fuzzy value according to this constraint. The previously two-dimensional shape, a series of line segments, has become a three-dimensional ridge following the former line segments. The lateral shape in this example is also linear – hence the conical profile – but can also be of a different type, for example logarithmic or exponential (see figures 2.17 through 2.22 on page 74).

A more complex fuzzy variant of linear features could modulate the fuzzy value of the linear feature itself. In the previous example, a point with no distance to the line scored a fuzzy value of 1. This value could be influenced by other factors as well, which would lead to a three-dimensional fuzzy distribution. In this distribution, the z-value would indicate the fuzzy value at the respective position in planar space. Hence, two-dimensional features can be represented by three-dimensional fuzzy distributions.

Polygons

Essentially similar to line features, the fuzzification of polygons necessitates the same means. A two-dimensional feature in planar space leads to a three-dimensional fuzzy distribution. The only difference between lines and polygons is that a polygon has an interior and an exterior. However, this is only another factor of many, which, in the end, determine the fuzzy value for an arbitrary location (regarding the polygon) in planar space. For example, if the polygon were shaped like a circle (e.g. by adding a huge

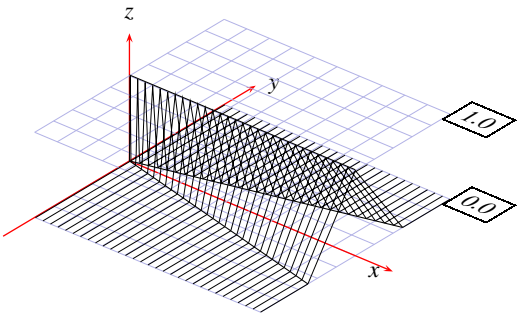


Figure 2.28.: Fuzzy Notion of an Angular Value ($\theta_c = 0$)

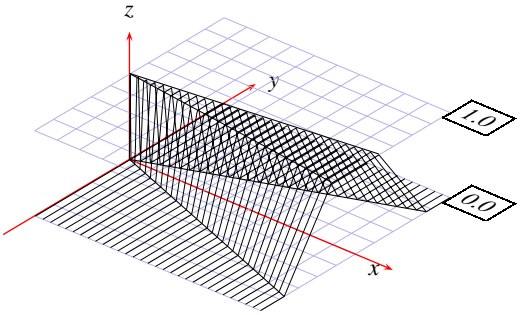


Figure 2.29.: Fuzzy Notion of an Angular Value ($\theta_c > 0$)

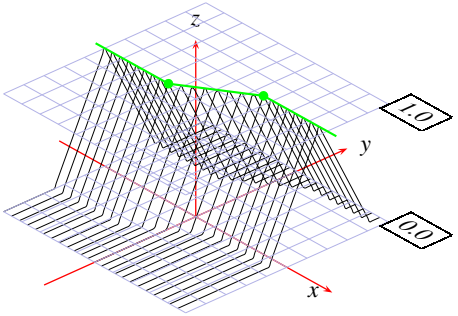


Figure 2.30.: Linear Fuzzification of a Line Segment

number of points as an approximation), then a fuzzy distribution expressing geometric distance from the "ring" would look similar to the distribution shown in Fig. 2.23 (see page 75).

2.8. Context and User Modelling

To facilitate the specification of individual operations and metrics means to provide a basic set of functions which can be used to generate composite functions. As laid out in detail in the previous sections of this chapter, desirable individual functions pertain, for example, to proximity or direction, such as *"close to"* or *"south of"*. Because almost always there exist multiple interpretations of these notions, there must be a way of using unambiguous basic notions to define them.

Common Definitions

Most spatial relations depend on the context, the user preferences, and the user profile. Generally, all factors that influence these relations can be summed up in the category *context*. In the literature, we do not find clear evidence of consensus in the definition of the notion of context. There are several approaches on how to encode and represent context.

Representational Approach

Context is, for example, defined as *"location and the identity of nearby people and objects"* by Schilit and Theimer [144]. Ryan, Pascoe, and Morse [142] give a similar definition: *"location, identity, environment, and time"*. In their investigations of context-based computing, Dey, Abowd, and Salber [43] give a broader definition. They define context as *"any information that can be used to characterize the situation of entities"* and elaborate further: *"typically the location, identity, and state of people, groups, and computational and physical objects"*. The broadest definition, though, is given by Schilit, Adams, and Want [143]: *"Context encompasses more than just the user's location, because other things of interest are also mobile and changing. Context includes lighting, noise level, network connectivity, communication costs, communication bandwidth, and even the social situation; e.g., whether you are with your manager or with a co-worker."*.

Interactional Approach

In more recent work, Dourish [46] comes to the conclusion that a *representational* approach might not be satisfactory in all cases. He proposes to treat context as an *interactional* problem: *"context isn't something that describes a setting; it's something that people do"*. He further makes four assumptions:

- contextuality is a *property* of information; information may or may not be relevant to some particular activity
- contextual features are not defined a priori, their *scope is dynamically defined*
- contextual features are *not stable*, they change over time
- context arises from activity, it is *actively produced*

Definition

Without favouring any approach in particular, we also propose a rather broad definition of context. Within the scope of this work, context is any kind of information, data, or property pertaining to spatial entities, devices, systems, the environment, culture, space, or time, exerting any influence on the semantics of the processing, reasoning, or execution of spatial relations, routing and navigation, or related tasks.

The solution to a certain problem, for example route planning, changes considerably depending on such factors: It is important whether it is daytime or nighttime, whether it is raining or not, whether today is a holiday or not, whether the user is able to speak the language spoken at the current location, of what composition the group is, what devices are used, what the current location is, and many more.

An apt examination of context and user modelling is not within the scope of this work. Many researchers are working in this broad and complex field. To name but a few different issues, for example Aroyo, Denaux, Dimitrova, and Pye [5] are working on ontology-based user knowledge acquisition. Agarwal, Huang, and Dimitrova [2] deal with individual ontologies for personalisation. There are also several publications available within the REWERSE Network of Excellence, e.g. by Henze [74, 75, 73], by Baldoni, Baroglio, Martelli, Patti, and Torasso [9], and by Abela and Montebello [1].

2.9. Summary

This chapter introduced the basic concepts of the underlying domain of geospatial information processing. As laid out in the next two chapters, these are reflected by the basic types and functions of MPLL. In this chapter, an introductory example established the most important concepts, namely landmarks and qualitative representations of direction and distance. Especially the concept of landmarks was discussed in detail, as one of the key concepts, along with others, such as reference systems and fuzziness. Suitable basic data types and basic spatial relations have shown the fundamental motivation for the design of MPLL. The next chapter contains an overview of the system architecture and describes a number of modules which provide functionality around its central component MPLL.

3. System Architecture

This chapter provides a comprehensive description of the underlying system architecture constituting the environment in which MPLL is meant to be used. An overview in the next section is followed by detailed descriptions of the different modules and their application.

3.1. Overview

Spatial notions are so diverse that it is impossible to hard-code even the most important ones in a knowledge representation system. An alternative, therefore, is to develop a spatial specification language. MPLL is designed for this purpose. It facilitates the definition of application specific spatial notions in a symbolic way, based on a number of predefined basic types, algorithms and functions.

MPLL is a specification language for spatial notions with a concrete operational semantics. It is not, however, a general purpose programming language. The parser, compiler, and the abstract machine are not standalone systems. They must be embedded into a host system which provides the data structures and algorithms (e.g. for configurations, coordinate systems, graphs), and which serves as the interface to an application.

As shown in Fig. 3.1, MPLL is a central component in the overall system architecture, which serves several purposes. It provides a specification language for spatial notions, along with a number of basic functions and operations. Applications can connect to MPLL in order to make use of the language and processing services. In order to fulfil the processing tasks, MPLL employs internally implemented methods, as well as external services, which are linked to MPLL as independent modules (denoted by the square boxes above MPLL in Fig. 3.1). The TransRoute Service, for example, provides routing services to MPLL, but can also access MPLL for processing of spatial information as it can be accessed also directly by (web) applications. MPLL makes use of the routing service in the module, TransRoute can use the processing capabilities of MPLL in order to annotate route descriptions. Similarly, other modules provide a range of services to the

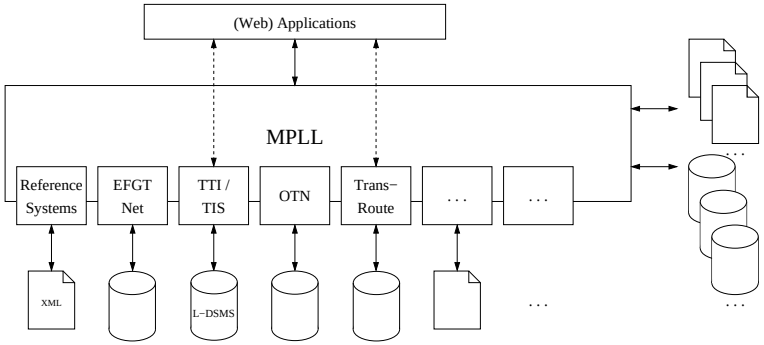


Figure 3.1.: Overview of the MPLL System Architecture

central component. A number of data sources (e.g. data bases, data streams, documents), provide information to the processing components.

Once connected to MPLL, task specific libraries can be loaded by an application. Certain spatial relations allow for a broad range of interpretation. To ensure that the appropriate interpretation is used (depending, for example, on the task, the context, environmental factors, or specific user preferences), different libraries can be accessed and functions can be (re)defined. First and foremost, the MPLL Standard Library (see section 4.6) should be loaded, since it contains a huge number of type constructors, function definitions, and constants. Additional libraries offer different functionality, or simply serve to redefine some functions. Furthermore, the data sources required for the task can be accessed and spatial entities can be introduced to start processing.

3.2. Modules

A broad and complex field such as geospatial information processing calls for modularisation of specific mechanisms and data structures. A monolithic approach would integrate too many heterogeneous elements and therefore interfere with a clean system architecture. Depending on the specific application, the combination of different modules, i.e. algorithms and data structures, must be possible.

Specific Requirements

The subtasks in one application, for example, could require route planning, as well as access to an ontology pertaining to the application's specific requirements. The respective modules should provide services for the different subtasks.

Another application might involve transformation of data of different coordinate systems. Some aspects could be depending on free space navigation (as opposed to network

routing) and spatial relations. Generally, a huge number of different combinations could make sense.

The specific requirements of these individual modules is so high, that there is no possibility to specify constructs and mechanisms within MPLL which provide the exact same service, at least not in an equally efficient manner. Examples for this issue are provided in the following descriptions.

This section describes some basic modules, which are most likely applicable to a wide range of tasks in spatial information processing.

3.2.1. Spatial Reference Module

This Module deals with the spatial reference systems and their inherent specificities. Much like a calendar system anchors temporal constructs in time, a spatial reference system provides a frame of reference for spatial entities in which spatial processing can be conducted.

Computer Screen

A simple reference system could be the graphics subsystem of a computer: a desktop, displayed on a screen. A desktop like this has a coordinate system with, say, about a million (1280 * 1024) basic elements called pixels, which are numbered from left to right and top to bottom. Managing elements on the screen (e.g. windows, buttons, mouse pointer) usually involves using screen coordinates, i.e. a tuple of horizontal and vertical coordinates, for example $(517, 753)$. Additionally, there exist some semantics which are specific to such a screen, or, to be more exact, specific to a desktop. One thing specific to a desktop and the mouse pointer is, for example, that it is not possible for the mouse pointer to leave the screen on the left hand side and reappear on the right hand side. There is no wraparound, neither horizontal, nor vertical. Of course, there are exceptions. Technically, facilitating such a wraparound would be a trivial issue. However, a need for this did obviously never arise. Also, nowadays, a system can, for example, include two screens which are combined to display a large desktop. The single desktop is split into two sections and, of course, in such a setup it is desirable (and possible) to move the mouse pointer off one screen, onto the other screen.

The Globe

Geospatial reference systems are organised in a similar way. Obviously, the earth has to be modelled as a sphere and a different coordinate system must be used. However, locations on a globe can also be identified using a tuple of coordinates. Due to the spherical shape, it is possible for an entity to move in one direction and never actually reach an “end” of the reference frame. A mouse pointer on the earth’s surface – for lack of a better connection to the previous example – could be moved indefinitely into any direction. There is no concrete boundary to the reference system, as long only movement on the sphere’s surface is modeled.

Highly specific elements, such as wraparound, are a substantial part of a reference system’s logics and are, therefore, usually hard coded within the respective module. From

MPLL, this module can be accessed via predefined keywords, which select the respective reference system to be used within a certain application.

3.2.2. Graph Routing Module

Graph data structures and graph processing build an essential foundation of MPLL, especially for the processing of distal relations using route planning. Incorporating both the necessary data structures and algorithms into the functional frame of MPLL would be cumbersome at best, and certainly not very elegant. By providing an interface to the TransRoute [151] system, which has been developed parallel to MPLL, the necessary functionality can be integrated into MPLL.

TransRoute offers standard graph modelling mechanisms and a number of different routing algorithms for modelling transport network based problems. A short introduction to TransRoute can be found in in this chapter in section 3.3.3, a more detailed description is available in section 6.3.3, pp. 184.

3.2.3. Traffic Information Module

This module facilitates access to the Local Data Stream Management System (L-DSMS) which delivers traffic information based on the Traffic Message Channel (TMC), broadcast via the Radio Data System (RDS). Further information about RDS and TMC can be found in section 6.4. A short introduction to L-DSMS can be found in in this chapter in section 3.3.1, a more detailed description is available in section 6.3.2, pp. 181.

Traffic information plays a significant role in the routing and navigation subtasks which are used by MPLL (or which make use of MPLL respectively).

3.2.4. OTN Module

This module provides an interface to the Ontology of Transportation Networks (OTN). OTN [78] provides a comprehensive modelling of transport networks in urban areas. The ontology includes typical classes, such as different kinds of paths, designated points of interest, and designated areas, but also public transport, such as busses, trams, or subways. The purpose of OTN is to provide a usable model of transport networks. Access to such a model is especially important for routing and navigation tasks, which are, in turn, used to provide a practical and highly flexible distance metric.

A short introduction to OTN can be found in in this chapter in section 3.3.2, a more detailed description is available in section 6.3.1, pp. 179.

3.2.5. Topological Reasoning Module

Unfortunately, this module is not available yet. It should be integrated to extend the reasoning capabilities of MPLL to include topological reasoning, since topological rea-

soning represents an important aspect of qualitative spatial reasoning. Due to the fact that extensive research has been done in this field, and is still going strong today, suitable implementations should be readily available.

3.3. Related Projects

During the course of this work several related projects have been realised under the author's supervision. These projects are more or less closely related to the work presented in this thesis as they can either function as modules which can be integrated into the MPLL system architecture (e.g. TransRoute) or they aid in processing and/or providing data to be processed with MPLL. This section gives a short overview of the individual projects for introductory purposes. A more in-depth description about each of these projects can be found in section 6.3, pp. 179.

3.3.1. Local Data Stream Management System

In order to have access to real time traffic information, a modular system for receiving, processing, and filtering data streams has been developed: the Local Data Stream Management System (L-DSMS) [105]. This system transforms traffic messages received via FM radio signals into an XML stream for the purpose of annotation[1] and subsequent filtering, as well as for the use in routing algorithms which need to take congestion information into account while producing travel routes within road networks.

Statistical Evaluation

In supplemental project work Michael Buschmann and Markus Krieser [20] focussed on persistent storage and statistical evaluation of RDS/TMC data. Storing TMC messages in a database system facilitates the statistical evaluation, and, subsequently, calculation of the likelihood of incidents on certain road segments at certain times. On weekdays, late afternoons, it is, for example, very likely that there is a traffic jam on the northbound highway A9 near Munich. Information like this is very valuable if the planning phase of routing and navigation is conducted well in advance of the execution phase.

Data Presentation

In another related project work [71], Christian Hänsel developed an interface for the TMC data provided by L-DSMS to be displayed in the popular Google Earth Client [66]. This work demonstrates the possibilities of integrating highly dynamic geospatial data with the static data provided by Google Earth, using the Keyhole Mark-Up Language (KML) [93].

Each of these projects is discussed in more detail in section 6.3.2, pp. 181.

[1] The binary data stream contains only codes about locations and events, which have to be transformed into (among other information) human readable real names of locations and descriptions of events.

3.3.2. Ontology of Transportation Networks

Another project involved the development of the Ontology of Transportation Networks (OTN). OTN [106] aims at providing yet another source of specially tailored data for routing applications. By providing the data in form of instances of an ontology, the data can be handled in a more intuitive way and the semantic information can be used by the routing applications in order to produce more suitable routes. Closely related work in progress concerns a comprehensive user and context model which is needed for the routing applications in order to produce personalised and optimised results.

As a means of producing maps that can easily be displayed in Web browsers and to facilitate interactive features, an ontology based system for map generation has been developed [104] in close relation to OTN. The system offers not only static map display, but also means of displaying dynamic map elements, such as weather or traffic information, or information about public transport.

For a more detailed description of OTN see section 6.3.1, pp. 179.

3.3.3. TransRoute

TransRoute is an object-oriented framework for routing applications which is used to model various real-world transport networks (e.g. street networks, public transport, buildings) as hierarchical graph structures. Its basic functionality is to compute shortest paths, conduct nearest neighbour searches, and provide other, primarily graph-related, services.

Hierarchical graph structures are a key element in providing a data model which allows for true multimodal planning. TransRoute provides the mechanisms required for basically any human locomotion related processing, from single buildings and complexes consisting of multiple buildings to regional and international networks of trains or airlines.

Section 6.3.3, pp. 184, contains a more comprehensive discussion of the basic features of TransRoute.

3.3.4. Indoor Positioning and Navigation

In a collaborative effort, Andreas Heindel [72] and Thomas Rickinger [139] each developed part of a prototypical system for indoor positioning and navigation.

Fingerprinting

Rickinger worked on an indoor positioning system using client based Wireless Local Area Network (WLAN)[2] fingerprinting. Such a system determines a position in space by comparing the radio signals received from a number of WLAN access points with a previously recorded signal pattern of a number of positions within a building. This technique does not operate continuously, but in a discrete manner, so that the current

[2] IEEE 802.11(a/b/g), the Wi-Fi standard, denotes a set of Wireless LAN/WLAN standards developed by working group 11 of the Institute of Electrical and Electronics Engineers (IEEE) LAN/MAN Standards Committee (IEEE 802).

position is always one of several predefined positions. This, however, can also be an advantage, since indoor scenarios can operate much better on symbolic positions (i.e. *"in the entrance hall"* or *"in room Z 1.03"*) than on numeric coordinates. Providing an interface using the NMEA protocol, which is widely used by GPS equipment, the system can be seamlessly integrated with outdoor positioning systems.

Indoor Navigation

The system developed by Heindel (based on previous work by Doreen Mizzi [111]) uses Rickinger's positioning system for indoor navigation. Once the starting position has been determined and the destination has been specified, a path planner provides a path through the network of corridors, rooms, stairways, lifts, etc. The output of the route is twofold. While Mizzi provided a natural language text output, Heindel worked on an additional graphical display of the navigation instructions. Combining these two techniques, a very detailed and unambiguous route description can be achieved. Furthermore, Heindel extended the system to operate on several floors. The prototype developed by Mizzi was limited to a single floor.

3.3.5. PlanML

A generic language for path or plan descriptions is the focus of the work of Matthias Schmeisser [145]. The language *PlanML*, a markup language for expressing planning results, is currently under development.

Generally, a plan consists of a number of actions, which have to be performed in a certain sequence at certain locations. PlanML aims for providing a generic means for expressing such planning results. One main feature of PlanML is a hierarchical structure, which facilitates flexible and transparent handling of very detailed descriptions. This means, a plan can provide a very detailed description, but this fact does not have to be revealed if the user does not request it. A typical plan might lead the user through familiar as well as unfamiliar territory. Albeit the complete plan is available in a certain (high) level of detail, the user might want to skip the details in familiar surroundings (e.g. the way from his/her home to the airport – which has been travelled many times), but rely on, and request, higher levels of detail in unfamiliar ones (e.g. the way from a foreign airport to a hotel in a city he/she has never been to).

3.4. Summary

MPLL aims for providing several services in order to facilitate spatial information processing for a number of applications and scenarios. Basic features, i.e. types, functions and constants, are included in MPLL. The MPLL Standard Library contains many composite functions defined in MPLL syntax, for example overloaded versions of existing functions and constructors, as well as predefined constants. Additional functionality is separated into different modules, which can be accessed locally or as external (web) ser-

vices. Depending on the specific task, only the required functionality and modules are accessed. Error management must be handled by the host application.

This architecture facilitates the use of different processing and reasoning techniques. Because MPLL offers means to specify qualitative as well as quantitative data, the language serves to bridge the gap between the two heterogeneous concepts. Subsequently, the concepts and methods best suited to handle a specific task are made available using standardised language constructs and interfaces.

If some aspect of spatial information processing is not covered by the system – which is very likely – the modular architecture facilitates easy extension, either in form of MPLL constructs, or in form of additional modules providing specific services. Extending and/or modifying the hard coded parts of the implementation of MPLL is also possible.

4. MPLL – Multi-Paradigm Location Language

A spatial specification language should facilitate the definition of application specific spatial notions in a symbolic way. It should also allow to compile these specifications into executable code. Furthermore, it needs to be expressive enough to define spatial notions in an easy and intuitive way, and it needs to have the relevant data structures and algorithms built in.

The Kernel

The language MPLL uses the time-independent parts of the GeTS [124] as a kernel. MPLL extends the kernel of GeTS with location specific concepts. The built-in data structures of GeTS are various numeric types, one-dimensional fuzzy intervals over real numbers, labelled partitionings for modelling periodic temporal notions, calendar systems and durations. The numeric types and the one-dimensional fuzzy intervals are also relevant for MPLL. All other data types are not needed. MPLL is a typed functional language with the usual control constructs, local variable bindings, but also assignments and a few other imperative constructs.

The following section deals with the transition from GeTS to MPLL and the inherent commonalities and differences. Then, the language is introduced and design decisions are discussed, followed by an in-depth look at the generic language constructs. The subsequent four sections (sections 4.4 through 4.7) contain the complete MPLL specification, including basic types and functions, as well as the MPLL standard library. A short summary concludes this chapter.

4.1. From GeTS to MPLL

From a modelling perspective, the domains of time and space show a number of analogies as well as differences. These have to be taken into account since this work presents

Time and Space

the spatial specification language MPLL which in its foundation is based on parts of the temporal specification language GeTS. The implementation frame has basically been stripped from any structures which are only relevant for the temporal domain. However, some structures, for example fuzzy intervals and some basic types, are relevant for both domains and therefore remain.

Building MPLL on the fundamentals of GeTS shows some distinct advantages:

- Reusing generic components of GeTS saves time by reducing redundant work which is not domain specific.
- Generic components are not duplicated, resulting in less code to be maintained and a decreased possibility for errors.
- Common subsets of the two languages and identical development environments help in a possible merge of the two languages at a later time to possibly produce a spatio-temporal language.
- A common and compatible syntax facilitates easy use of both systems.

4.1.1. Granularities

Time Intervals

In GeTS [124], the modelling of time is partly based on the assumption that the system should not primarily handle time *points*, but time *intervals* of different granularity. Therefore, time is discretised in different types of granules, for example seconds, days, weeks or months. The granularities can be flexibly defined (e.g. ”my work week”) and can be used in various relations (e.g. ”the first holiday during my work week after Easter”). The main reason for not using time points explicitly is the fact that humans tend to define points in time using a granule that is sufficiently exact for the specific purpose (days, minutes, seconds). An inifinitely small granule is generally of very limited use, i.e. there exists no such thing as a single point in time which has no extent.

Points in Space

Being as continuous in nature as time, space can also be discretised quantitatively, for example by means of a coordinate system. In technical terms, it is very easy to specify space by quantitative means like coordinates. Like with time, humans do not really use *points* in space which have no extent. They tend to use objects which have a certain extent (regions, volumes). However, point entities, i.e. infinitely small points in space which mark a specifiifed position, are used to determine the position of said objects. This is usually done by specifying an object-specific reference point and its position in space [132, 112, 108]. All these entities are then represented by points, lines, polygons, and/or volumes. Therefore, the two languages will show some similarities regarding their basic constructs.

4.1.2. Basic Types

GeTS Basic Types

Two groups of basic types are specified in GeTS: *data structure types* and *enumeration types*. The former represent built-in data structures, the latter are used to operate specific functionality of some algorithms.

Table 4.2 on page 111 shows the predefined enumeration types which are provided by MPLL. These types are used to operate some of the algorithms because their individual meaning depends on the meaning of the built-in function where they occur as parameters.

Commonalities

Basic types in MPLL include some of the types provided by GeTS. Obviously for example integers, floats, and strings can be used for a variety of purposes and are not bound to any application domain – spatial, temporal or any other. Apart from these types there are some which are only time-specific (e.g. `Partitioning` or `DateFormat`). The type (fuzzy) `Interval` is somewhat significant because of its high specificity on one hand and its relevance for both domains on the other. In addition to their application in the temporal domain, fuzzy intervals can be used, for example, in connection with distal information (sometimes even expressed by temporal means, i.e. travel times, but also in a purely metrical sense) or with angular expressions such as “in front of”.

MPLL Basic Types

The spatial domain requires a number of distinct basic types. The classic basic entities for spatial abstraction, which are also used in MPLL, are points, lines and polygons.

Spatial abstraction begins with positions of entities, for example positions of the user, points of interest, landmarks and many other objects which require to be put in spatial relation. Points in space are the classic measure to mark these positions. The position (and orientation) of more complex objects can be achieved by specifying a reference point and an orientation.

Orientation, usually a single angular value, is what completes a point to become a configuration. Configuration space is three- or four-dimensional space, in which one dimension is specified by an object’s orientation. This is a common model, for example, for robot navigation [96] with (non-) holonomic robots and vehicles [103].

Linear features have a number of real life counterparts: streets, railway lines, borders, rivers, etc. Linear abstractions are built using lists of points, possibly supported by interpolation points to more closely model the shape of linear features in contrast to plain topology.

Finally, regions are an essential form of spatial features, also having many real life counterparts ranging from small scale (rooms, buildings, cadastral data, etc.) to large scale (districts, regions, states, countries, and so forth) including many things in-between. Regions are represented by polygons which are modelled similar to lines. In addition, polygons have to be normalised, i.e. always define a region in counter-clockwise manner (without intersections), and have to be closed.

A more detailed discussion of these types can be found in the following section.

4.1.3. Geospatial Primitives in MPLL

The most basic way humans reason about space is two-dimensional [36]. Maps, by their very existence, are two-dimensional representations of space and have been used effectively for centuries. In the cases where two dimensions were not sufficient, two-dimensional maps could be annotated using different techniques to incorporate a third dimension, although this was only necessary in special cases. For most cases, annotation was a sufficient means to model a third dimension (e.g. depth of the water, elevation of the ground). The idea of 2.5-dimensional representations is mostly found in the domain of architecture. Some applications do not require full 3D modelling, but can operate on several overlapping layers of 2D data, such as floor plans. The discretisation of the third dimension into separate floors provides the underlying cognitive model.

Discrete Model of Space

Each of the three axes of space corresponds to an axis isomorphic to the real numbers $\mathbb{R}$, therefore we define space as $\mathbb{R}^3$. Although space is continuous, coordinate systems measure space in discrete units, for example in degrees, minutes, and seconds, or as (finite) decimal fractions. Scale notwithstanding, a discrete model of space is sufficient for most applications of spatial information, especially in the geospatial domain. In this context we mean by "geospatial" that we handle the same information as humans do in their everyday dealings, which is anchored in a geographical frame of reference. This domain is neither extremely precise, due to the qualitative nature of human spatial reasoning, nor do we operate on an extremely broad scale (we deal with kilometres, metres or millimetres, not nanometres or parsecs). In any case, we deal with discrete models of space which can easily be represented as generic numerical values. The type of the respective coordinate system is not of great relevance for the underlying structures, since most coordinate systems can be translated into others by simple transformation. Due to the geospatial frame, we default to WGS84 coordinates, if not explicitely stated otherwise.

Angles

Angles are used primarily to model two important factors of spatial cognition: orientation and bearing. As laid out in section 2.4.2, angular expressions can be used for various purposes and play asubstantial role in human spatial cognition. The technical realisation of angular expressions is described in Def. 4.19.

Points

Points are locations in space which have no spatial extent. Although in the real world there exist no objects without spatial extent, the use of points in space is an accepted abstraction for the position of reference points which are in turn associated with a certain object. The user of a mobile GPS receiver will for example always obtain the discrete position of the device (i.e. of him-/herself) as a single point in space, which is the very purchase of application.

Although points are primarily two-dimensional, the point data structure in MPLL can store a vertical component as the z-axis. For the rare occasions where three-dimensional information is needed, this component can be used. In general, it can be neglected.

Configurations

Configurations combine a position in space, i.e. a point entity, and an orientation. These two properties are usually sufficient to model mobile entities moving around in a model of the real world. Optionally, further information can be included, such as shape and other properties.

Lines and Polylines

Lines and polylines are structures comprised of one or more linear segments, which are represented by pairs of points in space. These segments are always straight lines with no possibility of interpolation or support for rounded instances. A line is therefore a single straight connection between two points in space, and a polyline with n segments is an object consisting of n straight (and not necessarily co-directional) lines, whereas it consists of $n+1$ coordinate points – one initial pair of coordinates and one additional point for each segment after the first one. If a curved structure is to be represented, the number of segments can be increased to simulate more complex forms. However, this approximation is sufficient most of the time.

Polygons

Polygons are very similar to polylines. The only difference is that they are not open, i.e. the last and first point is identical, no two line segments do intersect, and they are always normalised in the sense of computational geometry [110, 92, 14, 39].

Circular Intervals

Circular Intervals are a special form of intervals, which only cover a certain closed interval. An example for such an interval is an orientation-dependent fuzzy value, which is defined in a specific interval, e.g. $[0, 2\pi[$. Cardinal directions are preferably modelled using circular intervals in order to be able to represent qualitative notions such as "approximately north".

4.1.4. Reference Systems

Calendars

A reference system in the temporal domain is usually called *calendar*. In the CTTN system [121], of which GeTS is a part, a calendar is a set of partitionings, such as minutes, seconds, months and years. These partitionings, some extra data and algorithms, and calendric calculations [42] are used to model a calendar, including some "inconvenient" features, such as leap seconds or daylight saving time schemes. Due to the complexity, calendars and their specification are not included in GeTS, but instead as one of several modules within the CTTN system. These issues have been discussed in more detail by Hans Jürgen Ohlbach [116, 117, 120, 121, 125] and Hans Jürgen Ohlbach and Dov Gabbay [127] respectively.

Coordinate Systems

Spatial reference systems are also commonly referred to as *coordinate systems* since for example the position of a point P in Euclidean space $\mathbb{R}^n$ is given using an n-tuple $P = (r_1, \ldots, r_n)$ of real numbers, the *coordinates* of P. In Euclidean geometry, these coordinates are also called *cartesian coordinates*, in respect to the French mathematician and philosopher René Descartes, who, among other things, worked to merge algebra and Euclidean geometry.

A Cartesian coordinate system is used to uniquely determine each point within a plane by two numbers, called the x-coordinate and the y-coordinate of the point. To define the coordinates, two perpendicular directed lines (the x-axis or abscissa and the y-axis or ordinate), are specified, as well as the unit length, which is marked off on the two axes. Cartesian coordinate systems are also used in space (where three coordinates are used) and in higher dimensions.

In the scope of this work, there are several factors pertaining to the use of coordinate systems which have to be handled within the module providing the reference systems logics (as opposed to handling these within the language MPLL). An introductory description is given in the following paragraphs, technical details can be found in section 4.4.8.

The phenomenon of *wraparound* exists in several variants and perfectly serves to illustrate the complexity of reference system features. A planar projection of the globe in its common form shows two very different kinds of wraparound which have to be handled accordingly.

Horizontal Wraparound

If an aeroplane were to move parallel to the equator, it would sooner or later reach "the end" of the map and "reappear" on the other side, since the vertical edges of the map designate identical locations in reality. There exists a *horizontal wraparound* in the direction of the x-axis of the map, which is only influenced by its arbitrarily defined anchor on the x-axis. Therefore, the map can be panned along the x-axis to horizontally centre it on some specific part of the globe. This does not resolve the wraparound, although it shifts its position.

Vertical Wraparound

Likewise, there exists a *vertical wraparound*, which has completely different characteristics. In fact, the horizontal edges of the planar map represent not linear structures, but two *points* on the globe, which are not identical, as in the case of horizontal wraparound: these are the north and south poles of the globe. Thus, an aeroplane moving along perpendicularly to the equator will, at some point along the journey, end up at one of the poles and *will not* reappear on the opposite edge of the map, but on the same edge at a different longitude.

These forms of wraparound are just two examples pertaining to geospatial scenarios. Other scenarios might feature other types of wraparound, or no wraparound at all (for example desktops, i.e. single screen computer systems).

Projection and Geometry

Should the aforementioned aeroplane move over the globe at an angle not perpendicular or parallel to the equator, it would not be able to ever reach either pole. It might not even come close to the polar regions due to the geometry of the globe, i.e. its spherical shape. A straight journey – from the point of view of the aircraft – would show up on the planar projection as a curved line, much like the ones commonly seen on the charts of airlines representing their networks, connections, and airports they service.

4.2. The Language MPLL

The design of the language MPLL is largely based on its temporal counterpart GeTS. The following considerations pertaining to MPLL adhere closely to those of Hans Jürgen Ohlbach regarding GeTS [124]. There are several publications [118, 119, 122] available, which give a good general overview of his work on temporal reasoning.

1. The language MPLL is not intended as a general purpose programming language, although it has many features of a functional programming language. However, it is a specification language for spatial notions with a concrete operational semantics.

2. The parser, compiler, and, in particular, the underlying MPLL abstract machine are not standalone systems. They must be embedded into a host system which provides the data structures and algorithms for configurations, coordinate systems, graphs, etc., and which serves as the interface to the application. MPLL provides a corresponding application programming interface (API).

3. The language should be simple, intuitive, and easy to use. It should not be cluttered with too many features which are mainly necessary for general purpose programming languages.

4. The two previous properties are primary reasons against a solution where MPLL is only a particular module in a functional language like SML or Haskell. The host system was developed in C++. Linking a C++ host system to an SML or Haskell interpreter for MPLL would be more complicated than developing MPLL in C++ directly. The drawback is that features like sophisticated type inferencing or general purpose data structures, such as lists or vectors, are not available in the current version of MPLL. If it turns out that they are useful for some applications, however, they can still be integrated into MPLL at a later time.

5. Developing MPLL from scratch instead of using an existing functional language has another advantage. The design of the syntax of the language can be done in a way which better reflects the semantics of the language constructs. This makes it easier to understand and use. As an example, the syntax for a configuration constructor is just `Configuration`$(expression_1, \ldots, expression_n)$. The freedom in designing a syntax is, however, limited by the available parser technology, in this case, flex [162] and bison [55]. Therefore, regarding some of the language features, a compromise between intuitiveness and technical constraints had to be accepted.

4.2.1. Examples

MPLL is a strongly typed functional language with a few imperative constructs. The following examples should give a general idea of its structure, which is laid out in detail in the subsequent sections.

Example 4.1 (Configuration)
The definition

$$\texttt{Configuration}(\mathit{ADir}, \mathit{Ax}, \mathit{Ay}, \texttt{true})$$

specifiex a configuration as follows: a new configuration of type `Configuration` *is constructed, with the standard properties. ADir is of type* `Angle` *and specifies the orientation of the spatial entity denoted by the configuration. Ax and Ay, both of type* `Angle`*, specify the position of the configuration in planar space.*

Example 4.2 (Filter)
The expression

```
Let C = Configuration(bearing(P3,P4),P4) in
  Let threshold = 0.8 in
    filter(lambda(Point P)(
      (lambda(Point Q)(
         maxDistance(C, Q, close)(Q)) &&
         (bearing(C, P, front, threshold) ||
          bearing(C, P, left, threshold)  ||
          bearing(C, P, back, threshold)  ||
          bearing(C, P, right, threshold))
       (P)), landmarks))
```

filters a list of landmarks (point entities) as follows:

The current reference position, i.e. the position of the user, is set to a particular position and orientation in planar space (line 1). This particular position is the current location on a route, which has been given by a routing application. In order to find suitable landmarks for the annotation of the route (i.e. generating a suitable route description), the surrounding landmarks are filtered by two properties. First, landmarks which are not located "close" to the user are discarded (line 5). Then, landmarks which are not located in one of the given cardinal directions ("front", "left", "back", "right") are also discarded (lines 6–9). The threshold value defined in line 2 pertains to the minimum fuzzy equality of the bearings. This function returns a list of landmarks which fulfil the given requirements.

Example 4.3 (Reference System)
The definition

$$\texttt{ReferenceSystem(geospherical,}-\pi,\frac{\pi}{4},\pi,-\frac{\pi}{4},\texttt{true,true)}$$

specifies a reference system as follows: A new reference system of type `geospherical` *is constructed, which has upper left coordinates* $(-\pi,\frac{\pi}{4})$ *and lower right coordinates* $\pi,-\frac{\pi}{4}$ *and vertical and horizontal wraparound.*

This means in particular that no x coordinates outside of the interval $]-\pi,\pi]$*, and no y coordinates outside of the interval* $[-\frac{\pi}{4},\frac{\pi}{4}]$ *are allowed. Furthermore the internal (hard coded) mechanisms of the reference system type* `geospherical` *have to handle horizontal and vertical wraparound.*

The orientation of the reference system defaults to the direction of the positive y axis.

4.2.2. Variable Naming Conventions

The default naming conventions for variables are as follows:

Bools: Denoted by the single capital letter *B* or a variable name beginning with a capital *B*. If a function features more than one `Bool`, the subsequent capital letters *C*, *D*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Integers: Denoted by the single capital letter *N* or a variable name beginning with a capital *N*. If a function features more than one `Integer`, the subsequent capital letters *O*, *P*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Floats: Denoted by the single capital letter *F* or a variable name beginning with a capital *F*. If a function features more than one `Float`, the subsequent capital letters *G*, *H*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Angles: Denoted by the single capital letter *A* or a variable name beginning with a capital *A*. If a function features more than one `Angle`, the subsequent capital letters *B*, *C*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Points: Denoted by the single capital letter *P* or a variable name beginning with a capital *P*. If a function features more than one `Point`, the subsequent capital letters *Q*, *R*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Configurations: Denoted by the single capital letter *C* or a variable name beginning with a capital *C*. If a function features more than one `Configuration`, the

subsequent capital letters *D*, *E*, etc. can be used in addition – provided there is no conflict with other types' variable names.

In deictic settings, the order of variables is always (1) the user's position (*C*), (2) the referent (*D*), and (3) the relatum (*E*). In all other settings, the function of the relatum is fulfilled by (1), there exists no variable (3) and and (2) remains unchanged.

Lines: Denoted by the single capital letter *L* or a variable name beginning with a capital *L*. If a function features more than one `Line`, the subsequent capital letters *M*, *N*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Polygons: Denoted by the single capital letter *R* (region) or a variable name beginning with a capital *R*. If a function features more than one `Polygon`, the subsequent capital letters *S*, *T*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Lists: Denoted by the single capital letter *L* or a variable name beginning with a capital *L*. If a function features more than one `List`, the subsequent capital letters *M*, *N*, etc. can be used in addition – provided there is no conflict with other types' variable names. Note that a possible conflict with lines' variable names must be avoided.

Intervals: Denoted by the single capital letter *I* or a variable name beginning with a capital *I*. If a function features more than one `Interval`, the subsequent capital letters *J*, *K*, etc. can be used in addition – provided there is no conflict with other types' variable names.

Direction: (Enumeration type) – Denoted by the variable name *Dir*. If a function features more than one direcional expression, then variable names can be varied but must feature a heading or trailing "*Dir*".

4.3. Language Constructs

MPLL has a number of general purpose functional and imperative language components. Additionally, a number of language constructs are geared to manipulating points, lines, polygons, etc. As mentioned above, the language is strongly typed, i.e. the type of each expression is determined by the top level function name together with the types of its arguments.

MPLL tries to minimise the required number of parentheses in the expressions. Nevertheless, it is usually clearer and easier to understand when additional parentheses are used. The language has an operational semantics. It is described more or less formally when the language constructs are introduced.

Some aspects of the language depend on the context where it is used. For example, MPLL itself has no exception handling mechanisms. However, exceptions are thrown and have to be caught by the host programming system.

Definition 4.1 (Function Definitions)
An MPLL function definition has one of the forms

(1)	*name*	=	*expression*
(2)	*name*()	=	*expression*
(3)	$name(type_1\ var_1,\ldots,type_n\ var_n)$	=	*expression*
(4)	$type : name(type_1\ var_1,\ldots,type_n\ var_n)$	=	*expression*
(5)	$type : name(type_1\ var_1,\ldots,type_n\ var_n)$		

The five versions of function definitions can have a trailer: '`explanation`*: any string'. The explanation is attached to the newly defined function. It can be accessed by the host system.* ■

Version (1) and (2) are for constant expressions, i.e. the name on the left hand side is essentially an abbreviation for the expression on the right hand side. Version (3) is the standard function definition. The type of the function is $type_1 * \ldots * type_n \mapsto T$ where T is the type of the *expression*. Version (4) declares the range type of the function explicitly. It can be used for recursive function definitions, where the name of the newly defined function occurs already in the body. In this case, it is necessary to know the range type of the function before the *expression* can be fully parsed. The factorial function, for example, must be defined in this way:

$$\texttt{Integer: fac(Integer } n) = \texttt{if } (n == 0) \texttt{ then } 1 \texttt{ else } n * \texttt{fac}(n-1) \quad (4.1)$$

Finally, version (5) is a forward declaration. It must be used for mutually recursive functions.

Remark 4.3.1 (Overloading)
Function definitions can be overloaded . They are distinguished by their argument types*, not by the* result type*. This means, two function definitions*

```
f(Integer n) = ... and
f(Float m) = ...
```

yield different functions, whereas the second definition in

```
Integer:f(Integer n) = ... and
Float:f(Integer n) = ...
```

overwrites the first one, or is rejected. This depends on the global control parameter `MPLL::overwrite`*.* ■

Definition 4.2 (Literals)
Literals are strings which can be interpreted as constants of a certain type. See Remark 4.4.1 on page 109 for the string representation of literals. ■

4.3.1. Arithmetic Expressions

MPLL supports the same kind of arithmetic expressions as many other programming languages.

Definition 4.3 (Binary Arithmetic Expressions)
Let N be a number type (i.e. $N =$ `Integer` or $N =$ `Float`).
If n and m are valid arithmetic expressions, then the following binary operations are allowed:

$n+m$	*addition*
$n-m$	*subtraction*
$n*m$	*multiplication*
n/m	*division*
n % m	*modulo*
$max(n,m)$	*maximum*
$min(n,m)$	*minimum*
$pow(n,e)$	*exponentiation* (n^e)

The types are determined according to the following rules:
for the operators '+', '-', '', '/', max and min:*

```
Integer  *  Integer  ↦  Integer
Float    *  Integer  ↦  Float
Integer  *  Float    ↦  Float
Float    *  Float    ↦  Float
```

`Float` *values are not allowed for the modulo operator %. Therefore the remaining type patterns for % are:*

```
Integer  *  Integer  ↦  Integer
```

The exponentiation operator $pow(a,n)$ is only allowed for `Integer` *exponents n and for* `Float` *or* `Integer` *bases a.*

```
Integer  *  Integer  ↦  Integer
Float    *  Integer  ↦  Float.
```

■

Flat expressions like $a+b+c+d$ without parentheses are allowed. The operator precedence is $-$, $+$, $/$, $*$, i.e. $*$ binds most. The functions *min* and *max* accept more than one argument.

Definition 4.4 (Unary Arithmetic Expressions)
There are four unary arithmetic operators in MPLL:

$-n$	$[N \mapsto N]$	*N is any number type*
`float`(b)	[`Bool` $\mapsto$ `Float`]	
`round`(a)	[`Float` $\mapsto$ `Integer`]	
`round`$(a,$`up/down`$)$	[`Float` $*$ `UpDown` $\mapsto$ `Integer`]	

■

For a definition of enumeration types, e.g. `UpDown`, see section 4.17 or table 4.2 respectively.

$-n$ negates the number n.
n can be an expression of type $N = \texttt{Integer}$ or $N = \texttt{Float}$.

$\texttt{float}(b)$ turns a boolean value b into a floating point number:
$\texttt{float}(\texttt{false}) = 0.0$ and $\texttt{float}(\texttt{true}) = 1.0$.

$\texttt{round}(a)$ rounds a `Float` value a to the nearest integer. 1.5 is rounded to 1, 1.51 is rounded to 2. -1.5 is rounded to -1, -1.51 is rounded to -2.
$\texttt{round}(a, \texttt{up})$ rounds the `Float` value a up, and
$\texttt{round}(a, \texttt{down})$ rounds the `Float` value a down.

Definition 4.5 (Trigonometry)
Let N be a number type (i.e. $N = \texttt{Integer}$ or $N = \texttt{Float}$).
If n and m are valid arithmetic expressions and $0 \leq m \leq 1$, then the following trigonometric operations are allowed:

operator		*argument*	*result*
$sin(n)$	*sine*	*radian*	$[-1,1]$
$cos(n)$	*cosine*	*radian*	$[-1,1]$
$asin(m)$	*inverse sine*	$[-1,1]$	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
$acos(m)$	*inverse cosine*	$[-1,1]$	$[0,\pi]$
$sind(n)$	*sine*	*degree*	$[-1,1]$
$cosd(n)$	*cosine*	*degree*	$[-1,1]$
$asind(m)$	*inverse sine*	$[-1,1]$	$[-180°, 180°]$
$acosd(m)$	*inverse cosine*	$[-1,1]$	$[0°, 360°]$

The types for these operators are determined according to the following rules (with the restriction of the ranges above):

$\texttt{Integer} \mapsto \texttt{Float}$
$\texttt{Float} \mapsto \texttt{Float}$

■

Definition 4.6 (Arithmetic Comparisons)
If n and m are arithmetic expressions of type `Integer` *or* `Float` *then*
$(n \quad < \quad m)$
$(n \quad <= \quad m)$
$(n \quad > \quad m)$
$(n \quad >= \quad m)$
are the usual arithmetic comparison operators. The result is one of the boolean values `true` *or* `false`*. These operators compare different types, i.e.* $(3.9 <= 4)$ *yields* `true`*, as expected.* ■

The equality and inequality predicates compare numbers in the expected way, but also every other data type.

Definition 4.7 (Equality and Inequality)
If n is an expression of type T and m is an expression of type Q then

n `==` m *and* n `!=` m

are expressions of type `Bool`.

$n == m$ *yields* `true` *iff*

1. *T and Q are one of the number types* `Integer` *or* `Float`*, and the numbers are equal, i.e.* $4.0 == 4$ *yields* `true`.
2. $T = Q$*, both are enumeration types, and n and m are the same strings. This means in particular: if* $T =$ `Hull`*,* $Q =$ `Region`*,* $n =$ `core` *and* $m =$ `core` *then* $n == m$ *yields* `false` *(because* $T \neq Q$*).*
3. $T = Q =$ `Interval` *and n and m are the same intervals (i.e. the same polygons).*
4. $T = Q =$ `Partitioning` *and n and m are pointer-equal partitionings*
5. $T = Q =$ `Duration` *and n and m are the same durations.*

n `!=` m *yields* `true` *iff* n `==` m *yields* `false`. ■

4.3.2. Boolean Expressions

MPLL has the standard Boolean connectives: negation (-), and ('and' or '&&'), or ('or' or '||') and exclusive or ('xor' or '^').

Definition 4.8 (Boolean Expressions)
If a and b are Boolean expressions then

	$-a$		$[\texttt{Bool} \mapsto \texttt{Bool}]$
a	`and`	b	$[\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}]$
a	`or`	b	$[\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}]$
a	`xor`	b	$[\texttt{Bool} * \texttt{Bool} \mapsto \texttt{Bool}]$

are Boolean expressions with the corresponding meaning. ■

Flat Boolean expressions without parentheses are also allowed. The operator precedence is `xor`, `or`, `and`, i.e. `and` binds most.

4.3.3. Control Constructs

MPLL features the common 'if-then-else' construct. In addition, there is a `case` construct to avoid the need for a nested application of if-then-else. A 'while' loop is also available. Since MPLL is a functional language, the `while` construct needs a return value. Therefore, in addition to the `while` loop body, it has a separate return expression. In the body, however, only imperative constructs (with return type `Void`) are allowed.

Definition 4.9 (`if-then-else`)
If c is an expression of type `Bool` *and a and b are expressions of the same type T then*

$$\texttt{if}\ c\ \texttt{then}\ a\ \texttt{else}\ b$$

is an expression of type T.

Therefore, the type of the `if` *construct is in general* $\texttt{Bool} * T * T \mapsto T$.

There is one exception: If a is of type `Float`*, and b of type* `Integer`*, or vice versa, then the integer is cast to* `Float`*. The type of* `if` *is in this case:*
$\texttt{Bool} * \texttt{Float} * \texttt{Integer} \mapsto \texttt{Float}$ *or* $\texttt{Bool} * \texttt{Integer} * \texttt{Float} \mapsto \texttt{Float}$.
Example: '`if true then 3 else 4.0`*' yields 3.0 as a* `Float` *number.* ■

The definition of the factorial function (4.1) is a typical example for the use of if-then-else.

Definition 4.10 (`case`)
If $C_1,\ldots,C_n$ are Boolean expressions and $E_1,\ldots E_n$ and D are expressions of the same type T then

$$\texttt{case}\ C_1 : E_1, ..., C_n : E_n\ \texttt{else}\ D$$

is an expression of type T. ■

The operational semantics of this `case` construct is: the conditions $C_1,\ldots,C_n$ are evaluated in this sequence. If C_i is the first condition which yields `true`, then E_i is evaluated and its result is returned as the result of `case`. If all C_i evaluate to `false`, then the result of D is returned.

Exceptions for the requirement that $E_1,\ldots E_n, D$ are expressions of the same type T are: if $T = \texttt{Float}$ then some of the $E_1,\ldots E_n$ and D may have type `Integer`. These integers are then automatically cast to `Float`.

Definition 4.11 (`while`)
Let C be an expression of type `Bool`*, $E_1,\ldots,E_n$ expressions of type* `Void` *and 'result' an expression of type T then*

$$\texttt{while}\ C\ \{E_1, ..., E_n\}\ \textit{result}$$

is an expression of type T. ■

The operational semantics of this `while` construct is: as long as the evaluation of C yields `true`, evaluate the expressions $E_1,\ldots,E_n$ in this sequence. As soon as C yields `false`, evaluate *result* and return this as value of `while`.
An iterative definition of the factorial function is a typical example where the `while` construct is used.

$$\texttt{factorial}(\texttt{Integer}\ n) \quad = \quad \texttt{Let}\ f = 1 \quad \texttt{in} \quad \texttt{while}(n > 0)\{f := f * n, n := n - 1\} f$$

This example also illustrates the binding construct `Let` and the assignment operation.

Definition 4.12 (`Let`)
The construct
`Let` *variable* = *expression*1 `in` *expression*2
of type T
*evaluates the expression*1*, binds the result to the variable and then evaluates expression*2 *under this binding.*
*T is the type of expression*2. ■

Definition 4.13 (Assignment)
If x is a variable of type T and E is an expression of type T, then $x := E$ *is an expression of type* `Void`. ■

This is the usual assignment operation: the result of the evaluation of E is assigned to x.

Exceptions for the requirement that x and E have the same type are: if x has type `Float` then E may have type `Integer`. The value is automatically cast to `Float`.
Note that the assignment operation returns no value. It can only occur in the body of the `while` statement.

4.3.4. Functional Arguments

A *function call* in MPLL is an expression of the form $name(argument_1, \ldots, argument_n)$ where '*name*' is either the name of a built-in function, or the name of a previously defined function (or a function with forward declaration), or a variable with suitable functional type.

Since variables can have functional types, and MPLL allows overloading of function definitions, it needs a notation for functional arguments. A functional argument can either be just a variable with appropriate functional type, or a function name with argument type specifications, or a lambda expression. A function name with argument type specifications is necessary to choose among different overloaded functions.

Definition 4.14 (Functional Arguments)
A functional argument *in MPLL is either*

1. *a variable with the appropriate functional type,*
2. *an expression* $name[type_1 * \ldots * type_n]$ *of a previously defined function with that name and with argument types* $type_1 * \ldots * type_n$*, or*
3. *a lambda expression:*
   $lambda(type_1\ variable_1, \ldots, type_n\ variable_n)$ *expression.*
   If T is the type of 'expression' then $type_1 * \ldots * type_n \mapsto T$ *is the type of the lambda-expression.*
   'expression' can contain variables which are lexically bound outside the parameter list of lambda. ■

4.3.5. Compound Types

Definition 4.15 (Compound Type)
A compound type in MPLL is an expression $T_1 * \ldots * T_n \mapsto T$ *where* T *and the* T_i *are either basic types or compound types.*

A type expression *is either a basic type or a compound type expression.* ■

4.4. Basic Types

MPLL includes a number of generic basic types, which can also be found in other languages. These generic types are extended by basic spatial types, which are specially tailored for the spatial purpose of MPLL. All basic types can be combined to functional types $T_1 * \ldots * T_n \mapsto T$. They are represented by certain data structures and keywords.

4.4.1. Basic Spatial Types

There are two groups of basic types, the data structure types and the enumeration types. The data structure types represent built-in data structures.

Definition 4.16 (Data Structure Types)
A list of data structure types is given in table 4.1.

type	*description*
`Integer`	standard integers
`Float`	standard floating point numbers
`String`	strings
`Angle`	floating point numbers representing angles
`Point`	points (two-dimensional)
`Configuration`	configurations in space (`Point`, `Angle`)
`Line`	lines and polylines
`Polygon`	polygons
`List`	lists
`Interval`	fuzzy intervals
`CircularInterval`	fuzzy intervals
`ReferenceSystem`	reference system
`Route`	route (from graph/network routing)

Table 4.1.: Definition of Data Structure Types

Integers and Floats *– The data structure types abstract away from the concrete implementation. The* `Integer` *type, for example, corresponds to a 32 bit 'signed integer' data, the* `Float` *type corresponds to a 32 bit 'float' data type.*

Strings *are sequences of 8-bit characters*[1].

Angles *(Def. 4.19) are internally stored as a* `Float` *containing a grad value. Additionally, the* `Angle` *data type holds some properties which define the range of possible angular values.*

Points *(Def. 4.21) consist of integer coordinates* (x,y)*. The coordinates x and y are* `Integer` *values which can be adapted to the respective coordinate system by a multiplier to accomodate for decimal fractions. For example for the WGS84 coordinate system this means that all coordinates are multiplied by the factor* 10^6 *in order to express, for example, the coordinate value* 11.73452° *as the integer* 11734520*. This multiplication is not mandatory though.*

Configurations *(Def. 4.23) serve the purpose of defining the position of an object and its orientation in (planar) space. This is achieved by combining a* `Point` *and an* `Angle` *data type in the* `Configuration` *data type. Configurations can optionally hold additional properties, although this would require changing the implementation.*

Lines *(Def. 4.25) are realised as (open) polygons with integer coordinates. A line is a sequence of pairs* $L = (x_0,y_0),\ldots,(x_n,y_n)$*, with* $n > 0$ *(there has to be at least one segment). The* x_i *and* y_i *are* `Integer` *coordinates with the same properties as for point and polygon coordinates.*

Polygons *(Def. 4.27) are realised as closed normalised polygons with integer coordinates. A polygon is a sequence of pairs* $P = (x_0,y_0),\ldots,(x_n,y_n)$*, with* $n > 1$ *and the final line segment between* x_n,y_n *and* x_0,y_0 *(there have to be at least three segments, counter clockwise, no intersections). The* x_i *and* y_i *are* `Integer` *coordinates with the same properties as for point and line coordinates.*

Lists *(Def. 4.30) serve to handle ordered lists of MPLL entities, such as points, configurations, or any other type. They are used, for example, in the representation of multilines or polygons. Apart from points and configurations, as well as floats and integers, (which are automatically cast whenever necessary), the list elements have to be of the same type.*

Intervals *(Def. 4.35) are realised as* polygons *with integer coordinates. An interval is therefore a sequence of pairs* $I = (x_0,y_0),\ldots,(x_n,y_n)$*. The* x_i *are* `Angle` *values and the* y_i *are fuzzy values. Internally, the* y_i *are realised as short integers between 0 and 1000. From the MPLL point of view, however, the* y_i *are* `Float` *numbers between 0 and 1. The interval I is* negative infinite *if* $y_0 \neq 0$*. I is* positive infinite *if* $y_n \neq 0$*. The internal representation of* `Interval` *data, however, is completely*

[1] This may change in future releases to support Unicode.

invisible to the MPLL user. Details about the internal representation and the algorithms can be found in the REWERSE [164] deliverable A1-D1 [128].

■

The data structure types are used as types for variables, but they can also be used explicitly as constants, so-called literals. To this end, there is a string representation of the data structure types. These strings are parsed by the MPLL parser and mapped to the internal representation.

Remark 4.4.1 (String Representation of Data Structure Types)
The data structure types have the following string representation:

`Integer`*: Sequences of digits, optionally preceded by '+' or '-'. Examples are 123, +4, -345. The maximum length of these sequences depends on the internal representation of* `Integer` *values.*

`Float`*: Standard representation of* `Float` *or* `Double` *values. Examples are -1.5, 3.4e-2, -77e+5. The length of base and exponent depends on the internal representation of* `Float` *values.*

`String`*: Arbitrary sequences of characters enclosed in quotes: "characters". The two characters "\n" are interpreted as the newline command. A quote (") within the string must be escaped with a \ character. Therefore, the character sequence "*`ab\"cd\"ef`*" is parsed as the string "*`ab"cd"ef`*".*

`Angle`*: Standard representation of angular values, optionally preceded by '+' or '-' and followed by 'D', 'G', or 'R', denoting degree, grad[2], and radian. Examples: -1.5D, +45.654D, 3.141528R. The length of these representations depend on the internal representation of* `Float` *or* `Double` *values.*

If an angular value is not preceded by '+' or '-' and not followed by 'D', 'G', or 'R', the sign can be given alternatively by a trailing 'N' or 'E' (meaning '+') or a trailing 'S' or 'W' (meaning '-'). This format pertains to geospatial coordinates which adhere to this convention. Also, if coordinates are specified this way, longitude (direction E–W) is restricted to $]-180°,180°]$ *and latitude (direction N–S) is restricted to* $[-90°,90°]$*. The input/output of coordinates specified in this way is always done as degree, i.e. as if the value was specified with a trailing 'D'. Internally, however, angles are always stored and processed as radian.*

`Point`*: Points are pairs of* `Float` *or* `Integer` *values which are enclosed by the standard brackets "(" and ")" and separated by a space "␣" character, for example* `(45.24 11.37)` *or* `(2433 3732)`*.*

[2] The internationally standardised denomination *gon* has not yet replaced the English *grad*. However, the unit identifier ^{g} is unambiguous.

`Configuration`: *A configuration is a compound of an angle and a point. Therefore, the standard representation is a collection of the two in the following form:* `(Angle, Point)`*, for example* `(45.24D, (57.23 42.37))`*. An angle specified this way is always restricted to the respective interval* $[-2\pi, 2\pi[$ *(radian) or* $[-360°, 360°[$ *(degree), or* $[-400^g, 400^g[$ *(grad).*

`Line`: *A line consists of two or more point coordinates which are comma separated, without additional brackets. Examples are:*
`(45.23 11.67, 52.32 14.53, 77.23 10.28)` *or*
`(2423 3632, 3732 1521, 5323 1521)`
Commas may be followed by an additional space "␣" character.

`Polygon`: *A polygon is almost identically defined as a line with the exception that curly braces are used instead of standard brackets. In addition, polygons are always closed, i.e. the last coordinate tuple is connected to the first one. Examples are:*
{`45.43 11.37, 52.42 14.56, 77.23 10.27`} *or*
{`4323 6732 1521, 4723 3732 1521, 2523 3732 1821`}.
Commas may be followed by an additional space "␣" character.

`List`: *The string representation of a list is a comma separated concatenation of list elements. The string representation of the elements depends on the type of the elements.*

`Interval`: *Intervals cannot be explicitly referenced within an MPLL function definition. The only exception is the empty interval, which is represented by [ ]. The MPLL module, however, provides an interface function which allows one to call MPLL functions with a string representation of the arguments. This function accepts non-negative integers as identifiers for the intervals, together with a vector of pointers to the actual intervals. The integer identifiers are used as indices to this vector.*

■

A number of enumeration types is predefined in MPLL. They are used to control some of the algorithms. Their meaning therefore depends on the meaning of the built-in function where they occur as parameters.

Definition 4.17 (Enumeration Types)
A list of enumeration types is given in table 4.2.

■

Notice the multiple use of some keywords in table 4.2. For example, the keyword `core` occurs in the enumeration types `Region` and `Hull`. To determine which type has to be used, the context has to be evaluated. If the keyword `core` for example occurs in the `during` construct, then it can only be of type `Region`.

type	*possible values*
`Bool`	`true`, `false`
`Side`	`left`, `right`
`PosNeg`	`positive`, `negative`
`UpDown`	`up`, `down`
`ForwardBackward`	`forward`, `backward`
`InsideOutside`	`inside`, `outside`
`CarDir`	`north`, `northeast`, `east`, `southeast`, `south`, `southwest`, `west`, `northwest`, `N`, `NNE`, `NE`, `ENE`, `E`, `ESE`, `SE`, `SSE`, `S`, `SSW`, `SW`, `WSW`, `W`, `WNW`, `NW`, `NNW`
`EgoDir`	`front`, `left_front`, `left`, `left_back`, `back`, `right_back`, `right`, `right_front`
`EgoDirD`	`in_front_of`, `left_of`, `behind`, `right_of`
`Distance`	`very_close`, `close`, `commensurate`, `far`, `very_far`
`RSType`	`cartesian`, `geospherical`
`Region`	`core`, `kernel`, `support`, `maximum`
`Hull`	`core`, `kernel`, `support`, `maximum`, `crisp`, `monotone`, `convex`
`Fuzzify`	`linear`, `gaussian`
`SDVersion`	`Kleene`, `Lukasiewicz`, `Goedel`

Table 4.2.: Definition of Enumeration Types

However, if the context cannot be determined implicitly, for example in comparisons 'expression == keyword', one can use the explicit versions `Rcore` (of type `Region`) or `Hcore` (of type `Hull`). The same holds for other keywords which are used more than once.

An unknown string is parsed in the following order:

1. is it an `Integer` value?
2. is it a `Float` value?
3. is it an `Angle` value?
4. is it an (empty) `Interval`?
5. is it a keyword of one of the enumeration types?

If none of these types can be detected, a parse error is generated.

Definition 4.18 (Basic Types)
A Basic Type in MPLL is either a data structure type *(Def. 4.16, table 4.1), an* enumeration type *(Def. 4.17, table 4.2), or the special type* `Void` *for expressions which do not return any values.* ■

Automatic Type Conversion:
Automatic type conversion is done from the type `Integer` to the type `Float`. That means, the type `Integer` is also acceptable whenever a type `Float` is required.

4.4.2. Angles

Angles are an important basic data structure for MPLL since they represent an essential building block for human spatial reasoning, as has been illustrated in sections 2.4.2 and 2.5.

Internally, angles are represented by `Double` variables containing grad values. If degree or radian are to be used for input/output, this must specifically be requested. Negative angles represent left turns, positive angles represent right turns.

Angles are used for different purposes. Geospatial coordinates are usually specified as angles in the interval $]-\pi,\pi]$ (respectively $]-180°,180°]$) for lateral coordinates, and in the interval $[-\frac{\pi}{2},\frac{\pi}{2}]$ (respectively $[-90°,90°]$) for longitudinal values (see section 2.3.4). Here, negative values represent southern or western locations, positive values represent northern or eastern locations.

In other cases, for example orientation, angles denote the geometric orientation of an object in the interval $[0,2\pi[$ (respectively $[0°,360°[$). Note the different sizes and positions of the intervals.

All these and future applications of angles should, and can, be represented by a single `Angle` data type, which includes the necessary mechanisms to deal with the specific aspects in a transparent way.

Definition 4.19 (Construction)
The default method to construct an angle is to provide a grad value and the respective minimum and maximum values as `Float` *data types.*

Alternative constructors are part of the standard library (see Def. 4.79, pp. 153, in section 4.6).

(1) `Angle`(*Fangle*, *Fmin*, *Fmax*) `Float*Float*Float` $\mapsto$ `Angle`

■

The constructor (1) accepts a negative or positive grad value as a `Float` or an `Integer` value (*Fangle*) and the minimum and maximum values as `Float` or `Integer` values (*Fmin*, *Fmax*). If the angle is manipulated and its value leaves the range $]Fmin,Fmax]$, it is treated as $Fangle\%(Fmax-Fmin)$, i.e. $\mathtt{mod}(Fangle,(Fmax-Fmin))$.

Definition 4.20 (Predicates)
MPLL provides the following predicates to ascertain the properties of an angle A.

(1) `inverse`(A) `Angle` $\mapsto$ `Angle`
(2) `isNegative`(A) `Angle` $\mapsto$ `Bool`
(3) `radian`(A) `Angle` $\mapsto$ `Float`
(4) `degree`(A) `Angle` $\mapsto$ `Float`
(5) `grad`(A) `Angle` $\mapsto$ `Float`
(6) `min`(A) `Angle` $\mapsto$ `Float`
(7) `max`(A) `Angle` $\mapsto$ `Float`
(8) `modulus`(A) `Angle` $\mapsto$ `Float`
(9) `abs`(A) `Angle` $\mapsto$ `Angle`

■

(1) returns an angle representing the inverse of the angle A. Note that the inverse of an angle holding a real value is its negative, but the inverse of an angle holding a grad, degree or radian value is the angle poiting in the opposite direction.

(2) returns `true` if the angle A contains a negative value.

(3) returns the value of the angle A in radian as a `Float`.

(4) returns the value of the angle A in degree as a `Float`.

(5) returns the value of the angle A in grad as a `Float`.

(6) returns the minimum value of the angle A in radian as a `Float`. Minimum pertains to the valid minimum, i.e. the lowest value allowed for this type of angle. This value is the lower bound for the interval returned by function (5).

(7) returns the maximum value of the angle A in radian as a `Float`. Maximum pertains to the valid maximum, i.e. the highest value allowed for this type of angle. This value is the upper bound for the interval returned by function (5).

(8) returns the modulus of the interval for valid angles of this type as a `Float`.

(9) returns the absolute value of A as an `Angle`.

4.4.3. Points

This basic type is an essential part of the `Configuration` data type. Since some situations require only this part of configurations, we provide the individual data type `Point`.

It is important to clarify why the coordinates are held in the `Angle` data type, rather than a regular numerical type, such as `Integer` or `Float`. Primarily, this is done this way, because the processing of spatial information in the application domain of MPLL, i.e. geospatial scenarios, mostly operates on *angular* coordinates, such as WGS-84 (as provided, for example, by GPS receivers; see also section 2.3.4). This is, however, not

a restriction, since the `Angle` data type can also hold real values (see Def. 4.19), and could easily be extended to hold integer values as well, should the need arise.

Definition 4.21 (Construction)
The default method to construct a `Point` *is to provide two coordinate values as* `Angle` *data types.*

Alternative constructors are part of the standard library (see Def. 4.81, pp. 156, in section 4.6).

(1) `Point`(Ax, Ay) `Angle*Angle` $\mapsto$ `Point`

■

(1) accepts a tuple of coordinates specified as `Angle` values.

Definition 4.22 (Predicates)
MPLL provides the following predicates to ascertain the properties of a point P:

(1) `xCoordinate`(P) `Point` $\mapsto$ `Angle`
(2) `yCoordinate`(P) `Point` $\mapsto$ `Angle`

■

(1) returns the *x* coordinate of a point as an `Angle`.

(2) returns the *y* coordinate of a point as an `Angle`.

4.4.4. Configurations

Configurations (see section 2.4.1) in MPLL provide the necessary data structures to hold point entities and an orientation value in planar space.

Definition 4.23 (Construction)
Configurations are constructed by explicitly specifying the orientation and the x and y coordinates as `Angle` *values, and by specifying the validity of the orientation as a* `Bool` *value.*

Alternative constructors are part of the standard library (see Def. 4.82, pp. 156, in section 4.6).

(1) `Configuration`(A, Ax, Ay, Bo)
`Angle*Angle*Angle*Bool` $\mapsto$ `Configuration`

■

(1) accepts an orientation and a pair of coordinates specified as `Angle` values. Additionally, `hasOrientation` must be specified as a `Bool` value. Setting *Bo* to `false` only makes sense if *A* is not relevant and is not set or set to 0 (which is equivalent).

Definition 4.24 (Predicates)
MPLL provides the following predicates to ascertain the properties of a configuration C:

(1) `hasOrientation`(C) `Configuration` $\mapsto$ `Bool`
(2) `orientation`(C) `Configuration` $\mapsto$ `Angle`
(3) `point`(C) `Configuration` $\mapsto$ `Point`
(4) `xCoordinate`(C) `Configuration` $\mapsto$ `Float`
(5) `yCoordinate`(C) `Configuration` $\mapsto$ `Float`

■

(1) returns whether the configuration C features an orientation. This cannot simply be represented by a value of 0, since this would render a valid angle. Internally, this is represented by an additional binary member variable.

(2) returns the orientation of a configuration, i.e. the direction of its intrinsic front (or zero-point), as an `Angle`.

(3) returns the x and y coordinate of a configuration in form of a `Point`.

(4) returns the x coordinate of a configuration as an `Integer`.

(5) returns the y coordinate of a configuration as an `Integer`.

4.4.5. Lines

Lines in MPLL hold the necessary data to handle linear entities in planar space.

Definition 4.25 (Construction)
Lines are constructed by specifying a sequence of points, whereas the number of points must be at least 2*. Lines are always stored in normalised form, i.e. all line segments are specified by a sequence of points. Therefore, all segments are defined in the same "direction".*

Alternative constructors are part of the standard library (see Def. 4.83, pp. 158, in section 4.6).

(1) `Line`($P1, P2, \ldots, Pn$)
`Point`$*$`Point`$*\ldots*$`Point` $\mapsto$ `Line`

■

(1) accepts a sequence of n points with $n \geq 2$.

Definition 4.26 (Predicates)
MPLL provides the following predicates to ascertain the properties of a line L:

(1) `xMin`(L) `Line` $\mapsto$ `Angle`
(2) `yMin`(L) `Line` $\mapsto$ `Angle`
(3) `xMax`(L) `Line` $\mapsto$ `Angle`
(4) `yMax`(L) `Line` $\mapsto$ `Angle`
(5) `pointList`(L) `Line` $\mapsto$ `List`

■

(1) returns the minimum value of x coordinates of *all* points which make up the line. This is the lower horizontal bound (leftmost point), i.e. the lowest x coordinate of the bounding box.

(2) returns the minimum value of y coordinates of *all* points which make up the line. This is the lower vertical bound (lowest point), i.e. the lowest y coordinate of the bounding box.

(3) returns the maximum value of x coordinates of *all* points which make up the line. This is the upper horizontal bound (rightmost point), i.e. the highest x coordinate of the bounding box.

(4) returns the maximum value of y coordinates of *all* points which make up the line. This is the upper vertical bound (highest point), i.e. the highest y coordinate of the bounding box.

(5) returns the ordered list of points which make up the line.

4.4.6. Polygons

Polygons in MPLL hold the necessary data to handle region entities in planar space.

Definition 4.27 (Construction)
Polygons are constructed by specifying a sequence of points, whereas the number of points must be at least 3*, and the polygon is, by default, closed (i.e. there exists an implicit connection between the last and first point in the sequence).*

Alternative constructors are part of the standard library (see Def. 4.85, pp. 158, in section 4.6).

(1) `Polygon`$(P1, P2, \ldots, Pn)$
`Point`$*$`Point`$*\ldots*$`Point`$\mapsto$`Polygon`

■

(1) accepts a sequence of n points with $n \geq 3$.

Definition 4.28 (Predicates)
MPLL provides the following predicates to ascertain the properties of a polygon R:

(1) `xMin`(R) `Polygon`$\mapsto$`Angle`
(2) `yMin`(R) `Polygon`$\mapsto$`Angle`
(3) `xMax`(R) `Polygon`$\mapsto$`Angle`
(4) `yMax`(R) `Polygon`$\mapsto$`Angle`
(5) `pointList`(R) `Polygon`$\mapsto$`List`

■

(1) returns the minimum value of x coordinates of *all* points which make up the polygon. This is the lower horizontal bound (leftmost point), i.e. the lowest x coordinate of the bounding box.

(2) returns the minimum value of y coordinates of *all* points which make up the polygon. This is the lower vertical bound (lowest point), i.e. the lowest y coordinate of the bounding box.

(3) returns the maximum value of x coordinates of *all* points which make up the polygon. This is the upper horizontal bound (rightmost point), i.e. the highest x coordinate of the bounding box.

(4) returns the maximum value of y coordinates of *all* points which make up the polygon. This is the upper vertical bound (highest point), i.e. the highest y coordinate of the bounding box.

(5) returns the ordered list of points which make up the polygon. The last point in the list is not identical to the first. Although polygons are always closed, the final connection is handled implicitely, no redundant data is stored.

Definition 4.29 (Topological Predicates of Polygons)
MPLL provides the following predicates to ascertain topological relations between polygons R and S. These predicates are derived from the Region Connection Calculus (RCC), respectively the RCC-8 relations.

(1) `DC`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(2) `EC`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(3) `PO`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(4) `TPP`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(5) `NTPP`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(6) `TPPinverse`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(7) `NTPPinverse`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

(8) `EQ`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `Bool`

■

The Region Connection Calculus (RCC) and the RCC-8 relations are described in section 1.2.3, pp. 17. Currently, MPLL does not feature qualitative reasoning based on the RCC. However, an interface to an external service or module could facilitate this in the future.

(1) checks whether the two polygons R and S are disconnected (DC).

(2) checks whether the two polygons R and S are externally connected (EC).

(3) checks whether the two polygons R and S overlap partially (PO).

(4) checks whether the polygon R is a tangential proper part of the polygon S (TPP).

(5) checks whether the polygon R is a non-tangential proper part of the polygon S (NTPP).

(6) checks whether the polygon S is a tangential proper part of the polygon R (TPP^{-1}). This is the inverse of predicate (4).

(7) checks whether the polygon S is a non-tangential proper part of the polygon R (NTPP^{-1}). This is the inverse of predicate (5).

(8) checks whether the two polygons R and S are equal (EQ).

4.4.7. Lists

Lists in MPLL provide the necessary structures to handle lists of types and spatial entities.

Definition 4.30 (Construction)
Lists can be constructed in two ways. An empty list can be constructed using a valid type as the single parameter. Lists can also be explicitly constructed, by specifying a number of (same type) parameters in a comma separated sequence, enclosed in curly braces ({ and }).

(1) `emptyList(`T`)`
`T` $\mapsto$ `List`

(2) $\{T_1, \ldots, T_n\}$
`T` $*\ldots*$ `T` $\mapsto$ `List`

■

(1) constructs an empty list for objects of type T.

(2) constructs a list containing objects $T_1, \ldots, T_n$ of type T.

Definition 4.31 (Predicates)
MPLL provides the following predicates to ascertain the properties of a list L:

(1) `head`(L)
of type
`List<T>` $\mapsto$ `T`

(2) `tail`(L)
of type
`List<T>` $\mapsto$ `List<T>`

(3) `length`(L)
of type
`List<T>` $\mapsto$ `Integer`

(4) `sublist`(L,N,M)
of type
`List<T>` $*$ `Integer` $*$ `Integer` $\mapsto$ `List<T>`

(5) `prefix`(L,N)
of type
`List<T>` $*$ `Integer` $\mapsto$ `List<T>`

(6) `suffix`(L,N)
of type
`List<T>` $*$ `Integer` $\mapsto$ `List<T>`

(7) `append`(T,L)
of type
`T` $*$ `List<T>` $\mapsto$ `List<T>`

(8) `append`(L,L)
of type
`List<T>` $*$ `List<T>` $\mapsto$ `List<T>`

(9) `split`(L,N)
of type
`List<T>` $*$ `Integer` $\mapsto$ `List<List<T>>`

(10) `filter`(*Condition*,L)
of type
(`T` $\mapsto$ `Bool`) $*$ `List<T>` $\mapsto$ `List<T>`

(11) `map`(*Function*,L)
of type
(`T` $\mapsto$ `S`) $*$ `List<T>` $\mapsto$ `List<S>`

■

(1) returns the first element of the list L, it is of type T.

(2) return the list L without the first element, i.e. without the head.

(3) returns the number of elements of the list as an integer.

(4) returns the sublist from at a specific index N to a specific index M.

(5) returns the sublist from the first element of the list L to a specific index N.

(6) returns the sublist from a specific index N to the end of the list L.

(7) appends an element of type T as the first element of the list L.

(8) concatenates two lists L and M.

(9) splits a list L at a given position into two lists returning a list containing both lists.

(10) returns a list of all elements of type T in L for which condition(T) holds true.

(11) returns a list $\{Function(T_1), \ldots, Function(T_n)\}$ of type S.

4.4.8. Reference Systems

Reference Systems in MPLL provide the necessary data structures which represent the frame for spatial modelling. Currently, there are two different types of reference systems available: cartesian (planar) and geospherical.

Reference systems need to be constructed with several parameters. The origin is always located implicitly at $(0,0)$, or $(0,0,0)$ respectively. This is important for coordinate transformation from one reference system to another. Maxima and Minima for x, y, and optionally z axes, have to be specified. Coordinates outside of these limits are invalid. The type of the reference system has to be specified as the enumeration type `RSType`. This has great influence on all computations, as well as on wraparound. Therefore, all specific properties of the reference system type have to be backed by the reference system module of the implementation. The underlying mechanisms cannot be coded in MPLL, they have to be hard-coded in the implementation. In addition, the optional vertical and/or horizontal wraparound has to be specified using binary values. Wraparound is also specific for the reference system type. Its internal mechanisms are also found in the implementation.

Definition 4.32 (Construction)
There exist two different constructors for reference systems:

(1) `ReferenceSystem`($cartesian/geospherical, N, N, N, N, B, B$)
`RSType*Integer*Integer*Integer*Integer*Bool*Bool`
$\mapsto$ `ReferenceSystem`

(2) `ReferenceSystem`($cartesian/geospherical, P, P, B, B$)
`RSType*Point*Point*Bool*Bool`
$\mapsto$ `ReferenceSystem`

■

Note that the `RSType` defines, for example, how the upper and lower bounds for x and y coordinates have to be interpreted. In case the reference system is of type `geospherical`, this results in the half-open interval $]-\pi,\pi]$ for x values and the closed interval $[-\frac{\pi}{4},\frac{\pi}{4}]$ for y values. For a `cartesian` reference system, these intervals are both closed.

(1) accepts a reference system type `RSType`, four `Float` or `Integer` coordinates which define minimum x, maximum y, maximum x and minimum y coordinates[3], and two `Bool` values which mark vertical and horizontal wraparound.

(2) accepts a reference system type `RSType`, two `Point` variables defining upper left and lower right coordinates, and two `Bool` values which mark vertical and horizontal wraparound.

Definition 4.33 (Predicates)
MPLL provides the following predicates to ascertain the properties of a reference system RS:

(1)	`isCartesian`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(2)	`isGeospherical`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(3)	`isOfType`($cartesian/geospherical, RS$)	`RSType` $*$ `ReferenceSystem` $\mapsto$ `Bool`
(4)	`hasVerticalWrap`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(5)	`hasHorizontalWrap`(RS)	`ReferenceSystem` $\mapsto$ `Bool`

■

(1) returns whether the reference system is of `RSType` `cartesian`.

(2) returns whether the reference system is of `RSType` `geospherical`.

(3) returns whether the reference system is of the given `RSType` (`cartesian` or `geospherical`).

(4) returns whether the reference system features vertical wraparound.

(5) returns whether the reference system features horizontal wraparound.

Definition 4.34 (Property Access)
MPLL provides the following functions to access the properties of reference systems C:

(1)	`xMin`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(2)	`yMin`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(3)	`xMax`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(4)	`yMax`(RS)	`ReferenceSystem` $\mapsto$ `Bool`
(5)	`upperLeft`(RS)	`ReferenceSystem` $\mapsto$ `Point`
(6)	`lowerRight`(RS)	`ReferenceSystem` $\mapsto$ `Point`

■

(1) returns the minimum value of x coordinates which are allowed within the reference system.

(2) returns the minimum value of y coordinates which are allowed within the reference system.

[3]This order is derived from the convention to specify minima and maxima using *upper left* and *lower right* coordinate pairs.

(3) returns the maximum value of x coordinates which are allowed within the reference system.

(4) returns the maximum value of y coordinates which are allowed within the reference system.

(5) returns the topmost leftmost point of the reference system in form of a `Point`.

(6) returns the lowest rightmost point of the reference system in form of a `Point`.

4.4.9. Intervals

There are different types of intervals, for example the ones handling angles (wraparound, deg or rad) or distances (no wraparound, different distance metrics).

Explicit Construction of Intervals

Fuzzy intervals (type `Interval`) are one of the built-in data structures in MPLL. It is possible to create new empty intervals and fill them up with coordinate points. There are three ways to create new intervals in MPLL:

Definition 4.35 (New Intervals)

1. *The expression* $[]$ *stands for the empty interval.*

2. *The expression* $[x_1, x_2]$ *of type* `Float*Float` $\mapsto$ `Interval` *constructs a new crisp interval with boundaries* x_1 *and* x_2.

3. *The expression* $[(x_1, y_1), (x_2, y_2)]$ *of type* `Float*Float*Float*Float` $\mapsto$ `Interval` *constructs a new fuzzy interval with the given two points.* ■

Definition 4.36 (Extending Intervals)
The function
`pushBack`$(I, value, fuzzyvalue)$
of type
`Interval*Float*Float` $\mapsto$ `Void`
adds the point $(value, fuzzyvalue)$ *to the end of the interval I. I must be an interval which was constructed with* `newInterval()` *(see Def. 4.35). value must be greater than the last* x_i *in the interval. value must be a* `Float` *value between 0 and 1.* ■

The `pushBack`$(I, value, fuzzyvalue)$ function can only fill up the interval I from smaller values to larger values. It throws an error if *value* is smaller than the highest value in I.

Set Operations on Intervals

For crisp intervals, the standard set operators exist: `complement`, `intersection`, `union` etc. These are uniquely defined. There is no choice. Unfortunately (or fortunately, because it gives you more flexibility), there are no such uniquely defined set operators for fuzzy intervals. Set operators are essentially transformations of the membership functions, and there exist a number of different ones.

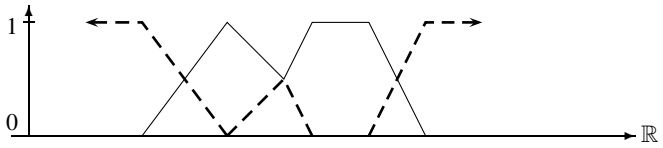
Like GeTS, MPLL offers the same standard versions of the set operators, parameterised set operators of the Hamacher family, and set operators with transformation functions for the membership function as parameter. These allow one to customise the set operators in an arbitrary way.

Definition 4.37 (Complement of Intervals)
Let I be an expression of type `Interval`*. The complement operation for intervals comes in three versions:*

(1) `complement`(I) $\quad$ `Interval` $\mapsto$ `Interval`
(2) `complement`(I,λ) $\quad$ `Interval*Float` $\mapsto$ `Interval`
(3) `complement`$(I,$*negation_function*$)$
$\quad$ `Interval*(Float` $\mapsto$ `Float)` $\mapsto$ `Interval`

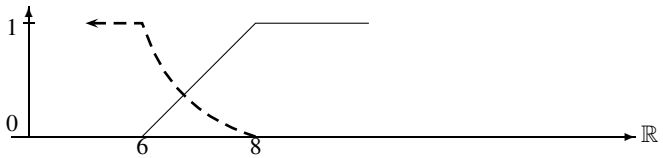
■

Version (1) is the standard complement. Each point (x,y) of the membership function of I is turned into $(x,1-y)$.



Standard Complement for a Fuzzy Interval

Version (2) is the *lambda-complement*. For $\lambda > -1$, each point (x,y) of the membership function of I is turned into $(x, \frac{1-y}{1+\lambda y})$. The ordinary complement is computed for $\lambda \leq -1$.



λ-Complement for $\lambda = 2$

Finally, with version (3) it is possible to submit a user defined negation function. For example, with

```
lambda_complement(Interval I, Float lam)
= complement(I,lambda(Float y) (1-y)/(1+lam*y))
```

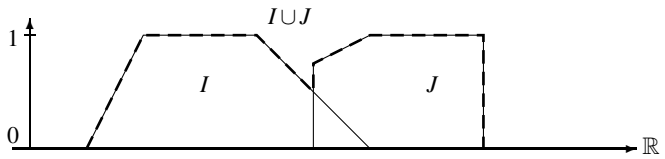
one can define the same lambda-complement with a user defined negation function.

Definition 4.38 (Union of Intervals)
Let I and J be expressions of type `Interval`*. The union operation for intervals comes in three versions:*

(1) `union`(I,J) `Interval*Interval` $\mapsto$ `Interval`
(2) `union`(I,J,β) `Interval*Interval*Float` $\mapsto$ `Interval`
(3) `union`$(I,J,$*co_norm*$)$
`Interval*Interval*(Float*Float` $\mapsto$ `Float)` $\mapsto$ `Interval`

■

Version (1) is the standard union. Each pair (x,y_1) and (x,y_2) of points of the membership function of I and J is turned into $(x,max(y_1-y_2))$.



Standard Union of Fuzzy Sets

Version (2) is the so-called *Hamacher–Union*. For $\beta \geq -1$, each pair (x,y_1) and (x,y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1+y_2+(\beta-1)y_1y_2}{1+\beta y_1y_2})$. The ordinary union is computed for $\beta < -1$.

Hamacher–Union with $\beta = 0.5$

Version (3) of the union function facilitates the submission of a user defined co-norm[4]. For example, with

```
Hamacher_Union(Interval I, Interval J, Float beta)
= union(I, J, lambda(Float y1, Float y2)
    (y1+y2+((beta - 1)*y1*y2))/(1+beta*y1*y2))
```

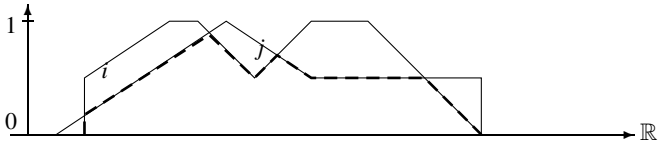
one can define the same Hamacher union with a user defined co-norm.

Definition 4.39 (Intersection of Intervals)
Let I and J be expressions of type `Interval`*, The intersection operation for intervals comes also in three versions:*

(1) `intersection`(I,J) $\quad$ `Interval*Interval` $\mapsto$ `Interval`
(2) `intersection`(I,J,γ)
$\quad$ `Interval*Interval*Float` $\mapsto$ `Interval`
(3) `intersection`$(I,J,norm)$)
$\quad$ `Interval*Interval*(Float*Float` $\mapsto$ `Float)` $\mapsto$ `Interval`

■

Version (1) is the standard intersection. Each pair (x,y_1) and (x,y_2) of points of the membership function of I and J is turned into $(x,min(y_1-y_2))$.



Standard Intersection of Fuzzy Sets

Version (2) is the Hamacher–Intersection. For $\gamma \geq 0$, each pair (x,y_1) and (x,y_2) of points of the membership function of I and J is turned into $(x, \frac{y_1 y_2}{\gamma+(1-\gamma)(y_1+y_2-y_1y_2)})$. The ordinary intersection is computed for $\gamma < 0$.

Hamacher–Intersection $\gamma = 0.5$

[4]Norms and co-norms are binary functions on membership values of fuzzy sets. They satisfy conditions which make sure that the corresponding set operations can be considered as union and intersection [47].

Version (3) provides the possibility for specifying a user defined norm. For example, with

```
Hamacher_Intersection(Interval I, Interval J, Float gamma)
 = intersection(I, J, lambda(Float y1, Float y2)
     (y1*y2)/(gamma + (1-gamma)*(y1 + y2 -y1*y_2))
```

one can define the same Hamacher-Intersection with a user defined norm.

Definition 4.40 (Set Difference between Intervals)
Let I and J be expressions of type `Interval`*. The set difference operation for intervals comes also in three versions:*

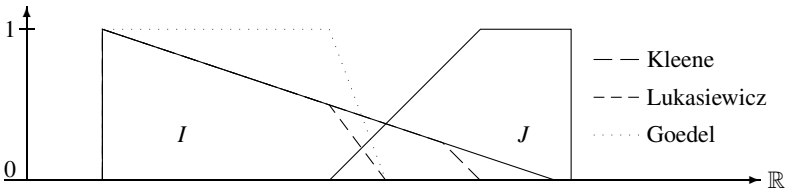
(1) `setdifference`(I,J)
$$\texttt{Interval}*\texttt{Interval}\mapsto\texttt{Interval}$$
(2) `setdifference`$(I,J,$*version*$)$
$$\texttt{Interval}*\texttt{Interval}*\texttt{SDVersion}\mapsto\texttt{Interval}$$
(3) `setdifference`$(I,J,$*intersection*$,$*complement*$)$
$$\texttt{Interval}*\texttt{Interval}*(\texttt{Interval}*\texttt{Interval}\mapsto\texttt{Interval})*(\texttt{Interval}\mapsto\texttt{Interval})\mapsto\texttt{Interval}$$

■

(1) extends the crisp correspondence: $I \setminus J = I \cap J'$ where J' is the complement of J, `set-difference(I,J)` is therefore an abbreviation for `intersection(I,complement(J))` with standard intersection and complement functions.

(2) computes the set difference operator by means of a binary function on the membership functions. The following versions are possible:

`SDVersion`	Function
`Kleene`	$(I \setminus J)(x) \stackrel{\text{def}}{=} min(I(x), 1-J(x))$
`Lukasiewicz`	$(I \setminus J)(x) \stackrel{\text{def}}{=} max(0, I(x)-J(x))$
`Goedel`	$(I \setminus J)(x) \stackrel{\text{def}}{=} 0$ if $I(x) \leq J(x)$ and $1-J(x)$ otherwise



Set Difference

(3) is a generalisation of the first version:

$$\texttt{setdifference}(I,J,\textit{intersection},\textit{complement}) \stackrel{\text{def}}{=} \textit{intersection}(I,\textit{complement}(J))$$

intersection is a user defined binary function on intervals, and *complement* is a user defined unary function on intervals.

Predicates of Intervals

Fuzzy intervals have many properties. They can be checked with suitable MPLL predicates.

Definition 4.41 (Predicates)
MPLL provides the following predicates to check the structure of an interval I:

(1)	`isCrisp`(I)	`Interval` $\mapsto$ `Bool`
(2)	`isCrisp`(I,`left/right`)	`Interval*Side` $\mapsto$ `Bool`
(3)	`isEmpty`(I)	`Interval` $\mapsto$ `Bool`
(4)	`isConvex`(I)	`Interval` $\mapsto$ `Bool`
(5)	`isMonotone`(I)	`Interval` $\mapsto$ `Bool`
(6)	`isInfinite`(I)	`Interval` $\mapsto$ `Bool`
(7)	`isInfinite`(I,`left/right`)	`Interval*Side` $\mapsto$ `Bool`

■

`isCrisp`(I) checks whether the interval I is a, possibly non-convex, crisp interval. `isCrisp`(I,`left`) checks whether the interval I is crisp at its left end. I may be infinite at this side, but the fuzzy value must be 1 in this case. Similar for `isCrisp`(I,`right`).

`isEmpty`(I) checks whether the interval I is empty.

`isConvex`(I) checks whether the interval I is convex. I can be non-convex even if I is crisp because it may consist of several different components.

`isMonotone`(I) checks whether the membership function of the interval I is monotonically rising to a maximal value, and then monotonically falling again.

`isInfinite`(I) checks whether the interval I is infinite.

`isInfinite`(I,`left`) checks whether the interval I is infinite on the left hand side. `isInfinite`(I,`right`) checks whether the interval I is infinite on the right hand side.

The boundaries of infinite intervals are of course the infinity. Infinity has a special representation in the `Angle` datatype. This can be checked with the `isInfinity` predicate:

Definition 4.42 (Infinity)

`isInfinity`(A)	`Angle` $\mapsto$ `Bool`
`isInfinity`(A,`positive/negative`)	`Angle*PosNeg` $\mapsto$ `Bool`

■

`isInfinity`(A) checks whether *A* represents an infinity.
`isInfinity`(A,`positive`) checks whether the *A* represents the positive infinity.
`isInfinity`(A,`negative`) checks whether the *A* represents the negative infinity.

The next three predicates allow one to check basic relations between angle values and intervals, or between intervals and intervals.

Definition 4.43 (`during, isSubset, doesOverlap`)

(1) `during`$(A,I,$`core/kernel/support`$)$
$\texttt{Angle} * \texttt{Interval} * \texttt{Region} \mapsto \texttt{Bool}$
(2) `isSubset`$(I,J,$`core/kernel/support`$)$
$\texttt{Interval} * \texttt{Interval} * \texttt{Region} \mapsto \texttt{Bool}$
(3) `doesOverlap`$(I,J,$`core/kernel/support`$)$
$\texttt{Interval} * \texttt{Interval} * \texttt{Region} \mapsto \texttt{Bool}$

■

(1) `during`$(A,I,$*region*$)$ checks whether *A* is inside the given region of the interval *I*.

(2) `isSubset`$(I,J,$*region*$)$ checks whether the corresponding region of the interval *I* is a subset of the corresponding region of the interval *J*.

(3) `doesOverlap`$(I,J,$*region*$)$ checks whether the corresponding region of the interval *I* overlaps the corresponding region of the interval *J*.

The point-interval `during` relation is one of the five point–interval relations 'before', 'starts', 'during', 'finishes' and 'after' for crisp intervals. Only `during` is built-in because it is one of the most frequently used relations. The other relations can easily be defined in MPLL.

Other Features of Intervals

With the first function in this paragraph one can access the fuzzy membership value of an angle within a given fuzzy interval.

Definition 4.44 (`member`) *The function*
`member`(A,I)
of type
$\texttt{Angle} * \texttt{Interval} \mapsto \texttt{Float}$
returns the value of the membership function of the interval I at angle value A. The value is a `Float` *number between 0 and 1.* ■

Definition 4.45 (Components)

1. *The function* `components`*(I) of type* $\texttt{Interval} \mapsto \texttt{Integer}$ *yields the number of components in the interval I.*

2. *The function* `component(I,k)` *of type* `Interval*Integer` $\mapsto$ `Interval` *extracts the* $\texttt{k}^{th}$ *component from the interval I.*

■

The function `size` below measures an interval I or parts of it by *integrating* over its membership function.

Definition 4.46 (`size`) *The function* `size` *comes in three versions.*

(1) `size`$(I)=$
`size`$(I,$`support`$)$
of type
`Interval` $\mapsto$ `Angle`
(2) `size`$(I,$`core/support/kernel`$)$
`Interval*IntvRegion` $\mapsto$ `Angle`
(3) `size`(I,A_1,A_2)
`Interval*Angle*Angle` $\mapsto$ `Angle`

■

(1) measures the size of the support of I.
(2) measures the size of the corresponding region of I.
(3) measures the area of I between A_1 and A_2.

The function '`point`' below can be used to access the boundaries of the three different regions of an interval: support, core and kernel, and the first and last maximal points.

Definition 4.47 (`point`) *The function*
`point`$(I,$`left/right,core/support/kernel/maximum`$)$
of type
`Interval*Side*PointRegion` $\mapsto$ `Angle`
returns the position of the boundaries of I's regions:

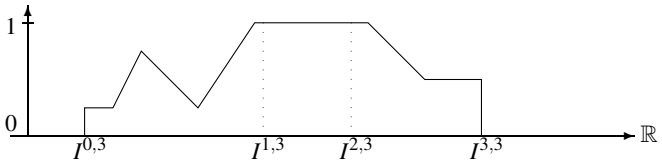
`point`$(I,$`left,support`$)$	*yields the position of the left support boundary*
`point`$(I,$`right,support`$)$	*yields the position of the right support boundary*
`point`$(I,$`left,core`$)$	*yields the position of the left core boundary*
`point`$(I,$`right,core`$)$	*yields the position of the right core boundary*
`point`$(I,$`left,kernel`$)$	*yields the position of the left kernel boundary*
`point`$(I,$`right,kernel`$)$	*yields the position of the right kernel boundary.*
`point`$(I,$`left,maximum`$)$	*yields the leftmost pos. of the max. fuzzy value.*
`point`$(I,$`right,maximum`$)$	*yields the rightmost pos. of the max. fuzzy value.*

■

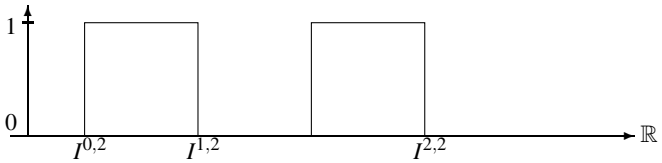
If `I` is just a convex crisp interval $[t_1,t_2[$ then
`point`$(I,$`left,support`$)=t_1$ and `point`$(I,$`right,support`$)=t_2$.

Centre Points
The n,m-*centre points* are used to express temporal notions like 'the first half of the year', or 'the second quarter of the year', or more exotic expressions like 'the 25th 49th of the weekend' etc. The notion of n,m-centre points makes only sense for finite intervals.

Example 4.4 (Centre Points) *The 1,2-centre point* $I^{1,2}$ *of I splits I in two halfs of the same size (integrated over the membership function). The 1,3-centre point indicates a split of I into three parts of the same size.* `centerPoint(I,1,3)` *is the boundary of the first third,* `centerPoint(I,2,3)` *is the boundary of the second third.*



$n,3$-*Centre Points*



$n,2$-*Centre Points*

■

Definition 4.48 (Centre Points) *The function*
`centerPoint`(I,n,m)
of type
`Interval*Integer*Integer` $\mapsto$ `Angle`
yields the (earliest) position of the n,m*-centre point.* ■

The centre points are computed such that for $n < m$:

$$\int_{centerPoint(n,m)}^{centerPoint(n+1,m)} I(x)\,dx = (\int I(x)\,dx)/m$$

Basic Manipulations of Intervals

In this paragraph we introduce some elementary transformation functions for fuzzy intervals.

Definition 4.49 (Shift of Intervals) *The function*
`shift`(I,A)
of type
`Interval*Angle` $\mapsto$ `Interval`
shifts the interval by the given angle value, i.e. `shift`$(I,A)(x) = I(x-t)$ ■

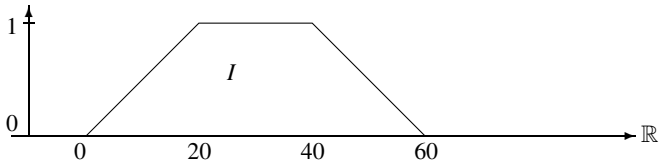
Definition 4.50 (`cut`)
The function
`cut`(I,A_1,A_2)
of type
`Interval*Angle*Angle` $\mapsto$ `Interval`
cuts the part of the interval I between the angles A_1 and A_2 out of I and returns it as a new interval. ■

The `hull` function below is able to compute different hulls of a fuzzy intervals.

Definition 4.51 (Hull Calculations)
The function
`hull`$(I,$`core/support/kernel/crisp/monotone/convex`$)$
of type
`Interval*Hull` $\mapsto$ `Interval`
computes a hull of the interval I. The second parameter determines which hull is to be computed. ■

The `core`, `support` and `kernel` hull compute the corresponding interval regions as crisp intervals. The `core` and `support` hull may therefore consist of different components, whereas the `kernel` hull consists of at most one single component.

There is a small problem with the `support` hull. Consider the following example:



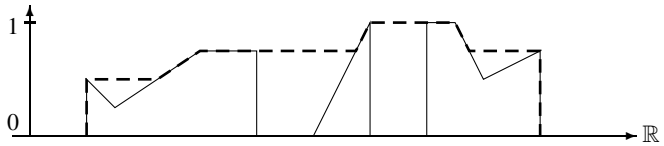
support hull problem

Since $I(0) = 0$, the support of I is the open interval $]0,60[$. The function `hull(I,support)`, however, calculates the interval boundaries 0 and 60, which are interpreted as the half open interval $[0,60[$. Strictly mathematical, this is not correct. In a correct implementation, however, we would have to distinguish open and half open intervals. Since the overhead for this is enormous, the current version of MPLL has to live with this error.

The `crisp` hull for crisp intervals is the usual convex hull of crisp intervals. It consists of the smallest crisp interval which contains all the components of the interval. The `crisp` hull for non-crisp intervals is the convex hull of the support of the interval. If the non-convex interval consists of one single component only, there is no difference between the crisp and support hull. In general we have

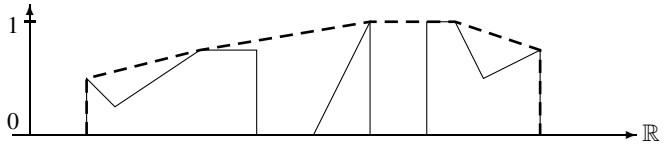
$$\texttt{hull}(I,\texttt{crisp}) = \texttt{hull}(\texttt{hull}(I,\texttt{support}),\texttt{crisp}).$$

The `monotone` hull of an interval I is the smallest *monotone* interval which contains I. An interval is *monotone* iff its membership function rises monotonically up to a maximal point, and then falls monotonically again.



Monotone Hull of a Fuzzy Interval

The `convex` hull of an interval I is the smallest *convex* interval which contains I. The notion 'convex', which is appropriate here, is the notion of a *convex polygon*. That means, if we follow the membership function from left to right there are only right curves. The next figure illustrates this.



Convex Hull of a Fuzzy Interval

If the interval I is crisp then the crisp, monotone and convex hull are the same.

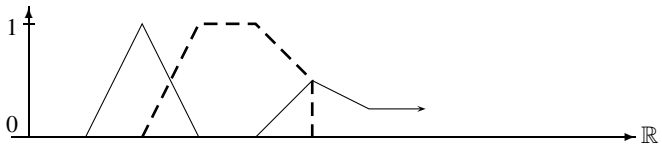
The next function can be used to extract the gaps between components of an interval. The `invert` function inverts the membership function, but only between the last maximal point of the first component and the first maximal point of the last component. `invert(I)` is zero outside these points.

Definition 4.52 (`invert`)
The function $\texttt{invert}(I)$ *of type* $\texttt{Interval} \mapsto \texttt{Interval}$ *inverts the membership function of the interval* I*:*

$$\texttt{invert}(I)(x) \stackrel{\text{def}}{=} \begin{cases} 1 - I(x) & \text{if } a \leq x < b \\ 0 & \text{otherwise.} \end{cases}$$

where a is the last maximal point of the first component of I, and b is the first maximal point of the last component of I. ■

Example:



Invert

`components(invert(`I`))` yields the number of gaps in the interval I.

The `scaleup` function below multiplies the membership function of an interval I with a factor f, such that the maximal value of $I(x) * f$ is 1.

Definition 4.53 (`scaleup`)
The function `scaleup`(I) *of type* `Interval` $\mapsto$ `Interval` *scales the membership function of I such that its maximum is 1.* ■

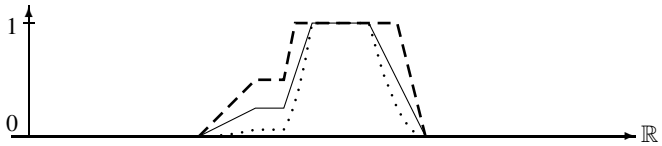
More general scaling functions are `times` and `exp`.

Definition 4.54 (`times` **and** `exp`)

`times`(I,f)	`Interval*Float` $\mapsto$ `Interval`
`exp`(I,e)	`Interval*Float` $\mapsto$ `Interval`

`times`$(I,f)(x) = min(I(x) \cdot f, 1)$.
`exp`(I,e) *computes an interval such that* $exp(I,e)(x) = I(x)^e$. ■

The dashed line in the next figure indicates `times`$(I,2)$ and the dotted line indicates `exp`$(I,2)$.

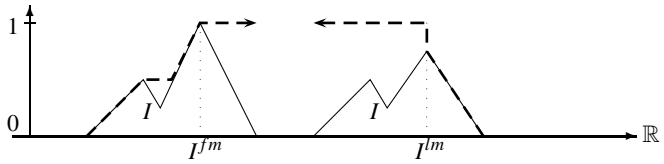


`times`$(I,2)$ *and* `exp`$(I,2)$

The rising part of a fuzzy interval is crucial for a fuzzy point-interval `before` relation. The falling part, on the other hand, is crucial for a point-interval `after` relation. The rising part of an interval I can be computed by following its monotone hull up to the first maximal point, and then extending it to the infinity. Similar with the falling part.

Definition 4.55 (Extend to Infinity)
The function
`extend`$(I,$`positive/negative`$)$
of type
`Interval*PosNeg` $\mapsto$ `Interval`
extends the interval to the infinity. `extend`$(I,$`positive`$)$ *raises the membership function of the monotone hull of I to 1 after the first maximum* I^{fm}. `extend`$(I,$`negative`$)$ *raises the membership function of the monotone hull of I to 1 before the right maximum* I^{lm}. ■

Example:

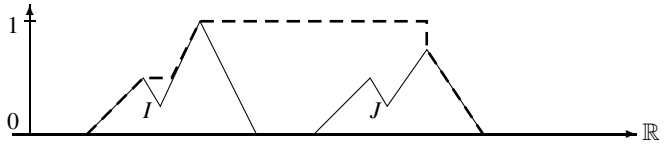


`extend(I,positive)` *and* `extend(I,negative)`

An example where the `extend` function is useful is the definition of the binary 'until' relation between two intervals.

```
until(Interval I, Interval J)
= intersection(extend(I,positive),extend(J,negative))
```
(4.2)

computes `until`(I,J) as the interval which lasts from the beginning of interval I until the end of interval J.



until

There is a further `extend` function in MPLL. It lengthens or shortens an interval by a certain amount.

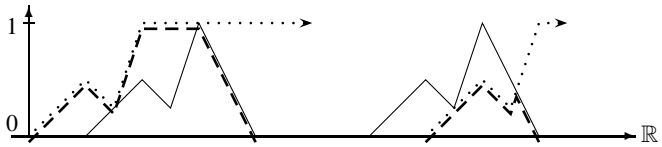
Definition 4.56 (Extend by a Certain Value)
The function

`extend`$(I, length, side)$
of type
`Interval*Angle*Side` $\mapsto$ `Interval`
extends the interval I by the given length. ■

The *side* parameter determines at which side the interval is extended. *side* = `left` extends it on the left side, *side* = `right` extends it on the right side. A positive *length* value causes the interval to be extended, whereas a negative *length* value causes the interval to be shrunken.

The algorithm for extending or shrinking a fuzzy interval works as follows: In a first step the interval I is split into the left/right part I_1 of the interval up to the first maximal point, and the rest I_2. I_1 is extended to the infinity. This part is shifted. If the interval is to be extended, then the union of the shifted I_1 with I_2 is computed. If the interval is to be shrunk then the intersection of the shifted I_1 with I_2 is computed. The next figure illustrates this. The dotted line shows the shifted front part of the interval. The dashed line is the result of the union/intersection.

Example:



extending and shrinking an interval by a certain duration

The `extend` function together with `shiftLength` can be used to extend an interval by a certain *duration*. For example,

`extend` $(I, -$`shiftLength(point(`I`,left,support),`
-1 `month,false,true),left)`

extends the left side of the interval I by 1 month. The month length is determined by a backwards shift of the left boundary of I's support.

Definition 4.57 (`integrate`)
The function
`integrate(`I`,positive/negative)`
of type
`Interval*PosNeg` $\mapsto$ `Interval`
integrates the membership function of I and normalises its value to 1. If the control parameter is `positive` *then I is integrated from left to right. If it is* `negative` *then I is integrated from right to left.* ■

Fuzzification

Fuzzy intervals could be defined by specifying the shape of the membership function in

some way. This is in general very inconvenient. Therefore MPLL provides an alternative. The idea is to take a crisp interval and to 'fuzzify' the front and back end in a certain way. For example, one may specify 'early afternoon' by taking the interval between 1 and 6 pm and imposing, for example, a linear or a Gaussian shape increase from 1 to 2 pm, and a linear or a Gaussian shape decrease from 4 to 6 pm. Technically this means multiplying a linear or Gaussian function with the membership values.

Definition 4.58 (Fuzzification)
There are two different versions of the `fuzzify` *function in MPLL. The first version allows one to specify the part of the interval I which is to be fuzzified in terms of percents of the interval length. The second version needs absolute coordinates.*

`fuzzify`$(I,$`linear/gaussian`$,$ `left/right`$,$*increase*$,$*offset*$)$
of type
`Interval,Fuzzify,Side,Float,Float` $\mapsto$ `Interval`

`fuzzify`$(I,$`linear/gaussian`$,$ `left/right`$,x1,x2,$*offset*$)$
of type
`Interval,Fuzzify,Side,Angle,Angle,Angle` $\mapsto$ `Interval`

■

The second parameter determines whether a linear or gaussian increase is to be imposed on the interval. The third parameter determines whether the increase is from left to right or from right to left. *increase* is a Float number in percent. *increase* $= 10$ means that the region to be modified consists of the first/last 10% of the *kernel* of the interval. *offset* is also a float number in percent. *offset* $= 20$ means that the interval is to be widened by 20% of the *kernel* of the interval. To this end the fuzzified part of the interval is shifted back (second parameter = `left`) or forth (second parameter = `right`) 20% of the kernel size.

$x1$ and $x2$ in the second version of the fuzzify function allows one to determine the part of the interval to be fuzzified in absolute coordinates.

$$\texttt{fuzzify}([0,100],\texttt{linear},\texttt{left},20,70,0),$$

for example, yields a polygon [(20,0) (70,1) (100,1) (100,0)].

$$\texttt{fuzzify}([0,100],\texttt{linear},\texttt{right},20,70,0),$$

on the other hand, yields a polygon [(0,0) (0,1) (20,1) (70,0)].

The offset widens the polygon:

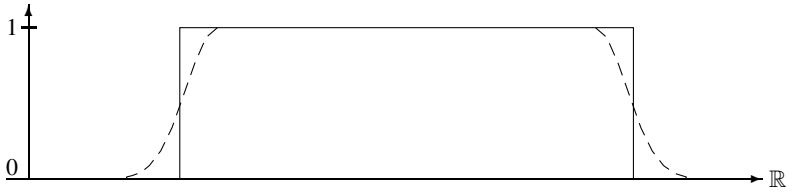
$$\texttt{fuzzify}([0,100],\texttt{linear},\texttt{right},20,70,20),$$

yields [(0,0) (0,1) (60,1) (90,0)].

A function which fuzzifies both ends of an interval in the same way could be

```
f(Interval I,Float increase,Float offset)=intersection(
   extend(fuzzify(I,gaussian,left,increase,offset),
          positive),
   extend(fuzzify(I,gaussian,right,increase,offset),
          negative))
```

$f(I,20,0)$ produces the following fuzzified interval.



Relative Gaussian Fuzzification

Notice that the obvious 'solution'

```
f(Interval I, Float increase, Float offset)
  = fuzzify(fuzzify(I,gaussian,right,inc,off),
            left,increae,offset)
```

yields no symmetric structure, because the inner `fuzzify` operation changes the kernel of the interval, such that the absolute increase and offset of the outer `fuzzify` operation are different to the absolute increase and offset of the inner `fuzzify` operation

Integration over Pairs of Intervals

One possibility to define an interval–interval relation like '$before(I,J)$' is, to take a point–interval relation '$PIRbefore(t,J)$' and average $PIRbefore(t,J)$ over the interval I. Averaging over an interval means integrating over its membership function. For purposes like this MPLL provides two integration operations.

Definition 4.59 (Integration)

MPLL has the two integration functions:

`integrateSymmetric`$(I,J,simple)$
`Interval*Interval*Bool` $\mapsto$ `Float` *and*
`integrateAsymmetric`(I,J)
`Interval*Interval` $\mapsto$ `Float`

■

`integrateAsymmetric`(I,J) computes $(\int I(x)\cdot J(x)\ dx)/|I|$.

`integrateSymmetric`$(I,J,simple)$ computes $(\int I(x)\cdot J(x)\ dx)/N(I,J)$

where $N(I,J) \stackrel{\text{def}}{=} \begin{cases} min(|I|,|J|) & \text{if } simple = \texttt{true} \\ max_a(\int I(x-a)\cdot J(x)\ dx) & \text{otherwise.} \end{cases}$

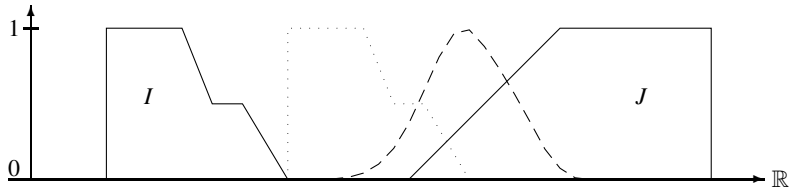
The next example shows an application of the symmetric `integrate` function. A fuzzy interval–interval relation `IIRMeets` is defined: Besides the two intervals, it takes the transformation functions F and S and integrates the interval $F(I)$ over $S(J)$. $F(I)$ should map the interval I to a finishing section of I and $S(J)$ should map the interval J to a starting section of J. The integration of $F(I)$ to $S(J)$ yields the final result.

Example 4.5 (Fuzzy Interval–Interval 'Meets' Relation) A possible definition for a fuzzy interval–interval meets relation is

```
IIRMeets(Interval I, Interval J,
         Interval->Interval F, Interval->Interval S) =
  if isEmpty(I)
     or isEmpty(J)
     or isInfinite(I,right)
     or isInfinite(J,left)
  then 0
  else integrateSymmetric(F(I),S(J),false)
```

■

The figure below shows the effect of the `IIRMeets` relation for suitable `F` and `S` operations. The dashed figure shows the result of `IIRMeets`$(I,J,\ldots)$ when the interval I is moved along the horizontal axis. The dotted figure shows the position of the interval I where `IIRMeets`$(I,J,\ldots)$ is maximal.



`IIRMeets` *for Fuzzy Intervals*

MPLL contains the very special purpose function `MaximizeOverlap` which is, so far, only needed for implementing the fuzzy interval–interval overlaps relation. The classical relation *I overlaps J* has two requirements:
1. a non-empty part I_1 of I must lie before J, and
2. another non-empty part I_2 of I must lie inside J.

Generalisation to fuzzy intervals encodes the first condition in the factor $1-D(I,E^+(J))$, where D is a `during` operator. $E^+(J)$ extends the rising part of J to the infinity. Therefore $D(I,E^+(J))$ measures the part of I which is after the front part of J. $1-D(I,E^+(J))$ then measures the part of I which is before the front part of J. This factor is multiplied with $D(I,J)$ which corresponds to the second condition. It measures to which

degree I is contained in J. The product is normalised with $\max_a((1-D(I_a,E^+(J)))\cdot D(I_a,J))$, where $I_a(x) \stackrel{\text{def}}{=} I(x-a)$. This corresponds to the maximal possible overlap when I is shifted along the horizontal axis. This guarantees that there is a position for I where I *overlaps* $J=1$. The normalization factor is computed with the function `MaximizeOverlap`

Definition 4.60 (`MaximizeOverlap`)
The function

`MaximizeOverlap`(I,J,EJ,D)
of type
`Interval*Interval*Interval*`
`(Interval*Interval`$\mapsto$`Float)`$\mapsto$`Float`
computes

$$\max_a((1-D(\texttt{shift}(I,a),EJ))\cdot D(\texttt{shift}(I,a),J))$$

■

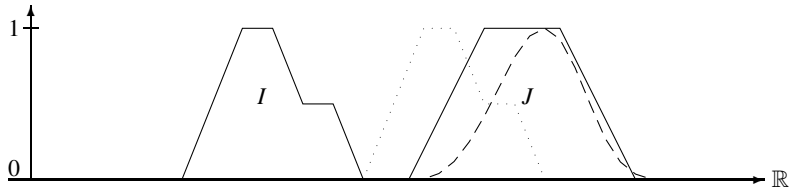
Notice that EJ can in principle be an arbitrary interval. For the encoding of the fuzzy overlaps relation, it should, however, be the extension of J to the infinity.

Example 4.6 *IIROverlaps*

```
IIROverlaps(Interval I, Interval J, Interval->Interval E,
            (Interval*Interval)->Float D) =
  case
    isEmpty(I) or isEmpty(J) or isInfinite(J,left) : 0,
    isInfinite(I,right) :
      float(point(I,left,support)<point(J,left,support)),
    isInfinite(J,right) :
      float(point(I,right,support)<point(J,left,support))
  else
    Let EJ = E(J) in
      (1 - D(I,EJ))*D(I,J) / MaximizeOverlap(I,J,EJ,D)
```

■

Example 4.7 (IIROverlaps for Fuzzy Intervals)
This example shows the result of the `IIROverlaps` relation where the standard `IIRDuring` operator is used (with the identity function as point–interval `during` operator).



Example: Overlaps Relation

The dashed line represents the result of the overlaps relation for an angle value A where the positive end of the interval I is moved to A. The dotted figure indicates the interval I moved to the position where `IIRoverlaps`(I,J) becomes maximal. ■

4.4.10. Circular Intervals

This type of interval is needed for fuzzy interval which pertain to finite circular intervals, such as those needed for representing fuzzy distributions over angular expressions.

Definition 4.61 (Construction)
The default method to construct a circular interval is to provide two coordinate values as `Angle` *data types, which represent the lower and upper limit of the interval.*

Alternative constructors are part of the standard library (see Def. 4.87, pp. 159, in section 4.6).

(1) `CircularInterval`$(Amin, Amax)$
`Angle*Angle` $\mapsto$ `CircularInterval`

(2) `CircularInterval`$(Amin, Amax, Acore, Asupport, Acentre)$
`Angle*Angle*Angle*Angle*Angle` $\mapsto$ `CircularInterval`

(3) `CircularInterval`$(Amin, Amax, Fvalue)$
`Angle*Angle*Float` $\mapsto$ `CircularInterval`

■

(1) accepts minimum and maximum of the interval specified as `Angle` values.

(2) accepts minimum and maximum of the interval specified as `Angle` values and the width of core and support, as well as the centre of the fuzzy distribution as `Angle` values. This results in a fuzzy distribution within the interval *Amin* and *Amax*, which starts with the fuzzy value 0 at $Acentre - (Asupport/2)$ and increases in a linear way to the fuzzy value 1 at $Acentre - (Acore/2)$. At $Acentre + (Acore/2)$ it starts to decrease in a linear way until it reaches the fuzzy value 0 again at $Acentre + (Asupport/2)$. This constructor is needed for generating the default intervals of cardinal directions, such as *"north"*, *"southeast"*, or *"left_of"*.
(3) operates analogous to (1) except that the interval has a constant value of $Fvalue$ over the entire range between *Amin* and *Amax*.

4.5. Basic Functions

MPLL includes a number of generic basic functions which are hard coded, either due to the complexity of computation, or because of performance issues. Currently, due to the lack of extensive testing of the prototype implementation in different scenarios, we cannot provide examples of functions which require hard-coding due to performance issues. Intuitively, however, such functions exist. The `List` data type, for example, cannot be coded in MPLL and, therefore, has to be hard-coded in its entirety since it involves polymorphism.

4.5.1. Transformations

The generic transformations, i.e. translation, rotation, and scaling, are part of the basic functions of MPLL. The mathematical formalisations of these functions have already been discussed in section 2.3.2.

Definition 4.62 (Translation)
MPLL provides the following function for linear translation in planar space.

(1) `translateCartesian`(C,Ax,Ay)
of type
`Configuration*Angle*Angle` $\mapsto$ `Configuration`

(2) `translateCartesian`(L,Ax,Ay)
of type
`Line*Angle*Angle` $\mapsto$ `Line`

(3) `translateCartesian`(R,Ax,Ay)
of type
`Polygon*Angle*Angle` $\mapsto$ `Polygon`

■

(1) modifies the position of the configuration C by a linear translation of the amount Ax into the direction of the x-axis and of the amount Ay into the direction of the y-axis. If either Ax or Ay contains 0 (or $NULL$), then a linear translation along the axis which contains a non-zero value is computed.
The following matrix translates the configuration C with coordinates $(C.x, C.y)$ to a new position $(C.x', C.y')$:

$$[C.x', C.y', 1] = [C.x, C.y, 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Ax & Ay & 1 \end{bmatrix}$$

(2) modifies the position of all points in L by a linear translation of the amount Ax into the direction of the x-axis and of the amount Ay into the direction of the y-axis. If either

Ax or *Ay* contains 0 (or *NULL*), then a linear translation along the axis which contains a non-zero value is computed.

(3) modifies the position of all points in *R* by a linear translation of the amount *Ax* into the direction of the x-axis and of the amount *Ay* into the direction of the y-axis. If either *Ax* or *Ay* contains 0 (or *NULL*), then a linear translation along the axis which contains a non-zero value is computed.

Definition 4.63 (Rotation)
MPLL provides the following function for rotation of spatial entities in planar space (i.e. rotation around the z-axis). Note that a positive angle A denotes clockwise rotation, whereas a negative angle denotes counter-clockwise rotation.

(1) `rotate`(*C*,*A*)
of type
`Configuration*Angle` $\mapsto$ `Configuration`

(2) `rotate`(*L*,*A*)
of type
`Line*Angle` $\mapsto$ `Line`

(3) `rotate`(*R*,*A*)
of type
`Polygon*Angle` $\mapsto$ `Line`

■

(1) rotates the position of *C* by the amount of the angle *A*, in the direction determined by the sign of *A*, around the origin of the reference system. Note that this function also modifies the orientation of *C* by the same amount.
The following matrix rotates a point $P = (P.x, P.y)$ around the z axis:

$$[P.x', P.y', 1] = [P.x, P.y, 1] \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that in two-dimensional scenes, only rotation around the z axis is possible. Otherwise, objects would leave the $x-y$ plane.

(2) rotates the positions of all points in *L* by the amount of the angle *A*, in the direction determined by the sign of *A*, around the origin of the reference system.

(3) rotates the positions of all points in *R* by the amount of the angle *A*, in the direction determined by the sign of *A*, around the origin of the reference system.

Definition 4.64 (Scaling)
MPLL provides the following functions to change the scale of spatial entities. Note that configurations, given that they have no spatial extent, cannot be scaled.

(1) `scale`(L,F)
of type
`Line*Float` $\mapsto$ `Line`
(2) `scale`(R,F)
of type
`Polygon*Float` $\mapsto$ `Polygon`

■

(1) modifies L by scaling it by the factor F from the origin of the current reference system.
The following matrix scales the line L consisting of the points $P_1,\ldots,P_n \in L$ with coordinates $(P_1.x,P_1.y)$, ..., $(P_n.x,P_n.y)$ by the factor F:

$$[P_i.x',P_i.y',1] = [P_i.x,P_i.y,1]\begin{bmatrix} F & 0 & 0 \\ 0 & F & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(2) modifies R by scaling it by the factor F from the origin of the current reference system. The same matrix as in version (1) applies, except that points $P_1,\ldots,P_n \in R$ are modified.

Definition 4.65 (Turning)
Turning pertains to the rotation of the orientation *of a configuration, not to a* rotation of its position*. MPLL provides the following function for changing the orientation of a configuration C.*

(1) `turn`(C,A)
of type
`Configuration*Angle` $\mapsto$ `Configuration`

■

(1) rotates the orientation of C by the amount of the angle A, in the direction determined by the sign of A. A positive angle denotes clockwise rotation, whereas a negative angle denotes counter-clockwise rotation. Note that this function is different from the `rotate` function (see Def.4.63).

4.5.2. Bearing

The bearing between spatial entities is one of the most important spatial relations in MPLL. Therefore, a number of different forms of this function are part of the MPLL basic functions, as well as the MPLL standard library.

Definition 4.66 (`bearing`)
MPLL provides the function `bearing` *in several versions, with configurations C, D, lines L, M, and polygons R, S.*

(1) `bearing`(C,D)
of type
`Configuration*Configuration` $\mapsto$ `Angle`

(2) `bearing`(C,L)
of type
`Configuration*Line` $\mapsto$ `CircularInterval`

(3) `bearing`(C,R)
of type
`Configuration*Polygon` $\mapsto$ `CircularInterval`

(4) `bearing`(L,C)
of type
`Line*Configuration` $\mapsto$ `Side`

(5) `bearing`(L,M)
of type
`Line*Line` $\mapsto$ `Side`

(6) `bearing`(L,R)
of type
`Line*Polygon` $\mapsto$ `CircularInterval`

(7) `bearing`(R,C)
of type
`Polygon*Configuration` $\mapsto$ `CircularInterval`

(8) `bearing`(R,L)
of type
`Polygon*Line` $\mapsto$ `CircularInterval`

(9) `bearing`(R,S)
of type
`Polygon*Polygon` $\mapsto$ `CircularInterval`

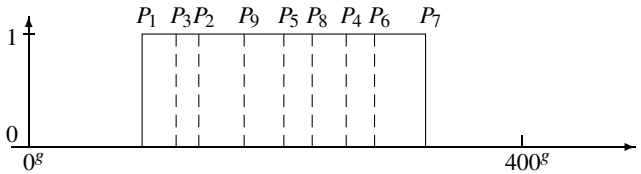
■

(1) returns the bearing from configuration C to configuration D (in radian) as follows: If $\triangle x = D.x - C.x$ and $\triangle y = D.y - C.y$ with $C = (C.x, C.y)$ and $D = (D.y, D.y)$, then

$$\theta_b(C,D) = \begin{cases} cos^{-1}(\frac{\triangle y}{\sqrt{\triangle x^2 + \triangle y^2}}) & \text{if } \triangle x \geq 0 \\ cos^{-1}(\frac{\triangle y}{\sqrt{\triangle x^2 + \triangle y^2}}) + \pi & \text{otherwise} \end{cases}$$

(2) returns the bearing from configuration C to line L as an interval of type `Circular-Interval`. The evaluation of this function includes the calculation of the bearings of all points of L and the subsequent construction of a crisp interval, where the fuzzy value is set to 1 for the bearing from C to the first point and all subsequent points of L and also between these bearings. This produces a fuzzy interval which consists of *exactly one crisp interval* of arbitrary extent. Note that this crisp interval *might* fully cover the

complete circular interval, but does not necessarily need to do so. An example of a circular interval computed for a line $L = (P_1, \ldots, P_9)$ is shown in the following diagram.



Fuzzy Point-to-Line Bearing as a Crisp Circular Interval

(3) returns the bearing from configuration C to region R as an interval of type `CircularInterval`. The evaluation of this function is done in two steps. First, a test for inclusion of C in R is conducted. If this test is successful, an interval consisting of a constant value of 1 is returned. If the test is not successful, the bearings of C to all points and segments of R are computed, in the same way as in version (2). This produces a fuzzy interval which consists of *exactly one crisp interval* of arbitrary extent. Note that this crisp interval *might* fully cover the complete circular interval, but does not necessarily need to do so.

(4) returns the bearing from a line L to a configuration C as a binary value of type `Side` (i.e. *left* or *right*). To this aim, the nearest line segment is determined by calculating the two nearest points of L. This line segment and the configuration C are then tested for either being a left turn or a right turn, which is returned as the result. Note that this requires the line L to be available in normalised form (which is the default, see Def. 4.25).

(5) returns the bearing from a line L to a line M as a binary value of type `Side` (i.e. *left* or *right*). If the two lines intersect, then this relation cannot be computed and returns an error.

(6) returns the bearing from a line L to a polygon R as a binary value of type `Side` (i.e. *left* or *right*). If the line and the polygon intersect, then this relation cannot be computed and returns an error.

(7) returns the bearing from a region R to a configuration C as an interval of type `CircularInterval`. Similar to version (3), the evaluation of this function is done in two steps. First, a test for inclusion of C in R is conducted. If this test is successful, an interval consisting of a constant value of 1 is returned. If the test is not successful, the bearings of C to all points and segments of R are computed, in the same way as in version (2). This produces a fuzzy interval which consists of *exactly one crisp interval* of arbitrary extent. Note that this crisp interval *might* fully cover the complete circular interval, but does not necessarily need to do so.

(8) this version operates analogous to (7) with the exception, that no test for inclusion is conducted, but a test for intersection with the line L. If the two features intersect, an

interval consisting of a constant value of 1 is returned. If not, a computation similar to (2) is conducted.

(9) this version operates analogous to (7) with the exception, that no test for inclusion is conducted, but a test for overlap with the polygon S. If the two polygons overlap, an interval consisting of a constant value of 1 is returned. If not, a computation similar to (3) is conducted.

Definition 4.67 (`side`)
The `side` *function is a special form of bearing from line entities, based on the fact that lines in MPLL are always defined with an inherent linear direction, which in turn determines the sides of the line entity.*

(1) `side`(L,C)
of type
`Line*Configuration` $\mapsto$ `Side`

(2) `side`(L,R)
of type
`Line*Polygon` $\mapsto$ `Side`

■

(1) returns the side of L on which C is located as enumeration type `Side`. Note that this depends on the order of points in L.

(2) returns the side of L on which R is located as enumeration type `Side`, provided, that $\nexists p : p \in L \wedge p \in R$. Note that this depends on the order of points in L.

4.5.3. Construction of Points

Several spatial scenarios require the construction of point entities by means of processing other spatial entities and/or relations. This section contains some important examples. In course of the use of MPLL this part of the implementation will be expanded as necessary constructs are added.

Definition 4.68 (`centerOfMass`)
MPLL provides the function `centerOfMass` *in several versions, with configurations C, D.*

(1) `centerOfMass`(R)
of type
`Polygon` $\mapsto$ `Point`

■

(1) computes the centre of mass of the polygon R. The centre of the mass (centroid) of a polygon is the arithmetical mean of the sum of the centres of masses of all triangles obtained by a triangulation. The latter can be computed fairly easy. It is the intersection of its medians which are the line segments of a vertex to the centroid of the opposite face.

Let the triangle $\mathcal{T}$ be given by three vertices $\vec{a} := (x_a, y_a)$, $\vec{b} := (x_b, y_b)$, and $\vec{c} := (x_c, y_c)$. As the centroid of a d-simplex divides each median in a $d : 1$ ratio (as seen from the vertex) and thus in a $2 : 1$ ratio for a triangle. The function μ_∇ that determines the centroid of a d-simplex $\mathcal{S}$ can be defined:

$$\mu_\nabla(S) := \frac{1}{d+1} \cdot \sum_{\vec{x} \in \mathcal{S}} \vec{x}$$

Thus for the 2-dimensional case of a triangle $\mathcal{T}$:

$$\mu_\nabla(T) = \frac{1}{3} \cdot \left(\vec{a} + \vec{b} + \vec{c} \right)$$

Now Let Δ be the set of triangles of an arbitrary triangulation of polygon R. As mentioned above the arithmetic mean of its triangles define the centre of mass:

$$\mu(R) := \frac{1}{|\Delta|} \cdot \sum_{\delta \in \Delta} \mu_\nabla(\delta)$$

Note that the same function can be applied to the arbitrary decomposition of a polytope into tetrahedra.

Definition 4.69 (`closestPoint`)
MPLL provides the function `closestPoint` *in several versions, with configurations C, D.*

(1) `closestPoint`(L, C)
of type
`Line*Configuration` $\mapsto$ `Point`

(2) `closestPoint`(R, C)
of type
`Polygon*Configuration` $\mapsto$ `Point`

■

(1) returns a single point $P \in L$, whose geometrical distance to C is minimal.

(2) returns a single point $P \in R$, whose geometrical distance to C is minimal.

Definition 4.70 (`intersectionPoint`)
MPLL provides the function `intersectionPoint`*, with lines L, M.*

(1) `intersectionPoint`(L, M)
of type
`Line*Line` $\mapsto$ `Point`

■

(1) returns the coordinates of a point p in form of a `Point`, if

$$\exists p \in L : p \in M$$

Otherwise, an exception is thrown.

4.5.4. Construction of Lines

The construction of line entities (lines and line segments) by means of processing other spatial entities and/or relations is also required in some situations. This section contains some examples. In course of the use of MPLL this part of the implementation will be expanded as necessary constructs are added.

Definition 4.71 (`closestSegment`)
MPLL provides the function `closestSegment` *in several versions, with configurations C, D.*

(1) `closestSegment`(L,C)
of type
`Line*Configuration` $\mapsto$ `Line`

(2) `closestSegment`(R,C)
of type
`Polygon*Configuration` $\mapsto$ `Line`

■

(1) returns a line consisting of one segment, i.e. consisting of two points $P1$ and $P2$, with $P1, P2 \in L$ and $P1, P2$ belonging to the same segment, and with the distance from $P1$ and $P2$ to C being minimal.

Let L be a sequence of line segments and p a point. There is a function that obtains the distance between a point and a line segment. It can be used to collect all closest segments of a line sequence. Note that there are several segments that may have the property to be nearest neighbour to a given point. The candidates can be defined as followes:

$$closestSegment\,(p,S) := \{s \in S | dist(p,s) \leq dist(p,x)\}$$

Note that it does not suffice to consider the distances between p and the two defining point of a line segment.

(2) returns a line consisting of one segment, i.e. consisting of two points $P1$ and $P2$, with $P1, P2 \in R$ and $P1, P2$ belonging to the same segment, and with the distance from $P1$ and $P2$ to C being minimal. The formal definition is analogous to (1).

4.5.5. Other Predicates

This section includes some predicates which have not been included in section 4.4 for clarity. In the future, entries in this section might be moved to other parts in the documentation as necessary. In course of the use of MPLL this part of the implementation will also be expanded as necessary constructs are added.

Definition 4.72 (`intersection`)
MPLL provides the function `intersection`*, with lines L, M.*

(1) `intersection`(L,M)
of type
`Line*Line` $\mapsto$ `Bool`

■

(1) returns `true` if

$$\exists p \in L : p \in M$$

Otherwise, it returns `false`.

4.6. The MPLL Standard Library – Types

The MPLL Standard Library contains a number of constructors and other type related functions which did not need to be included in the hard-coded MPLL kernel. These functions are defined as MPLL constructs and can, therefore, be easily extended and/or modified to suit specific purposes.

4.6.1. Naming Conventions

The definitions in the standard library adhere to the default naming conventions laid out in section 4.2.2.

4.6.2. Predefined Constants

Much like C++ macro definitions, MPLL uses some constants and internal defaults. However, unlike macro definitions, these MPLL constants can be overwritten, depeding on the global control parameter `MPLL::overwrite`.

This section lists the constants which are used subsequently in the standard library. Note that while, technically, any constant can be modified easily, modifying some constants which control internal processes could lead to malfunction. In the following list, these critical constants have an index below (50) and should only be modified with care.

Definition 4.73 (Predefined Constants)

(1) `Real` $= 0$ $\mapsto$ `Integer`
(2) `Grd` $= 1$ $\mapsto$ `Integer`
(3) `Deg` $= 2$ $\mapsto$ `Integer`
(4) `Rad` $= 3$ $\mapsto$ `Integer`
(50) `Pi` $= 3.141592653$ $\mapsto$ `Float`
(51) `defCore` $= 25$ $\mapsto$ `Integer`
(52) `defSupport` $= 50$ $\mapsto$ `Integer`

■

(1) is the constant identifier for real values. It is assigned the `Integer` value 0.

(2) is the constant identifier for grad values. It is assigned the `Integer` value 1.

(3) is the constant identifier for degree values. It is assigned the `Integer` value 2.

(4) is the constant identifier for radian values. It is assigned the `Integer` value 3.

(50) is the constant π. It is assigned the `Float` value 3.141592653. If, for some reason, this constant needs to be defined using higher precision, the constant can be overwritten – provided that the global control parameter `MPLL::overwrite` is set accordingly.

(51) and (52) are the default values for core and support angles used in defining fuzzy cardinal direction intervals. See Fig. 2.26 on page 76 for an illustration of how to use these values.

Definition 4.74 (Default Moduli) *(Constant)*
The default moduli for real, grad, degree, and radian angles are defined as follows.

```
        defMod(type) =
(100)   case  (type == Real) :  0,
(101)         (type == Grd) :   400,
(102)         (type == Deg) :   360,
(103)         (type == Rad) :   (2 * Pi),
        else                    0
of type
        Integer ↦ Float
```

■

Default moduli are necessary for `Angle` types, in order to define the range of possible values. This property is also used to distinguish *state* from *action*. There exists, for example, no orientation of 480° (this value would be treated modulo 360°, i.e. 120°), but a valid action would be to "turn 480° left" (resulting in more than a full turn).

(100) The default modulus for angles holding real values is 0. There exists no default minimum or maximum, except those imposed by the implementation restrictions of the data types (float or double) holding the real value.

(101) The default modulus for angles holding grad values is 400^g. The default minimum is therefore -400^g, and the default maximum is 400^g.

(102) The default modulus for angles holding degree values is 360°. The default minimum is therefore $-360°$, and the default maximum is 360°.

(103) The default modulus for angles holding radian values is 2π. The default minimum is therefore -2π, and the default maximum is 2π.

Definition 4.75 (Allocentric Cardinal Direction) *(Constant)*
The variable Dir must be one of the valid enumeration types of type `CarDir` *(see*

Def. 4.17) on page 110). The allocentric cardinal directions are defined as follows, all values are given in grad.

```
        carDir(Dir) =
(200)   case  (Dir == north):      0,
(201)         (Dir == northeast):  50,
(202)         (Dir == east):       100,
(203)         (Dir == southeast):  150,
(204)         (Dir == south):      200,
(205)         (Dir == southwest):  250,
(206)         (Dir == west):       300,
(207)         (Dir == northwest):  350,
        else                       0
of type
        CarDir ↦ Float
```

```
        carDir(Dir) =
(208)   case  (Dir == N):   0,
(209)         (Dir == NE):  50,
(210)         (Dir == E):   100,
(211)         (Dir == SE):  150,
(212)         (Dir == S):   200,
(213)         (Dir == SW):  250,
(214)         (Dir == W):   300,
(215)         (Dir == NW):  350,
        else                0
of type
        CarDir ↦ Float
```

■

(200) to (215) contain the cardinal direction constants in grad. These can be overwritten, depending on the global control parameter `MPLL::overwrite`.

Definition 4.76 (Egocentric Cardinal Direction) *(Constant)*
The variable Dir must be one of the valid enumeration types of type `EgoDir`*, or* `EgoDirD` *(see Def. 4.17) on page 110).*

```
        carDir(Dir) =
(300)   case  (Dir == front):          0,
(301)         (Dir == right_front):    50,
(302)         (Dir == right):          100,
(303)         (Dir == right_back):     150,
(304)         (Dir == back):           200,
(305)         (Dir == left_back):      250,
(306)         (Dir == left):           300,
(307)         (Dir == left_front):     350,
        else                           0
of type
        EgoDir ↦ Float

        carDir(Dir) =
(400)   case  (Dir == behind):         0,
(401)         (Dir == right_of):       100,
(402)         (Dir == in_front_of):    200,
(403)         (Dir == left_of):        300,
        else                           0
of type
        EgoDirD ↦ Float
```

■

The egocentric cardinal directions are defined here. All values are given in grad. They can be overwritten, depending on the global control parameter `MPLL::overwrite`.

(300) to (307) contain the egocentric cardinal direction constants in grad.

(400) to (403) contain the egocentric cardinal direction constants for deictic settings in grad.

Definition 4.77 (Fuzzy Allocentric Cardinal Direction) *(Constant)*
The intervals are defined using the default values for core and support (see Def. 4.73) and they are based on the direction constants listed in Def. 4.75.

```
(500)   carDirF(Dir) =
        CircularInterval(carDir(Dir), defCore, defSupport)
        of type
        CarDir ↦ CircularInterval
```

■

These are not constants per se, but predefined intervals representing fuzzy allocentric cardinal direction. Nonetheless, they best fit into the category of predefined constants.

(500) returns a fuzzy circular interval which represents the direction *Dir*, using the default constants for core and support. For an illustration of such an interval see Fig. 2.26

on page 76. See also Def. 4.87 on page 159 for the definition of the circular interval constructor.

Definition 4.78 (Euclidean Distance) *(Constant)*
This categorisation of qualitative distances was inspired by the proposal by Clementini, Di Felice, and Hernández [77]. The actual numeric values serve only to illustrate the use of distances within MPLL. Most likely, for individual applications, this set has to be carefully adjusted to reflect the user preferences and context.

```
        distance(Dist) =
(600)   case  (Dist == very_close):      10,
(601)         (Dist == close):           30,
(602)         (Dist == commensurate):    60,
(603)         (Dist == far):             100,
(604)         (Dist == very_far):        150,
        else                             0
of type
        Distance ↦ Float
```

■

The numeric values adhere to the convention that subsequent intervals (distances farther away) must be larger than the previous intervals, i.e. given distances d_i, $i = (1,\ldots,n)$, then $\forall d_j, j = (1,\ldots,n-1) : |d_j| \leq |d_{j+1}|$.

Other conventions might be different. It is, for example, also possible to require all subsequent intervals to be larger than the sum of the previous intervals: given distances d_i, $i = (1,\ldots,n)$, then $\forall d_i, i = (1,\ldots,n) : \sum_{j=1}^{i-1} |d_j| \leq |d_i|$.

(600 – 604) The table above has to be read like this: distances are given as the length of the intervals for each quality, i.e. `very_close` pertains to the interval $[0,10]$, `close` pertains to the interval $[10,30]$, `commensurate` pertains to the interval $[30,60]$, and so on.

4.6.3. Angles

An angle can be constructed in different ways. In addition to the only hard-coded variant (see Def. 4.19), there are a number of alternatives available. The differences exist mainly because of different angular values (grad, degree, and radian) and because of the optional specification of minimum and maximum values.

Definition 4.79 (Construction)
The MPLL Standard Library provides the following alternative constructors for angles:

(1) `Angle()` =
`Angle(0, −defMod(Grd), defMod(Grd))`
of type
↦ `Angle`

(2) $\texttt{AngleReal}(\mathit{Fangle}) =$
$\texttt{Angle}(\mathit{Fangle}, -\texttt{defMod}(\mathit{Real}), \texttt{defMod}(\mathit{Real}))$
of type
$\texttt{Float} \mapsto \texttt{Angle}$

(3) $\texttt{AngleGrd}(\mathit{Fangle}) =$
$\texttt{Angle}(\mathit{Fangle}, -\texttt{defMod}(\mathit{Grd}), \texttt{defMod}(\mathit{Grd}))$
of type
$\texttt{Float} \mapsto \texttt{Angle}$

(4) $\texttt{AngleDeg}(\mathit{Fangle}) =$
$\texttt{Angle}(\texttt{degToGrd}(\mathit{Fangle}), -\texttt{defMod}(\mathit{Grd}), \texttt{defMod}(\mathit{Grd}))$
of type
$\texttt{Float} \mapsto \texttt{Angle}$

(5) $\texttt{AngleRad}(\mathit{Fangle}) =$
$\texttt{Angle}(\texttt{radToGrd}(\mathit{Fangle}), -\texttt{defMod}(\mathit{Grd}), \texttt{defMod}(\mathit{Grd}))$
of type
$\texttt{Float} \mapsto \texttt{Angle}$

(6) $\texttt{AngleReal}(\mathit{Fangle}, \mathit{Fmod}) =$
$\texttt{Angle}(\mathit{Fangle}, -\mathit{Fmod}, \mathit{Fmod})$
of type
$\texttt{Float} * \texttt{Float} \mapsto \texttt{Angle}$

(7) $\texttt{AngleGrd}(\mathit{Fangle}, \mathit{Fmod}) =$
$\texttt{Angle}(\mathit{Fangle}, -\mathit{Fmod}, \mathit{Fmod})$
of type
$\texttt{Float} * \texttt{Float} \mapsto \texttt{Angle}$

(8) $\texttt{AngleDeg}(\mathit{Fangle}, \mathit{Fmod}) =$
$\texttt{Angle}(\texttt{degToGrd}(\mathit{Fangle}), -\texttt{degToGrd}(\mathit{Fmod}), \texttt{degToGrd}(\mathit{Fmod}))$
of type
$\texttt{Float} * \texttt{Float} \mapsto \texttt{Angle}$

(9) $\texttt{AngleRad}(\mathit{Fangle}, \mathit{Fmod}) =$
$\texttt{Angle}(\texttt{radToGrd}(\mathit{Fangle}), -\texttt{radToGrd}(\mathit{Fmod}), \texttt{radToGrd}(\mathit{Fmod}))$
of type
$\texttt{Float} * \texttt{Float} \mapsto \texttt{Angle}$

(10) $\texttt{Angle}(C, D, E) =$
$\texttt{Angle}(\texttt{bearing}(D, E) - \texttt{bearing}(D, C))$
of type
$\texttt{Configuration} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Angle}$

■

Version (1) is the default constructor. By default, the value of the angle is 0 and it is treated modulo 400^g. Therefore, it can hold values in the interval $]-400^g, 400^g[$.

Version (2) accepts a negative or positive real value as a `Float` or an `Integer` value. By default, the angle is not treated modulo a certain factor. Therefore, it can hold any real values. This version is intended for use with special kinds of coordinates, such as screen coordinates.

Version (3) accepts a negative or positive grad value as a `Float` or an `Integer` value. By default, the angle is treated modulo 400^g. Therefore, it can hold values in the interval $]-400^g, 400^g[$.

The constructors (4) and (5) operate analogous to (2) with the exception that values are explicitly given as degree (4) and radian (5), and the respective modulus is added. The modulus is always given as grad, but results in the intervals $]-360^\circ, 360^\circ[$ (3) and $]-2\pi, 2\pi[$ (4).

Version (6) accepts a negative or positive real value as a `Float` or an `Integer` value. The modulus is explicitly given by *Fmod*. Therefore, it can hold any real values in the Interval $[-Fmod, Fmod[$.

The constructors (7) to (9) operate analogous to (2) with the exception that both values are explicitly given as grad (7), degree (8), and radian (9). The modulus is explicitly given by *Fmod* and contains a grad (7), degree (8), or radian (9) value. The latter two are converted to grad before calling `Angle`.

The constructor (10) accepts three configurations *C*, *D*, and *E* as parameters and constructs an angle which is defined by the difference between the bearing between *D* and *E* and the bearing between *D* and *C*.

Definition 4.80 (Predicates)
MPLL provides the following predicates to ascertain the properties of an angle A.

(1) $\texttt{isPositive}(A) = \texttt{not}(\texttt{isNegative}(A))$ $\quad \texttt{Angle} \mapsto \texttt{Bool}$
(2) $\texttt{isRightTurn}(A) = \texttt{not}(\texttt{isNegative}(A))$ $\quad \texttt{Angle} \mapsto \texttt{Bool}$
(3) $\texttt{isLeftTurn}(A) = \texttt{isNegative}(A)$ $\quad \texttt{Angle} \mapsto \texttt{Bool}$
(4) $\texttt{minMax}(A) = \texttt{Interval}(\texttt{min}(A), \texttt{max}(A))$ $\quad \texttt{Angle} \mapsto \texttt{Interval}$

■

Predicate (1) is defined using the basic predicate `isNegative`. It returns `true` if `isNegative` returns `false`.

Predicates (2) and (3) are defined using the basic predicate `isNegative`, whereas positive angles represent right turns and negative angles represent left turns.

(4) returns the interval for valid angles of this type, specified by the interval between $\texttt{min}(A)$ and $\texttt{max}(A)$.

4.6.4. Points

Apart from the default constructor, which can be called without any parameters, there exist three alternative constructors.

Definition 4.81 (Construction)
The MPLL Standard Library provides the following alternative constructors for points:

(1) `Point()` = `Point(Angle(),Angle())`
of type
$\mapsto$ `Point`

(2) `Point`(Ax,Ay) = `Point`(Ax,Ay)
of type
`Angle`$*$`Angle` $\mapsto$ `Point`

(3) `Point`(Fx,Fy) = `Point(Angle`(Fx)`,Angle`(Fy)`)`
of type
`Float`$*$`Float` $\mapsto$ `Point`

■

(1) is the default constructor which creates a new `Point` positioned at the origin (0,0).

(2) accepts a pair of coordinates specified as `Angle` values.

(3) accepts a pair of coordinates specified as `Float` or `Integer` values. These are subsequently used to initialise the `Angle` values of the newly created `Point`.

4.6.5. Configurations

A number of constructors can be derived from the different possible combinations of specifying either the orientation or the position. Additionally, there exist variations from the individual form of the parameters (`Angle`, `Float`, or `Point`).

Definition 4.82 (Construction)
The MPLL Standard Library provides the following alternative constructors for configurations:

(1) `Configuration()` =
`Configuration(Angle(),Angle(),Angle(),`$false$`)`
of type
$\mapsto$ `Configuration`

(2) `Configuration`(A,Fx,Fy) =
`Configuration`$(A$`,Angle`(Fx)`,Angle`(Fy)`,`$true$`)`
of type
`Angle`$*$`Float`$*$`Float` $\mapsto$ `Configuration`

(3) `Configuration`(A,Ax,Ay) =
`Configuration`$(A,Ax,Ay,true)$
of type
`Angle`$*$`Angle`$*$`Angle` $\mapsto$ `Configuration`

(4) `Configuration`$(A) =$
`Configuration`$(A,$`Angle()`$,$`Angle()`$,true)$
of type
`Angle` $\mapsto$ `Configuration`

(5) `Configuration`$(A,P) =$
`Configuration`$(A,$`getX`$(P),$`getY`$(P),true)$
of type
`Angle*Point` $\mapsto$ `Configuration`

(6) `Configuration`$(P) =$
`Configuration`$($`Angle()`$,$`getX`$(P),$`getY`$(P),true)$
of type
`Point` $\mapsto$ `Configuration`

(7) `Configuration`$(Ax,Ay) =$
`Configuration`$($`Angle()`$,Ax,Ay,false)$
of type
`Angle*Angle` $\mapsto$ `Configuration`

(8) `Configuration`$(Fx,Fy) =$
`Configuration`$($`Angle()`$,Fx,Fy,false)$
of type
`Float*Float` $\mapsto$ `Configuration`

■

(1) is the default constructor. The position is set to default (the origin at $(0,0)$), the orientation defaults to 0. `hasOrientation` is set to `false`.

(2) accepts an orientation given as an `Angle` value and a pair of coordinates (two dimensions) specified as two `Float` or `Integer` values. `hasOrientation` is set to `true`.

(3) accepts an orientation given as an `Angle` value and a pair of coordinates (two dimensions) also specified as `Angle` values. `hasOrientation` is set to `true`.

(4) accepts an orientation given as an `Angle` value. The position is set to default (the origin at $(0,0)$). `hasOrientation` is set to `true`.

(5) accepts an orientation given as an `Angle` value and a pair of coordinates (two dimensions) specified as a `Point` value. `hasOrientation` is set to `true`.

(6) accepts a pair of coordinates (two dimensions) specified as a `Point` value. The orientation defaults to 0. `hasOrientation` is set to `false`.

(7) accepts a pair of coordinates (two dimensions) specified as two `Angle` values. The orientation defaults to 0. `hasOrientation` is set to `false`.

(8) accepts a pair of coordinates (two dimensions) specified as two `Float` or `Integer` values. The orientation defaults to 0. `hasOrientation` is set to `false`.

4.6.6. Lines

An alternative constructor accepts a sequence of points in form of pairs of individual coordinates (of type `Angle`).

Definition 4.83 (Construction)
The MPLL Standard Library provides the following alternative constructor for lines:

(1) $\texttt{Line}(Ax1, Ay1, Ax2, Ay2, \ldots, Axn, Ayn) =$
$\texttt{Line}(\texttt{Point}(Ax1, Ay1), \texttt{Point}(Ax2, Ay2), \ldots, \texttt{Point}(Axn, Ayn))$
of type
$\texttt{Angle} * \texttt{Angle} * \ldots * \texttt{Angle} \mapsto \texttt{Line}$

■

(1) accepts a sequence of n points in form of n pairs of coordinates of type `Angle`.

Definition 4.84 (Predicates)
MPLL provides the following predicates to ascertain the properties of lines L:

(1) $\texttt{upperLeftBB}(L) =$
$\texttt{Point}(\texttt{Angle}(\texttt{xMin}(P)), \texttt{Angle}(\texttt{yMin}(P)))$
of type
$\texttt{Line} \mapsto \texttt{Point}$

(2) $\texttt{lowerRightBB}(L) =$
$\texttt{Point}(\texttt{Angle}(\texttt{xMax}(P)), \texttt{Angle}(\texttt{yMax}(P)))$
of type
$\texttt{Line} \mapsto \texttt{Point}$

■

(1) returns the upper left point of the minimum bounding box of the line in form of a `Point`.

(2) returns the lower right point of the minimum bounding box of the line in form of a `Point`.

4.6.7. Polygons

An alternative constructor accepts a sequence of points in form of pairs of individual coordinates (of type `Angle`).

Definition 4.85 (Construction)
The MPLL Standard Library provides the following alternative constructor for polygons:

(1) $\texttt{Polygon}(Ax1, Ay1, Ax2, Ay2, \ldots, Axn, Ayn) =$
$\texttt{Polygon}(\texttt{Point}(Ax1, Ay1), \texttt{Point}(Ax2, Ay2), \ldots, \texttt{Point}(Axn, Ayn))$
of type
$\texttt{Angle} * \texttt{Angle} * \ldots * \texttt{Angle} \mapsto \texttt{Polygon}$

■

(1) accepts a sequence of n points in form of n pairs of coordinates of type `Angle`.

Definition 4.86 (Predicates)
MPLL provides the following predicates to ascertain the properties of polygons P:

(1) `upperLeftBB`$(P) =$
`Point`(`Angle`(`xMin`(P)),`Angle`(`yMin`(P)))
of type
`Polygon` $\mapsto$ `Point`

(2) `lowerRightBB`$(P) =$
`Point`(`Angle`(`xMax`(P)),`Angle`(`yMax`(P)))
of type
`Polygon` $\mapsto$ `Point`

■

(1) returns the upper left point of the minimum bounding box of the polygon in form of a `Point`.

(2) returns the lower right point of the minimum bounding box of the polygon in form of a `Point`.

4.6.8. Circular Intervals

Definition 4.87 (Construction)
The standard library features a default constructor and some alternatives.

(1) `CircularInterval`()
`CircularInterval`(`Angle`(0),`defMod`(`Grd`))
of type
$\mapsto$ `CircularInterval`

(2) `CircularInterval`($Fvalue$)
`CircularInterval`(`Angle`(0),`defMod`(`Grd`),$Fvalue$)
of type
`Float` $\mapsto$ `CircularInterval`

(3) `CircularInterval`(A)
`CircularInterval`(`Angle`(0),`defMod`(`Grd`),$defCore$,$defSupport$,A)
of type
`Angle` $\mapsto$ `CircularInterval`

■

(1) creates a default (empty) circular interval with a default minimum of 0 and a default maximum of `defMod`(`Grd`) as specified by the standard library (400^g) or overridden by the user.

(2) creates an interval which consists of only one fuzzy value specified by $Fvalue$ for the entire interval, which has the same extensions as the default interval described in (1).

(3) creates an interval using the default values for *Amax*, $defCore$, and $defSupport$ in order to construct a default circular interval for the angle *A*.

4.7. The MPLL Standard Library – Functions

The standard library contains a number of composite spatial functions which did not need to be included in the hard-coded MPLL kernel. These functions are defined as MPLL constructs and can therefore be easily extended and/or modified to suit specific purposes.

4.7.1. Transformations

This section provides a range of overloaded versions of generic transformations, i.e. translation, rotation, and scaling.

Definition 4.88 (Translation)
MPLL provides the following functions for linear translation in planar space.

(1) $\texttt{translatePolar}(C, Adir, Adist) =$
$\texttt{translateCartesian}(C, sin(Adir) * Adist, cos(Adir) * Adist)$
of type
$\texttt{Configuration} * \texttt{Angle} * \texttt{Angle} \mapsto \texttt{Configuration}$

(2) $\texttt{moveX}(C, Adist) =$
$\texttt{translateCartesian}(C, Adist, \texttt{Angle}(0))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(3) $\texttt{moveY}(C, Adist) =$
$\texttt{translateCartesian}(C, \texttt{Angle}(0), Adist)$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(4) $\texttt{move}(C, Ax, Ay) =$
$\texttt{translateCartesian}(C, Ax, Ay)$
of type
$\texttt{Configuration} * \texttt{Angle} * \texttt{Angle} \mapsto \texttt{Configuration}$

(5) $\texttt{move}(C, D, Adist) =$
$\texttt{translatePolar}(C, \texttt{bearing}(C, D), Adist)$
of type
$\texttt{Configuration} * \texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(6) $\texttt{move}(C, Dir, Adist) =$
$\texttt{translatePolar}(C, \texttt{carDir}(Dir), Adist)$
of type
$\texttt{Configuration} * \texttt{CarDir} * \texttt{Angle} \mapsto \texttt{Configuration}$

(7) `moveTo`$(C,D) =$
`Configuration(orientation(`C`),point(`D`))`
of type
`Configuration*Configuration` $\mapsto$ `Configuration`

(8) `moveTo`$(C,P) =$
`Configuration(orientation(`C`),`P`)`
of type
`Configuration*Point` $\mapsto$ `Configuration`

(9) `move`$(C,Dir,Adist) =$
`translatePolar(`C`,carDir(`Dir`),`$Adist$`)`
of type
`Configuration*EgoDir*Angle` $\mapsto$ `Configuration`

■

(1) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction of the angle *Adir*. This function uses the basic function `translateCartesian` and the respective trigonometric functions of sine and cosine.

(2) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction of the x-axis.

(3) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction of the y-axis.

(4) modifies the position of the configuration C by a linear translation of the amount Ax into the direction of the x-axis and the amount Ay into the direction of the y-axis.

(5) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction of the bearing from C to D.

(6) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction given by the enumeration type *Dir*. *Dir* must be a valid enumeration type of type `CarDir`.

(7) modifies the position of the configuration C by a linear translation to the position of the configuration D.

(8) modifies the position of the configuration C by a linear translation to the position of the point P.

(9) modifies the position of the configuration C by a linear translation of the amount *Adist* into the direction given by the enumeration type *Dir*. Note that *Dir* must be a valid enumeration type of type `EgoDir`. This operation is only possible if C features an orientation.

Definition 4.89 (Rotation)
MPLL provides the following function for rotation of spatial entities in planar space,

i.e. rotation around the z-axis (see section 2.3.2 for details). Note that a positive angle A denotes clockwise rotation, whereas a negative angle denotes counter-clockwise rotation.

(1) `rotateLeft`(C,A)
`rotate`$(C,-$`abs`$(A))$
of type
`Configuration*Angle` $\mapsto$ `Configuration`

(2) `rotateCCW`(C,A)
`rotate`$(C,-$`abs`$(A))$
of type
`Configuration*Angle` $\mapsto$ `Configuration`

(3) `rotateRight`(C,A)
`rotate`$(C,$`abs`$(A))$
of type
`Configuration*Angle` $\mapsto$ `Configuration`

(4) `rotateCW`(C,A)
`rotate`$(C,$`abs`$(A))$
of type
`Configuration*Angle` $\mapsto$ `Configuration`

(5) `rotateLeft`(L,A)
`rotate`$(L,-$`abs`$(A))$
of type
`Line*Angle` $\mapsto$ `Line`

(6) `rotateCCW`(L,A)
`rotate`$(L,-$`abs`$(A))$
of type
`Line*Angle` $\mapsto$ `Line`

(7) `rotateRight`(L,A)
`rotate`$(L,$`abs`$(A))$
of type
`Line*Angle` $\mapsto$ `Line`

(8) `rotateCW`(L,A)
`rotate`$(L,$`abs`$(A))$
of type
`Line*Angle` $\mapsto$ `Line`

(9) `rotateLeft`(R,A)
`rotate`$(R,-$`abs`$(A))$
of type
`Polygon*Angle` $\mapsto$ `Polygon`

(10) `rotateCCW`(R,A)
`rotate`$(R,-$`abs`$(A))$
of type
`Polygon*Angle` $\mapsto$ `Polygon`

(11) `rotateRight`(R,A)
`rotate`$(R,$`abs`$(A))$
of type
`Polygon*Angle` $\mapsto$ `Polygon`

(12) `rotateCW`(R,A)
`rotate`$(R,$`abs`$(A))$
of type
`Polygon*Angle` $\mapsto$ `Polygon`

■

(1), (2) rotate the orientation of C by the amount of the angle A to the left (or counter-clockwise).

(3), (4) rotate the orientation of C by the amount of the angle A to the right (or clockwise).

(5), (6) rotate the the points of the line L by the amount of the angle A to the left (or counter-clockwise) around the origin of the reference system. Note that for rotation around any other point translation before and after rotation is necessary.

(7), (8) rotate the the points of the line L by the amount of the angle A to the right (or clockwise) around the origin of the reference system. Note that for rotation around any other point translation before and after rotation is necessary.

(9), (10) rotate the the points of the polygon R by the amount of the angle A to the left (or counter-clockwise) around the origin of the reference system. Note that for rotation around any other point translation before and after rotation is necessary.

(11), (12) rotate the the points of the polygon R by the amount of the angle A to the right (or clockwise) around the origin of the reference system. Note that for rotation around any other point translation before and after rotation is necessary.

Definition 4.90 (Scaling)
MPLL provides the following functions to change the scale of spatial entities:

(1) `scale`(L,F)
of type
`Line*Float` $\mapsto$ `Line`

(2) `scale`(R,F)
of type
`Polygon*Float` $\mapsto$ `Polygon`

■

(1) modifies L by scaling it by the factor F from the origin of the current reference system.

(2) modifies R by scaling it by the factor F from the origin of the current reference system.

Definition 4.91 (Turning)
MPLL provides the following function for changing the orientation of a configuration C.

(1) $\texttt{turnLeft}(C,A) =$
$\texttt{turn}(C, -\texttt{abs}(A))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(2) $\texttt{turnCCW}(C,A) =$
$\texttt{turn}(C, -\texttt{abs}(A))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(3) $\texttt{turnRight}(C,A) =$
$\texttt{turn}(C, \texttt{abs}(A))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(4) $\texttt{turnCW}(C,A) =$
$\texttt{turn}(C, \texttt{abs}(A))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(5) $\texttt{turnTo}(C,A) =$
$\texttt{Configuration}(A, \texttt{point}(C))$
of type
$\texttt{Configuration} * \texttt{Angle} \mapsto \texttt{Configuration}$

(6) $\texttt{turnTo}(C,D) =$
$\texttt{Configuration}(\texttt{orientation}(D), \texttt{point}(C))$
of type
$\texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Configuration}$

(7) $\texttt{turnAround}(C) =$
$\texttt{if}(\texttt{hasOrientation}(C))$
$\texttt{then}$
$\texttt{Configuration}(\texttt{inverse}(\texttt{orientation}(C)), \texttt{point}(C), \textit{true})$
$\texttt{else}\ C$
of type
$\texttt{Configuration} \mapsto \texttt{Configuration}$

■

(1) rotates the orientation of C by the amount of the angle A to the left.

(2) rotates the orientation of C by the amount of the angle A, in counterclockwise direction.

(3) rotates the orientation of C by the amount of the angle A to the right.

(4) rotates the orientation of C by the amount of the angle A, in clockwise direction.

(5) returns a configuration located at the position defined by the coordinates of C with the orientation of the value of the angle A.

(6) returns a configuration located at the position defined by the coordinates of the configuration C with the orientation of the configuration D.

(7) returns a configuration at the same position, but with an inverted orientation. If the configuration features no orientation, it remains unchanged.

Definition 4.92 (Inverse)
This function computes the inverse of spatial entities:

(1) $\texttt{inverse}(P) =$
$\texttt{Point}(-\texttt{xCoordinate}(P), -\texttt{yCoordinate}(P))$
of type
$\texttt{Point} \mapsto \texttt{Point}$

(2) $\texttt{inverse}(C) =$
$\texttt{if}(\texttt{hasOrientation}(C))\texttt{then}$

$\texttt{Configuration}(\texttt{inverse}(\texttt{orientation}(C)),$
$\texttt{inverse}(\texttt{point}(C)), \mathit{true})\texttt{else}$

$\texttt{Configuration}(0, \texttt{inverse}(\texttt{point}(C)), \mathit{false})$
of type
$\texttt{Configuration} \mapsto \texttt{Configuration}$

■

(1) returns a point located at the position point symmetric to the point P.

(2) returns a configuration located at the position point symmetric to the configuration C. If the configuration has an orientation, then the orientation angle is inverted, too.

4.7.2. Direction, Bearing and Orientation

MPLL provides the following predicates to ascertain the spatial relation of configurations C, D and E. Note the different configurations denoting the referent, the relatum, and, in deictic settings, the point of view or utterance.

Definition 4.93 (`direction`)
The `direction` *function returns a newly constructed* `Angle` *using the values from the predefined constants as listed in Def. 4.73 on page 149.*

(1) $\texttt{direction}(\mathit{Dir}) = \texttt{Angle}(\texttt{carDir}(\mathit{Dir}))$ $\texttt{CarDir} \mapsto \texttt{Angle}$
(2) $\texttt{direction}(\mathit{Dir}) = \texttt{Angle}(\texttt{carDir}(\mathit{Dir}))$ $\texttt{EgoDir} \mapsto \texttt{Angle}$
(3) $\texttt{direction}(\mathit{Dir}) = \texttt{Angle}(\texttt{carDir}(\mathit{Dir}))$ $\texttt{EgoDirD} \mapsto \texttt{Angle}$

■

(1) returns the angle value associated with the cardinal direction *Dir* as an `Angle`(see Def. 4.73 on page 149 for the default mapping).

(2) returns the angle value associated with the egocentric cardinal direction *Dir* as an `Angle`.

(3) returns the angle value associated with the deictic cardinal direction *Dir* as an `Angle`.

Definition 4.94 (Absolute Cardinal Direction)
Absolute cardinal direction includes, for example, expressions like "east" *or* "southwest". *Because they are defined by an extrinsic RS, they do not depend on the orientation of the referent.*

(1) $\texttt{bearing}(Dir,C,D) =$
$(\texttt{bearing}(C,D) == \texttt{direction}(Dir))$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Bool}$

(2) $\texttt{bearingF}(Dir,C,D) =$
$\texttt{member}(\texttt{bearing}(C,D), \texttt{directionF}(Dir))$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Float}$

(3) $\texttt{bearingF}(Dir,C,D,Fth) =$
$\texttt{member}(\texttt{bearing}(C,D), \texttt{directionF}(Dir), Fth)$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} * \texttt{Float} \mapsto \texttt{Bool}$

(4) $\texttt{bearing}(Dir,C,L) =$
$(\texttt{bearing}(C,L) == \texttt{direction}(Dir))$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Bool}$

(5) $\texttt{bearingF}(Dir,C,L) =$
$\texttt{member}(\texttt{bearing}(C,L), \texttt{directionF}(Dir))$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Float}$

(6) $\texttt{bearingF}(Dir,C,L,Fth) =$
$\texttt{member}(\texttt{bearing}(C,L), \texttt{directionF}(Dir), Fth)$
of type
$\texttt{CarDir} * \texttt{Configuration} * \texttt{Configuration} * \texttt{Float} \mapsto \texttt{Bool}$

■

Predicate (1) tests whether configuration *D* is located in direction *Dir* from configuration *C*, whereas *Dir* is of type `CarDir` (e.g. *"east"* or *"southwest"*). The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this function is neither crisp nor fuzzy, but requires equal values to return `true`. See other variants for fuzzy treatment of values.

Predicate (2) tests whether configuration *D* is located in direction *Dir* from configuration *C*, whereas *Dir* is of type `CarDir` (e.g. *"east"* or *"southwest"*). The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this version is fuzzy, and the return value, therefore, is a `Float` within the interval $[0,1]$.

Predicate (3) operates analogous to (2), with the exception, that a threshold value Fth must also be specified, so that the function returns `true`, if the return value is $\geq Fth$, and `false`, otherwise.

Definition 4.95 (Relative Cardinal Direction)
Relative cardinal direction includes, for example, expressions like "left" *or* "front". *Because they are defined by an intrinsic RS, they depend on the orientation of the referent.*

(1) $\texttt{bearing}(Dir,C,D) =$
$(\texttt{bearing}(C,D) == \texttt{direction}(Dir,C))$
of type
$\texttt{EgoDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Bool}$

(2) $\texttt{bearingF}(Dir,C,D) =$
$\texttt{member}(\texttt{bearing}(C,D), \texttt{directionF}(Dir,C))$
of type
$\texttt{EgoDir} * \texttt{Configuration} * \texttt{Configuration} \mapsto \texttt{Float}$

(3) $\texttt{bearingF}(Dir,C,D,Fth) =$
$\texttt{member}(\texttt{bearing}(C,D), \texttt{directionF}(Dir,C), Fth)$
of type
$\texttt{EgoDir} * \texttt{Configuration} * \texttt{Configuration} * \texttt{Float} \mapsto \texttt{Bool}$

■

Predicate (1) tests whether configuration *D* is located in direction *Dir* from configuration *C*, whereas *Dir* is of type `EgoDir` (e.g. *"left"* or *"front_right"*), and the processing depends on the orientation of *C*. The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this function is neither crisp nor fuzzy, but requires equal values to return `true`. See other variants for fuzzy treatment of values.

Predicate (2) tests whether configuration *D* is located in direction *Dir* from configuration *C*, whereas *Dir* is of type `EgoDir` (e.g. *"east"* or *"southwest"*), and the processing depends on the orientation of *C*. The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this version is fuzzy, and the return value, therefore, is a `Float` within the interval $[0,1]$.

Predicate (3) operates analogous to (2), with the exception that a threshold value Fth must also be specified, so that the function returns `true`, if the return value is $\geq Fth$, and `false`, otherwise.

Definition 4.96 (Deictic Cardinal Direction)
Relative cardinal direction in a deictic setting includes, for example, expressions like "left of" *or* "in front of" *with respect to a certain point of view (configuration D). Because they are defined by an intrinsic RS, they depend on the orientation of the referent. The following definitions adhere to the default naming scheme, i.e. C denotes the user's position, D denotes the referent and E denotes the relatum.*

(1) $\texttt{bearing}(Dir, C, D, E) =$
$(\texttt{bearing}(C, D) == \texttt{direction}(Dir, \texttt{bearing}(E, C)))$
of type
$\texttt{EgoDirD} * \texttt{Configuration} * \texttt{Configuration} *$
$\texttt{Configuration} \mapsto \texttt{Bool}$

(2) $\texttt{bearingF}(Dir, C, D, E) =$
$\texttt{member}(\texttt{bearing}(C, D), \texttt{directionF}(Dir, \texttt{bearing}(E, C)))$
of type
$\texttt{EgoDirD} * \texttt{Configuration} * \texttt{Configuration} *$
$\texttt{Configuration} \mapsto \texttt{Float}$

(3) $\texttt{bearingF}(Dir, C, D, E, Fth) =$
$\texttt{member}(\texttt{bearing}(C, D), \texttt{directionF}(Dir, \texttt{bearing}(E, C)), Fth)$
of type
$\texttt{EgoDirD} * \texttt{Configuration} * \texttt{Configuration} *$
$\texttt{Configuration} * \texttt{Float} \mapsto \texttt{Bool}$

■

Predicate (1) tests whether configuration D is located in direction Dir from configuration C as seen from configuration E, whereas Dir is of type `EgoDirD` (e.g. *"left_of"* or *"behind"*), and the processing depends on the bearing from C to E. The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this function is neither crisp nor fuzzy, but requires equal values to return `true`. See other variants for fuzzy treatment of values.

Predicate (2) tests whether configuration D is located in direction Dir from configuration C as seen from configuration E, whereas Dir is of type `EgoDirD` (e.g. *"left_of"* or *"behind"*), and the processing depends on the bearing from C to E. The mapping from cardinal direction identifiers to numeric values can be modified to suit the application requirements. Note that this version is fuzzy, and the return value, therefore, is a `Float` within the interval $[0, 1]$.

Predicate (3) operates analogous to (2), with the exception, that a threshold value Fth must also be specified, so that the function returns `true`, if the return value is $\geq Fth$, and `false`, otherwise.

Definition 4.97 (`bearingPoint`)
MPLL provides the following composite functions, with configuration C and direction Dir:

(1) `bearingPoint`(*Dir*,*C*)
of type
`CarDir*Configuration` $\mapsto$ `Point`
(2) `bearingPoint`(*Dir*,*C*)
of type
`EgoDir*Configuration` $\mapsto$ `Point`
(3) `bearingPoint`(*Dir*,*C*)
of type
`EgoDirD*Configuration` $\mapsto$ `Point`

■

(1) returns a point which lies on the linear extension of the bearing parallel to an extrinsic reference system, for example `north/east/south/west`. The position of the point *does not depend* on the orientation of the configuration. It is defined as follows:

`move`(*C*,`bearing`(*C*,`bearing`(`north/east/south/west`,*C*)),
`unitlength`(*RS*))

(2)–(3) returns a point which lies on the linear extension of the bearing of the configuration, indicated by the enumeration type given in the function call, for example `in_front/behind`. The position of the point *depends* on the orientation of the configuration. It is defined as follows:

```
if    hasOrientation(C)
then  move(C,bearing(C,bearing(in_front/behind,C)),
      unitlength(RS))
else  ERROR
```

Definition 4.98 (`bearingRegion`)
MPLL provides the following composite functions, with configuration C, direction Dir, and distance Adist:

(1) `bearingRegion`(*Dir*,*C*,*Adist*,*Abeam*)
of type
`CarDir*Configuration*Angle` $\mapsto$ `Polygon`
(2) `bearingRegion`(*Dir*,*C*,*Adist*,*Abeam*)
of type
`EgoDir*Configuration*Angle*Angle` $\mapsto$ `Polygon`
(3) `bearingRegion`(*Dir*,*C*,*Adist*,*Abeam*)
of type
`EgoDirD*Configuration*Angle*Angle` $\mapsto$ `Polygon`

■

(1)–(3) returns a (virtual) point which lies on the linear extension of the bearing of the configuration, indicated by the enumeration type given in the function call, for example `in_front/behind`. The position of the point depends on the orientation of the configuration. It is defined as follows:

```
if    hasOrientation(C)
then  move(C,bearing(C,bearing(in_front/behind,C)),
      unitlength(RS))
else  ERROR
```

4.7.3. Other Composite Functions

A number of other functions in MPLL can be defined as composite MPLL constructs which do not directly rely on any hard coded internals. In course of the use of MPLL this part of the library will be expanded as necessary constructs are added. In this initial version there are currently no functions in this section.

4.8. Summary

This chapter introduced the fundamental design aspects of MPLL and showed the relation to its temporal counterpart, the Specification Language for Geo-Temporal Notions (GeTS). The complete MPLL specification has been laid out, including basic types and functions, as well as the complete MPLL Standard Library.

5. Application

This chapter shows the application of MPLL by illustrating several tasks and subtasks in spatial information processing, as well as possible solutions to the problems encountered.

In order to illustrate the use of MPLL, and for introductory purposes, the examples in the next section are restricted to selected subtasks of more complex processing sequences.

5.1. Properties

In this section we take a look at some examples regarding the properties of basic types. Then, we go through some subtasks of generating route descriptions which involve the classic spatial relations pertaining to direction and distance.

Some fundamental functions of MPLL concern the modification of the properties of spatial entities. First and foremost, this applies to the representation of mobile entities: the configuration data type. This type is used, for example, to model a user moving through planar space.

The position of a configuration C can be modified by several functions:

(1) `moveX`(C, *Distance*)
(2) `move`(C, *XDistance*, *YDistance*)
(3) `move`(C, D, *distance*)
(4) `moveTo`(C, D)

These functions realise movement along a single axis (1), along both axes (2), in direction of an entity D (3), or directly to the position of an entity D (4). Functions like these are needed for spatial translation, i.e. moving entities to certain locations in space.

(1) `turnTo`(C, D)
(2) `turnCW`(C, A)
(3) `turnAround`(C)

The orientation of C can be directed to face another entity (1), it can be modified in either direction (2), or it can be inverted (3). Since the orientation, or intrinsic front, of an entity greatly influences some spatial relations, it can be adjusted in these and various other ways.

5.2. Transformation of List Elements

In some cases, suitable MPLL functions may not be available and will have to be individually defined. For this example, we assume that the function `scale` does not exist. Furthermore, we want to find an expression, which not only substitutes this function, but also allows us to use it on a list of points. Let `pointList` be a list of points $(P_1, \ldots, P_n)$. The expression

```
 map(lambda(Point P)
2      (Point(xCoordinate(P)*3.0 , yCoordinate(P)*3.0)), pointList)
```

will scale the coordinates of all points in `pointList` by the factor 3.0 from the origin of the reference system.

5.3. Angular Relation in Route Descriptions

A typical example for the application of MPLL is the subtask of generating route descriptions, presented in this section. We assume that a route planner has produced a route from a starting location P_0 to a destination position P_n via $n-1$ intermediate locations P_i, with $i = (1, \ldots, n-1)$. The route segments $L_j = (P_{j-1}, P_j)$, with $j = (1, \ldots, n)$ are also part of the route data structure. The user's position is represented by the configuration U. A number of landmarks M_k are also available, which are stored in a database and can be selected, for example, by their type, their properties, or by geometric distance to a reference location.

The concrete scenario is illustrated in Fig. 5.1. The locations P_i designate the locations along the route. Landmarks are denoted by the green and red dots, partly marked M_k. For clarity, the landmarks denoted by red dots and the route segments have not been marked individually in the figure. The dotted circle denotes the geometric distance which is defined by the enumeration type `close`.
We further assume that in the sequential processing of the intermediate locations we are at the junction P_4 and that the list `landmarks` contains all landmarks M_k.

Now, the subtask involving MPLL is establishing *significant* spatial relations between the intermediate locations P_i and the landmarks M_k. By significant, we mean those relations, which are clear in the sense of human perception. In this case, we want landmarks,

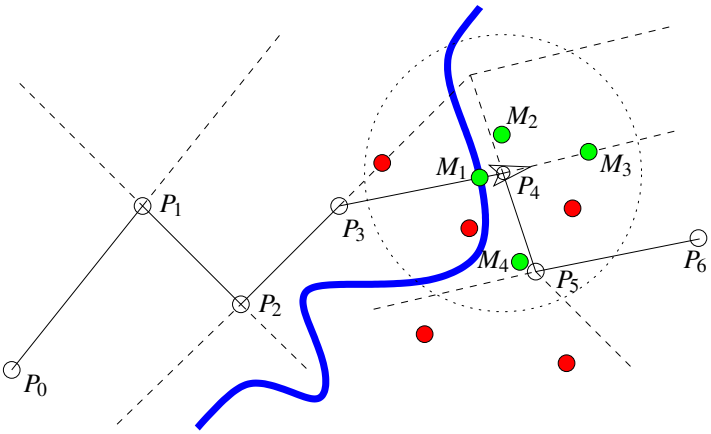


Figure 5.1.: Scenario for Angular Relation

which are located in either one of the four primary egocentric cardinal directions: *front*, *back*, *left*, or *right*.

First, the user's position and orientation must be updated to the current location, P_4, in the sequence of route locations. We accomplish this using the transformation functions `move` and `rotate`:

$$\texttt{moveTo}(\texttt{turnTo}(U, \texttt{bearing}(P3, P4)), P4)$$

It is important to note that this is not the only way to modify the properties of U, i.e. the *position* and the *orientation*. The modification of both values is relevant for the computation of the egocentric bearing. An alternative way, among others, is the configuration constructor:

$$\texttt{Configuration}(\texttt{bearing}(P3, P4), P4)$$

Whatever functions suit best depends on individual factors and has no technical consequences. Basically, readability is of prime importance. The latter example is used subsequently.

The next step towards generating route descriptions can be taken by evaluating the following expression. It is a filter expression with nested lambda expressions. The first expression filters out the landmarks which are not within `close` distance. The second one is a concatenated boolean expression which tests for the bearing to match either one of the four egocentric cardinal directions. This function is parameterised by the numeric value 0.8. This value means that only those bearings qualify, which score at least a

fuzzy value of 0.8 in comparison with the circular interval defining a cardinal direction (see Fig. 2.26 on page 76 and the discussion in section 2.7.3 for more details on fuzzy representation of direction).

```
Let C = Configuration(bearing(P3,P4),P4) in
  Let threshold = 0.8 in
    filter(lambda(Point P)(
      (lambda(Point Q)(
         maxDistance(C, Q, close)(Q)) &&
        (bearing(C, P, front, threshold) ||
         bearing(C, P, left, threshold)  ||
         bearing(C, P, back, threshold)  ||
         bearing(C, P, right, threshold))
       (P)), landmarks))
```

Alternatively, we need not nest the lambda expressions, but can instead nest the filter expressions. While producing the same result, the expression would look like this:

```
Let C = Configuration(bearing(P3,P4),P4) in
  Let threshold = 0.8 in
    filter((lambda(Point Q)(
           bearing(C, Q,front, threshold) ||
           bearing(C, Q,left, threshold)  ||
           bearing(C, Q,back, threshold)  ||
           bearing(C, Q,right, threshold))(P)),
      filter(lambda(Point P)
             (maxDistance(C,P,close)(P)),landmarks))
```

The result is a list of landmarks which qualify with respect to two properties:

1. the Euclidean (L_2) distance to the user's position
2. the bearing from the user's position to the landmark

The first property satisfies the requirement that only those landmarks are to be taken into account which are *"close"* enough to the user's position. The evaluation of the vague notion of closeness is achieved by the definition of the fuzzy distance denoted by the keyword *"close"*, which can be modified to reflect the context of use and the user preferences.

The second property serves to discard those landmarks that are not in a clear cardinal direction from the user's position and orientation. Since human spatial cognition features very clear expressions for the definition of the primary egocentric cardinal directions (*"in front"*, *"behind"*, *"left"*, and *"right"*), landmarks located in one of these directions are primarily important for the orientation and navigation of the user. Unclear expressions,

such as *"sharp left"* or *"right front"*, serve well neither as trigger cues nor as reassuring cues.

In a similar manner, direction can be expressed using surrounding landmarks. While in the previous step the task was to find suitable landmarks in order to facilitate user positioning (*"...after crossing the River (M_1) you come to a junction where you can see a big fountain (M_3) straight ahead and a church (M_2) to your left..."*, i.e. providing a reassuring cue, the process of generating a trigger cue might require a different set of landmarks. Non-cardinal directions can easily be expressed using landmarks, for example by directing users *towards* or *away* from them. The following expression serves to filter out all landmarks which are close enough to the user's position and are located in the direction the user has to proceed from there:

```
Let C = Configuration(P4) in
  Let Dir = CircularInterval(bearing(P4,P5)) in
    Let threshold = 0.8 in
      filter((lambda(Point Q)
               (member(bearing(C,Q),Dir)>threshold)(Q)),
        filter(lambda(Point P)
                (maxDistance(C,P,close)(P)),landmarks))
```

This expression returns a list of landmarks that are located in the same direction as the bearing from P_4 to P_5, i.e. in the direction the user has to move *towards*. Note that the (implicit) default parameters of the `CircularInterval` constructor (see Def. 4.87) can be modified so as to reflect the preferred definition of fuzzy direction (again, see Fig. 2.26 on page 76 and the discussion in section 2.7.3 for more details).

Landmarks in the opposite direction, i.e. the landmarks that the user has to move *away* from, can be produced by the following expression:

```
Let C = Configuration(P4) in
  Let IDir = CircularInterval(inverse(bearing(P4,P5))) in
    Let threshold = 0.8 in
      filter((lambda(Point Q)
               (member(bearing(C,Q),IDir)>threshold)(Q)),
        filter(lambda(Point P)
                (maxDistance(C,Q,close)(P)),landmarks))
```

5.4. Summary

The examples in this chapter can illustrate only a fraction of the possible applications of MPLL. The language was intended as a very flexible means for processing and reasoning in spatial scenarios and can, therefore, be applied to a huge number of very different and very specific tasks. As development progresses and more applications are found, this

section of the documentation will be populated with additional examples illustrating complementary scenarios and tasks.

6. Related Work

The following two sections introduce important work regarding qualitative orientation and distance, and indicate further reading. An overview of techniques and models developed in these two fields has been compiled, for example, by Escrig and Toledo [51] and Escrig [52]. Section 6.3 discusses related projects which have been developed concurrently to the work presented in this thesis and which have been supervised by the author. Then follows a detailed description of related standards. This concerns in particular the RDS/TMC system, which has been of major importance for some of the project work. A short summary concludes this chapter.

6.1. Qualitative Orientation

There mainly exist [52] three models for qualitative orientation that are not based on projection into external reference systems: Frank [57], Freksa [59, 60] (see also Freksa and Zimmermann [58]), and Hernández [76]. All these models divide space into qualitative regions using references systems that are centred on the reference objects, i.e. using local and egocentric RSs. The representational primitives for spatial objects are points. The following sections further introduce the three models.

6.1.1. Egocentric Motion-based Reference System (Freksa & Zimmermann)

The model proposed by Freksa [59, 60], and Freksa and Zimmermann [58] respectively, introduced the so-called *double-cross* calculus. Direction from a reference point to a located point is defined with respect to a perspective point. The approach uses three axes: one is specified by the reference point and the perspective point, the other two axes are perpendicular to the first one and go through the reference point and the perspective point respectively. As a result, 15 base relations are defined. The inference mechanism

allows for [51] the following reasoning[1]: ‘*“given two relationships ‘c wrt ab’ and ‘d wrt bc’, what is the relationship ‘d wrt ab’?”*

Scivos and Nebel [146] recently studied the computational properties of this calculus. They proved reasoning with the 15 base relations to be NP-hard.

6.1.2. Indoor fixed Spatial Orientation (Hernàndez)

Hernández [76] combines a cardinal reference system for orientation (such as shown in Fig. 2.2 c and d), on page 46) with a topological domain not unlike RCC-8, into a structure called *Relative Topological Orientation Node (rton)*. Each node in this rton denotes combinations of topology and orientation, while the edges denote neighbouring pairs.

The intrinsic front pertains to the main entrance of a room – the model is specifically designed for indoor spatial orientation – which could be prone to ambiguity in cases, where a room has multiple entrances or exits, or in cases of very large and/or irregularly shaped rooms. Escrig [51] provides a short introduction to this model, including an illustration of an rton.

6.1.3. Cardinal Reference System (Frank)

Frank [57] proposed two different models for describing cardinal directions of different granularity (e.g. *north*, *southeast*, *left of*, or *in front*) between point entities: a *cone-based* method and a *projection-based* method. The latter method facilitates the representation of nine different relations in terms of a point algebra (see Renz [137], section 2.5, pp. 27) by separately specifying a relation for each of the two axes. Ligozat [100] studied the computational properties and found, in particular, that reasoning with the projection-based approach is NP-complete. The cone-based approach is essentially similar [51] to the approach by Hernàndez described in section 6.1.2.

6.2. Qualitative Distance

Some early approaches dealing with qualitative distance were introduced by Mukerjee and Joe [114] and Zimmermann [173]. Chang and Jungert [22] presented a qualitative orientation model based on projection, which was extended in order to include named distances by Jungert [89]. Other qualitative orientation models have been extended with the concept of named distances by Zimmermann [174, 175], Frank [57], and Clementini, Di Felice, and Hernández [77, 26].

[1] The expression *wrt* is to be read *“with respect to”*.

6.3. Related Projects

Under the author's supervision, several related projects have been developed concurrently to this work. Each of these projects can either function as a module which can be integrated into the MPLL system architecture (e.g. TransRoute), or it can aid in the processing of data and/or in providing data to be processed with MPLL.

6.3.1. Ontology for Transportation Networks (OTN)

An essential foundation for interoperable applications is a holistic concept of the underlying structures of the data to be processed, i.e. an ontology. The purpose of the Ontology of Transportation Networks (OTN) [106] is to provide such a foundation for applications which deal with locations, locational relationships, and mostly with the aspects of locomotion and transport.

The Origins of OTN

Different parties have worked on standards and interfaces in geographic data interchange since the late eighties. In 1993, the technical commission TC204 of the *International Organisation for Standardisation* (ISO) [79] began work on *Intelligent Transport Systems* (ITS) [82]. The aim of their working group 3 was to review existing regional standards, which revealed to be highly heterogeneous. While the *Japan Digital Road Map Association* (JDRMA) [85] mainly worked on standards catering for navigation systems and the necessary optimisations therefore, the American *Spatial Data Transfer Standard* (SDTS) [147] was designed to facilitate the description of records, but not the standardisation of content. In Europe, the Geographic Data Format (GDF) [63] was developed as an extensible and application-independent data model for transport systems. Subsequently, seven countries[2] continuously revised and extended the GDF, which led to the release of GDF 4.0 on 21. March 2002 as the official ISO standard ISO-14825 [84] for geographic data interchange in transport applications.

While the underlying model of GDF mainly includes a thorough representation of car traffic and road networks, other modes of transport have received less attention. Additional elements, such as services or public transport, are not included in GDF. OTN incorporates the comprehensive model of road networks underlying GDF, and extends the ontology to compensate for the neglected fields. Further extension is not only possible, but also desirable, since there cannot be a complete ex-ante model of all traffic and transport related affairs – nor for any other domain for that matter. To be usable with today's web infrastructures, OTN is specified in the *Web Ontology Language* (OWL). OTN contains some extensions which are not present in GDF: schedules, services and meteorology.

[2] Australia, Canada, Germany, Japan, Korea, the Netherlands and the U.S.

Schedules

OTN was developed as an integrated approach to modelling private and public transport. Therefore, one of the most important features is the specification of schedules. In GDF, the specification of schedules is limited to providing a time frame (start and end time) and a network segment (road or ferry segment) to convey that, for example, the ferry from Staten Island to Manhattan operates from 04:30 am to 11:30 pm. Further specification of travel times, intervals and such is not possible.

Because of the importance for (multi-modal) routing, OTN facilitates the definition of departure times, travel times and time frames. A typical segment or connection in public transport has an attribute *"timetable"*, which holds a series of schedules. Each schedule is valid during *"validity_Period"*, i.e. the time frame in which the respective means of transport operates, while *"loop_Time"* defines the interval. *"starts_at"* contains the starting node (which can be located at either end of an edge), and *"travel_Time"* contains the regular or individual duration of travel. Optionally, a *"waiting_Time"* indicates an idle time before departure. A ferry, which commutes hourly between 06:30 am and 06:30 pm from node `A` to node `B`, has a travel time of 30 minutes, and can be loaded 20 minutes prior to departure, could have a schedule like the following:

```
<Timetable rdf:ID="Timetable_A-B">
  <starts_at rdf:resource="#A"/>
  <waiting_Time>m20</waiting_Time>
  <loop_Time>h1</loop_Time>
  <travel_Time>m30</travel_Time>
  <validity_Period>
    <Validity_Period rdf:ID='validity_Timetable_A-B'>
      <time_duration>h12</time_duration>
      <starting_Date>h6m30</starting_Date>
    </Validity_Period>
  </validity_Period>
</Timetable>
```

Services

OTN caters for different aspects beyond those pertaining to routing and navigation. *Services* represent one of these aspects.

GDF generally introduces the notion of services, although – among a series of proposed services – only one is implemented: the service *"Entry_Point"* defines the access to a service.

OTN includes most of the GDF proposed services and provides further extensions. The attribute *"is_Accessible_at"* renders the GDF-service *"Entry_Point"* useless[3], therefore it has not been taken up in OTN. One service, which is very important for the purpose of OTN, is called *"Transfer_Service"*. It describes means to change the transport

[3] *"Entry_Point"* only represents another service and it's accessibility.

vehicles, for example, from car to train. Parking places, for example, are modelled as part of transfer services.

Meteorology

New in OTN is the possibility to store weather information. There is the topic *'Meteorology'*, which is subdivided into the classes *'Temperature'* and *'Weather'*. These are subclasses of *'Face'* and define an area with the actual temperature and the kind of weather, which can be any one of the following: *snow*, *sleet*, *hail*, *dew*, *rain*, *shiver* and *storm*.

6.3.2. Local Data Stream Management System

Many practical applications of geospatial information processing systems benefit immensely from up-to-date dynamic information. Navigation systems are a prominent example where dynamic information is most useful. 'Static' queries to (XML-) databases can, however, also benefit from dynamic information. A straightforward answer to a query like *"where is the nearest pharmacy"* can be useless when the road to the closest pharmacy is currently blocked. As previously mentioned, there exist in fact many geospatial relations whose evaluation amounts to a path planning problem. Solving these problems essentially depends on the state of the underlying infrastructure and therefore requires up-to-date dynamic data about it.

Dynamic information usually comes in form of streams of data, and these data streams must be processed, usually in several steps, until they can be fed into the final application system.

In the REWERSE deliverable A1-D6 [105], we presented two developments. The first one is the *Local Data Stream Management System* (L-DSMS). Data stream management systems are, for example, used in grids to control the flux of large amounts of data from the data sources, telescopes, for example, to world wide distributed computer centres. This is *not* an application for L-DSMS. L-DSMS is *local* in the sense that it facilitates the specification and construction of a single Java program which consists of a network of nodes for processing streams of data. Each such node receives data from one or several data sources, processes them in a certain way, and delivers the processed data to one or more data drains. A data drain can be the data source for the next processing node in the network, or it can be the end application in the whole processing chain. One of the components of L-DSMS is the SPEX XML–filtering system [129, 130]. It processes XPath [23] queries on a stream of XML data and can be used to extract interesting information from XML streams.

The L-DSMS network is configured by XML–files. They contain the list of nodes, and for each node its sources and drains. Each node corresponds to a Java class whose methods do the actual processing. The L-DSMS reads the configuration files, loads the corresponding Java classes and arranges them into the required network.

The second development is an application of the L-DSMS for processing dynamic traffic information. The traffic information comes from RDS-TMC receivers and is processed in several steps before it is delivered to several application systems.

The availability of Traffic and Travel Information (TTI) for any device or application is depending on the accessibility of some source for this type of information. Car navigation systems can, for example, be connected to an RDS-compatible radio receiver. As sales have risen and RDS radios have become quite common, marginal costs for the necessary hardware components have been declining, which further opens the market to other devices and fields. GPS receivers for mobile solutions sometimes also contain an RDS-compatible FM receiver, although this is not as common.

However, in cases where there is no FM receiver at hand, an alternative source for TTI must be found. Routing applications running on standard PCs (which usually don't have built in FM receivers), for example, belong to this category. One possible solution is to substitute radio transmission and receiver hardware with an internet connection and a web service, which has been the objective of this project. The main goal of the project was to provide the following functionality:

- **FM Receiver**

  This is the first of only two hardware components of the system. In our testbed, there are currently two RDS-capable FM receivers attached via serial ports to a server. They can be tuned to different radio stations and can be set to provide a raw binary RDS data stream at the serial port. By design, the FM receiver shall neither block the frequencies from 15 to 23kHz nor above 53kHz, since this is where RDS data is transmitted (see Fig. 6.1).

- **RDS Decoder**

  This second hardware component produces the raw RDS bit stream by isolating and decoding the signals around 57kHz at a rate of 1187.5 bits per second. This stream is directly delivered to the receiver's serial port, which is connected to the server machine.

- **TMC Decoder**

  The first task entirely realised in software is the decoding of RDS groups from the raw data stream. RDS groups consist of 4 data blocks which contain 26 bits each. Of these 104 bits ($4 * 26$), 40 bits (10 in each block) are used for error correction, which leaves a net payload of 16 bits per block or 64 bits per group (see Fig. 6.2). TMC messages contain the group id '*8A*'.

- **XML Stream Generator**

  At this stage, the raw TMC data are transformed to XML corresponding to a customised schema. Furthermore, the data are enriched with the contents of the

Event Code List (ECL) and *Location Code List* (LCL). This enables devices which cannot access these code lists to nevertheless display the textual contents of TMC messages, instead of rather cryptic raw binary data.

- **SPEX Filter Mechanisms**

  In contrast to classic querying of relational data, which produces a result set designed to meet the users demands, the processing of data streams requires other mechanisms to query or filter the incoming data. L-DSMS makes use of a system called SPEX [129, 130], which has been developed by a former member of the Munich team, Dan Olteanu. It is a powerful filtering mechanism for our system. This way, the stream optionally passes one or more nodes which filter according to certain criteria to produce a suitable output stream.

- **Configuration Component**

  In order to facilitate easy (re-)configuration of the different components, the networks of nodes which the stream passes through is configured entirely via a single XML file. In this file, the respective sources and drains, as well as (filter-) nodes and the necessary parameters are specified.

- **Visualisation Component**

  Apart from the textual output of TMC messages, which strongly resemble the usual spoken announcements on the radio, graphical output in form of symbols on a map display is also provided. Easily implemented on different digital map systems, we show the basic procedure of how to integrate these graphical messages with an SVG-based map system rendered in a conventional internet browser window. Strictly speaking, this component is not part of the TMC to XML transformation prototype. Nevertheless, output mechanisms similar to the one provided here would logically be the consumers of the provided data stream.

In supplemental project work, Michael Buschmann and Markus Krieser [20] focussed on persistent storage and statistical evaluation of RDS/TMC data. Storing TMC messages in a database system facilitates the statistical evaluation, and, subsequently, calculating the likelihood of incidents on certain road segments at certain times. On weekdays, in the late afternoon, it is, for example, very likely that there is a traffic jam on the northbound highway A9 near Munich. Information like this is very valuable if the planning phase of in routing application is conducted well in advance of the execution phase.

In another related project [71], Christian Hänsel developed an interface for the TMC data provided by L-DSMS to be displayed in the popular Google Earth Client [66]. This work demonstrates the possibilities of integrating highly dynamic geospatial data with the static data provided by Google Earth, using the Keyhole Mark-Up Language (KML) [93].

6.3.3. TransRoute

This project started out as a diploma thesis [151] under the author's supervision and is now work in progress for a Ph. D. thesis, both by Edgar-Philipp Stoffel. TransRoute is an object-oriented framework for routing applications which is capable of not only representing various real-world transport networks (highways, public transport, etc.), but also of computing shortest path and nearest neighbour queries.

Basically, TransRoute distinguishes three different levels of abstraction by separating between *ontology concepts* describing real-world entities in the domain of *transport networks*, together with their attributes and relations, and a *graph structure*, which can be instantiated with the respective concepts from the ontology.

The core functionalities of the framework include the following aspects:

Ontologies – Functionality for loading OWL ontologies [161, 163] and for checking the domain types of graph elements against those is provided.

Graph Structure – An object oriented in-memory representation of a graph structure, mainly from the Java Universal Network/Graph Framework (JUNG) [88], is used. Graphs in TransRoute can contain vertices and both directed and undirected edges. The generic graph structure can be seen as a hull for the domain concepts defined in the ontology, which can be filled into the according graph element. The graph structure is reusable for different, for example hierarchical, concepts. For example, a vertex on a high abstraction level represents an entire transport network. Therefore, its incident edges represent connections to other transport networks. In contrast, another vertex may represent a concrete bus station whose adjacent edges are physical connections. Yet another application for a generic vertex is the modelling of a country, its incoming and outgoing edges representing adjacency, containment and part-of relations between other countries, continents and cities. Not only are all graph elements attributed, i.e. attributes can be attached to them arbitrarily, but also they comprise a built-in hierarchic structure which can be navigated and entails interesting semantic concepts ensuring its integrity. Especially for edges, numeric attributes can be considered to be edge weights. Some predefined attributes exist concerning location, geometry and the ontology class represented.

Persistence – In contrast to the transient in-memory structure mentioned above, the persistent representation of graphs is responsible for storing graphs to files adhering to the Graph Exchange Language (GXL) [69], a specialised Extensible Mark-Up Language (XML) derivate. By using the GXL API for Java [70], TransRoute offers functionality for both reading from and writing to GXL files. One of the primary benefits of GXL is that it facilitates representation of nested graphs within edges and vertices. Furthermore, the framework can be extended by additional plug-ins for different data formats (e.g. the Geography Mark-Up Language

(GML) [65] or the Geographic Data Format (GDF) [84]).

Entities of the underlying infrastructures are embedded in the ontology modelling, along with their attributes, which may either be fixed or dynamic. The corresponding graph elements take over all these properties.

Having decided to use directed and weighted graphs as mathematical formalism, routing is equivalent to searching shortest paths fulfilling special constraints in graphs by standard algorithms or some of their derivatives. Employing a shortest path algorithm, one can profit from an important principle of dynamic programming, stating that such an algorithm being too complex can be subdivided into smaller problems, as Cormen et al. have proved [35].

Mostly, these efficient derivates of standard algorithms in form of heuristics comprising techniques of reasoning are of practical value for large-scale computer models since they yield faster results at the trade-off of less accuracy. Some of them are showing proximity to human cognition and, therefore, can be considered to be a natural way of finding a solution [21].

A further improvement manifests itself in the hierarchic graph structure [19] making the framework applicable beyond real life transport networks: Buildings as well as cities including their interiors can, among other hierarchic entities, be represented by the generic graph structure. Integrity is maintained by strict restrictions for adding vertices and edges at the right place of the hierarchy.

Furthermore, multi-modal transport including transfers can be regarded more formally [10] in terms of combinations of transport modes forming certain patterns. Transport networks can be seen as generic vertices in an abstract graph, for which routing essentially decides the combination of transfers. All these aspects influence reasoning techniques for a more complex shortest path algorithm in transport networks. Altogether, the results presented by Stoffel [151] comprise a suitable module to be integrated into the MPLL system architecture.

6.4. Related Standards: Traffic Information via RDS/TMC

Apart from standards and technologies already mentioned in this chapter, such as the Geographic Data Format (GDF) [63, 84], the Geography Mark-Up Language (GML) [65], or the Web Ontology Language (OWL) [161, 163], mainly the set of standards regarding RDS/TMC [94, 80, 81, 83] has been used by related projects. In this section, we take a closer look at one of the Radio Data System (RDS) services, the Traffic Message Channel (TMC). TMC data was used as an example for a stream of traffic information to be incorporated into dynamic routing applications.

6.4.1. The Radio Data System (RDS)

The Radio Data System (RDS) [94] is a narrow-bandwidth data transmission channel for VHF/FM broadcasting. RDS supports data transmission alongside (see Fig. 6.1) sound broadcasts, and facilitates services which are based on sending a small amount of digital data to a great number of users. It was developed in the 1970s and early 1980s and is now implemented all over Western Europe, several Central and East European countries, in parts of Asia Pacific, South Africa and (using the slightly different [136] Radio Broadcast Data System (RBDS) [135] standard) the United States. Rather recent additions to RDS are TMC (see next section) and Open Data Applications (ODA) (not discussed here, see Kopitz and Marks [94], chapter 9, instead).

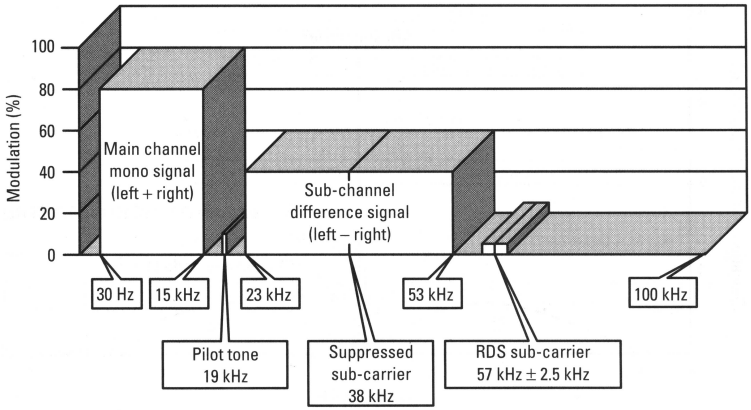


Figure 6.1.: Spectrum of a pilot-tone stereo multiplex signal with RDS [94]

Basic features of RDS include, among others, the following information features, tuning aids, and programme-related features. Because this list only serves to illustrate the basic ideas and functionalities of RDS, it is not exhaustive.

Information Features

- **Clock Time (CT)**: The current time and date can be transmitted in type '4*A*' groups by the radio stations to keep the internal clocks of the receivers within an accuracy of ± 0.1 seconds of a certain reference time (e.g. DCF77[4] (77.5

[4]DCF77 is not an abbreviation. It is the call-sign for the transmitter and stands for: "D" = Deutschland (Germany), "C" = long wave signal, "F" = Frankfurt, "77" = frequency: 77.5 kHz. It is transmitted three times per hour in Morse code.

kHz) in Germany or MSF[5] (60 kHz) in England).

- **Enhanced Other Networks (EON)** : Especially valuable for larger broadcast networks, EON information, which is transmitted in type '14' groups, allows the update of a number of features for programme services other than the currently tuned service. This includes for example AF, PIN, PS, PTY, TA (described below).

Tuning Aids (all of type '0*A*' group)

- **Programme Identification (PI)**: This identifier is not intended for display, but for identifying identical broadcasts on different frequencies. If reception quality is decreasing, and if they are equipped with a secondary FM tuner, RDS receivers can search for broadcasts on other frequencies with identical PI code which offer better reception quality (of the very same programme).
- **Programme Service Name (PS)**: This contains the static 8 character identifier to be displayed to the user.
- **Traffic Programme and Traffic Announcement (TP/TA)**: These flags indicate the availability of spoken traffic announcements on the currently tuned station (when used with EON also other stations). This enables the receiver to increase the volume and to stop CD or cassette playback whenever spoken announcements are transmitted.
- **Alternative Frequencies (AF)**: This feature provides alternative frequencies for the currently tuned station in order to optimise reception quality.

Programme-Related Features

- **Programme Type (PTY)**: A list of 29 standardised choices describing the broadcast programme enable the user to set the receiver to a certain programme type (e.g. news), and therefore not to choose a specific radio station, but a certain type of broadcast instead.
- **Radio Text (RT)**: Text messages of up to 64 characters can be coded and broadcast by the radio text feature, although many receivers, especially mobile ones, feature only displays with less than 64 characters.

6.4.2. The Traffic Message Channel (TMC)

TMC [80, 81, 83] was mainly developed in the years from 1984 to 1997 by a number of European companies and institutions, under the leadership of the European Broadcast Union (EBU), in order to broadcast TTI messages on VHF/FM broadcast transmissions

[5] MSF is also a call-sign (see previous footnote). "M" is the code for the United Kingdom, the letters "SF" were apparently randomly assigned. The call-sign does not feature the frequency.

using RDS [94]. TMC is one of several RDS features and services, although compared to some rather simple features such as tuning aids (PI, PS, TP) or programme-related features (PTY, RT, PIN), it is one of the most complex standards within RDS.

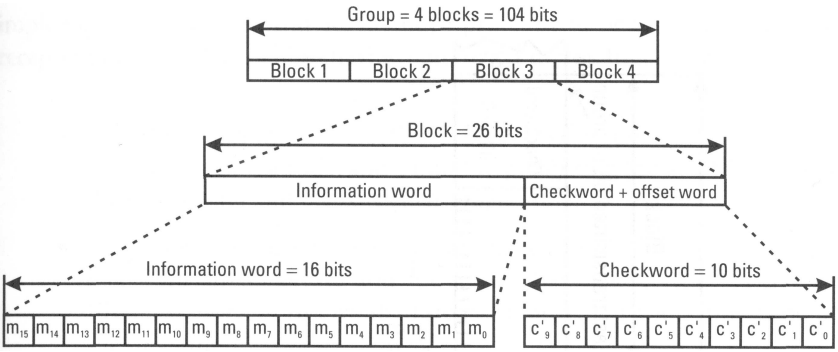


Figure 6.2.: Structure of RDS baseband coding [94]

The main advantages of digitally broadcast TMC messages over spoken traffic announcements are:

Asynchronous reception: Users need not be listening at the correct time to the correct radio station in order to receive information. This is especially important for individual traffic, since on-board systems must not interfere with the drivers' ability to concentrate on the traffic.

Caching mechanisms: Messages are stored in a client device and can be queried any time. Life cycle management ensures that outdated messages are erased from the memory.

Filtering mechanisms: Several mechanisms exist for filtering out unwanted content, for example by event type, current location, or projected path. Furthermore, short repetition cycles combined with duplicate elimination facilitate timely broadcast of information[6].

Language independence: The binary coded messages rely on the ability of the client devices to generate human understandable messages. This may require increased device intelligence, but also facilitates the use of different languages.

[6] A rather optimistic refresh cycle of 15 minutes would lead to an average latency of 7.5 minutes, whereas the typical TMC cycles of 120 seconds result in only 60 seconds of latency.

Message density: With 1187.5 bps the RDS bandwidth is comparably narrow from a current viewpoint. Although only some 300 messages can be transmitted per hour [94], this displays significant advantages over spoken messages. If the information of each message could be conveyed by an average of 15 seconds of spoken text, the same number of messages would still produce the unrealistic amount of about 75 minutes of announcement time.

Navigation assistance: By incorporating digital traffic announcements into car navigation systems, the task of navigation and route planning could be substantially improved.

Especially the last point is of great importance for intelligent transport systems, which need to take into account current traffic situations, as well as statistical data and data from simulations, in order to refine and further optimise the movement of goods and passengers in more and more complex scenarios.

In October 2004, the two major providers of digital map data, Tele Atlas [153] and NAVTEQ [115] announced future collaboration [101] on the standardisation of traffic codes for digital maps for the United States, which will be based on the European TMC Alert-C [80, 83] specifications.

6.5. Summary

This chapter described related work, primarily pertaining to qualitative direction and distance, along with a summary of related projects, diploma theses, and project theses, which have been carried out in connection with this thesis. The short introduction to the RDS/TMC standard in the previous section is primarily important in connection with the L-DSMS prototype.

7. Conclusion and Future Work

This section contains a discussion of the results presented in this thesis. Possible perspectives for extending this work are also laid out.

7.1. Conclusion

The previous chapters illustrated how the language MPLL facilitates bridging the gap between quantitative and qualitative spatial models, representations, and processing techniques.

MPLL enables users to individually specify the way they understand space by providing flexible and extendable language constructs which reflect the qualitative aspects of human spatial reasoning. It offers the means to adapt quantitative representations to these aspects, and it allows for the definition of individual functionality. As spatial notions are highly subjective, MPLL offers flexible means to adapt the specifications to individual needs and preferences, as well as the context of use in an implicit way. Comprehensive context and user modelling, however, was out of the scope of this work.

The overall architecture provides the framework in which different modules and services can be accessed by, or access, MPLL. This way, interfaces to different services, each specifically suited to perform special tasks, can be implemented. These services include, for example, qualitative reasoning on spatial relations between different entities, various geometrical computations, route planning, and more.

The fact that the MPLL specification in its current form is not final – probably any reader of the specification will find possible additions or might wish to make adjustments – only illustrates the flexibility of this approach and leads to the conclusion that a language like MPLL or GeTS will not be static, but a highly dynamic and evolving component.

7.2. Perspectives for Future Research

The work on MPLL is far from finished. Several possibilities for extending the language and the individual components present themselves. This section sketches some directions and the underlying ideas.

7.2.1. Ontology-based Language Constructs

In theory, a language like GeTS or MPLL could be modelled by using concepts which are defined in an ontology. Instead of generating different language constructs by hand (and extending the language as necessary over time), these could then be generated automatically through the ontology and using ontology reasoning. This idea would apply the basics of Model Driven Architecture (MDA) in software development to language design. It will be a great challenge to examine this issue and to specify the necessary mechanisms which are needed to reach this goal.

7.2.2. Comprehensive User and Context Modelling

As already indicated in section 2.8, the integration of a suitable context and user model is of key importance for the application of MPLL. While the language offers great flexibility to adapt spatial notions to individual tasks and domains, this is also necessary for the different modules and services which use, or are used by, MPLL.

A prominent example for this is route planning and navigation. The list of factors which influence the routing process is virtually endless. What is briefly summed up under the notion of context, is a very complex network of interdependent factors. The following examples are all but exhaustive and have to be adapted to individual cases.

Route planning for cars has been developed to marketability years ago. It is an available and proven technology which is, in addition, confined to a rather restricted environment (see section 2.1). However, already in this restricted domain the influence of context on the processing is very strong. Results depend primarily on the network structure, which is rather static. Still, the user can chose between the shortest and quickest route, which are usually very different. The user might prefer a different route at different times during the day, and another completely different one at nighttime. If we take traffic information into account, this opens up another dimension of context. Traffic is highly depending on the time of day, the date, cultural events (e.g. holidays, big events), the weather, the season, and many other factors which cannot be mentioned here. All these factors are part of the context.

Pedestrian navigation is yet more complex. Generally, if fewer restrictions and regulations exist, modelling becomes more complex. The individual user profile, also part of the context, becomes more important in pedestrian navigation. Some factors include age, gender, group composition, individual preferences, abilities, authorisation and licences, and many more.

Comprehensive user and context modelling is the key to flexible and adaptable systems and services, especially regarding spatial information processing. As considerable resources are already focussed on research in this field [5, 2, 74, 75, 73, 9, 1], there will be suitable models available for integration into MPLL and its components. However,

the individual requirements of the spatial domain must be considered, and they must be formally described.

7.2.3. Individual Libraries

If MPLL is used on a larger scale, its flexibility will, most likely, lead to a huge number of function definitions and extensions of the language in order to adapt to specific domains. As these individual libraries are developed, extended, and refined, some issues have to be considered.

Ideally, different libraries would cover clearly defined domains. They would not overlap in functionality and whatever generic functionality is missing would be added to the MPLL Standard Library. Although MPLL offers the basis for easy integration of different libraries (e.g. by namespaces), this has not been done on a larger scale.

7.2.4. Integration with GeTS

From the very beginning of the development of MPLL, the main reason for the close relation to the language GeTS was the possibility of later integrating the two languages, in order to get a spatio-temporal specification language. However, this integration process is far from being trivial. It will not consist of just merging the two implementations, but will involve a very complex adjustment of several factors. The construction of new data types, for example, will be necessary. These new types will have properties pertaining to both domains and the necessary functionality must be developed individually. This transition will not just add another dimension to space, but will raise some more complicated questions.

A. Language Reference

The language constructs are summarized and briefly explained.

A.1. Types

Data Structure Types
Data structure types are listed in table A.1.

Enumeration Types
Enumeration types are listed in in table A.2.

A.2. Arithmetics

Binary Arithmetic Operators (Def. 4.3)

operator	
addition	+
subtraction	-
multiplication	*
division	÷
modulo	%
maximum	`max`
minimum	`min`
exponentiation	`pow`

type	*description*
`Integer`	standard integers
`Float`	standard floating point numbers
`String`	strings
`Angle`	floating point numbers representing angles
`Point`	points (two-dimensional)
`Configuration`	configurations in space (`Point`, `Angle`)
`Line`	lines and polylines
`Polygon`	polygons
`List`	lists
`Interval`	fuzzy intervals
`CircularInterval`	fuzzy intervals
`ReferenceSystem`	reference system
`Route`	route (from graph/network routing)

Table A.1.: Data Structure Types

Unary Arithmetic Operators (Def. 4.4)

operator	
negation	-
casting (`Bool` $\mapsto$ `Float`)	`float`(b)
rounding	`round`(a)
rounding	`round`(a,`up/down`)

Trigonometry (Def. 4.5)

function		*result/argument*	
sine	`sin`(angle)	radian	`Float` $\mapsto$ `Float`
cosine	`cos`(angle)	radian	`Float` $\mapsto$ `Float`
inverse sine	`asin`(angle)	radian	`Float` $\mapsto$ `Float`
inverse cosine	`acos`(angle)	radian	`Float` $\mapsto$ `Float`
sine	`sind`(angle)	degree	`Float` $\mapsto$ `Float`
cosine	`cosd`(angle)	degree	`Float` $\mapsto$ `Float`
inverse sine	`asind`(angle)	degree	`Float` $\mapsto$ `Float`
inverse cosine	`acosd`(angle)	degree	`Float` $\mapsto$ `Float`

Comparisons

<, <=, >, >= (Def. 4.6).
==, != (Def. 4.7).

type	*possible values*
`Bool`	`true`, `false`
`Side`	`left`, `right`
`PosNeg`	`positive`, `negative`
`UpDown`	`up`, `down`
`ForwardBackward`	`forward`, `backward`
`InsideOutside`	`inside`, `outside`
`CarDir`	`north`, `northeast`, `east`, `southeast`, `south`, `southwest`, `west`, `northwest`, `N`, `NNE`, `NE`, `ENE`, `E`, `ESE`, `SE`, `SSE`, `S`, `SSW`, `SW`, `WSW`, `W`, `WNW`, `NW`, `NNW`
`EgoDir`	`front`, `left_front`, `left`, `left_back`, `back`, `right_back`, `right`, `right_front`
`EgoDirD`	`in_front_of`, `left_of`, `behind`, `right_of`
`Distance`	`very_close`, `close`, `commensurate`, `far`, `very_far`
`RSType`	`cartesian`, `geospherical`
`Region`	`core`, `kernel`, `support`, `maximum`
`Hull`	`core`, `kernel`, `support`, `maximum`, `crisp`, `monotone`, `convex`
`Fuzzify`	`linear`, `gaussian`
`SDVersion`	`Kleene`, `Lukasiewicz`, `Goedel`

Table A.2.: Reference: Enumeration Types

A.3. Boolean Operators

- (complement), `and` or '&&' (conjunction), `or` or '||' (disjunction), `xor` or '^' (exclusive or) (Def. 4.8).

A.4. Control Constructs

`if` c `then` a `else` b (Def. 4.9).

`case` $C_1 : E_1, \ldots, C_n : E_n$ `else` D (Def. 4.10).

`while` c $\{E_1, \ldots, E_n\}$ D (Def. 4.11).

`Let` *variable* $=$ *expression*1 `in` *expression*2 (local binding) (Def. 4.12).

$x := E$ (assignment) (Def. 4.13).

A.5. Points

`Point`(Ax,Ay) of type `Integer`$*$`Integer`$\mapsto$`Point` accepts a tuple of coordinates specified as `Angle` values.

Predicates on Points

`xCoordinate`(P) of type `Point`$\mapsto$`Angle` returns the x coordinate of a point as an `Angle`.

`yCoordinate`(P) of type `Point`$\mapsto$`Angle` returns the y coordinate of a point as an `Angle`.

A.6. Configurations

`Configuration`(A,Ax,Ay,Bo) of type `Angle`$*$ `Angle`$*$ `Angle`$*$ `Bool`$\mapsto$`Configuration` accepts an orientation and a pair of coordinates specified as `Angle` values. Additionally, `hasOrientation` must be specified as a `Bool` value. Setting Bo to `false` only makes sense if A is not relevant and is not set or set to 0 (which is equivalent).

Predicates on Configurations

`hasOrientation`(C) of type `Configuration`$\mapsto$`Bool` returns whether the configuration C features an orientation.

`orientation`(C) of type `Configuration`$\mapsto$`Angle` returns the orientation of a configuration.

`point`(C) of type `Configuration`$\mapsto$`Point` returns the x and y coordinate of a configuration form of a `Point`.

`xCoordinate`(C) of type `Configuration`$\mapsto$`Float` returns the x coordinate of a configuration as an `Integer`.

`yCoordinate`(C) of type `Configuration`$\mapsto$`Float` returns the y coordinate of a configuration as an `Integer`.

A.7. Lines

`Line`$(P1,P2,\ldots,Pn)$ of type `Point`$*$`Point`$*\ldots*$`Point`$\mapsto$`Line` accepts a list of n points with $n \geq 2$.

Predicates on Lines

`xMin`(L) of type `Line`$\mapsto$`Angle` returns the minimum value of x coordinates of *all* points which make up the line.

yMin(L) of type Line $\mapsto$ Angle returns the minimum value of y coordinates of *all* points which make up the line.

xMax(L) of type Line $\mapsto$ Angle returns the maximum value of x coordinates of *all* points which make up the line.

yMax(L) of type Line $\mapsto$ Angle returns the maximum value of y coordinates of *all* points which make up the line.

A.8. Polygons

Polygon($P1, P2, \ldots, Pn$) of type Point $*$ Point $*\ldots*$ Point $\mapsto$ Polygon accepts a sequence of n points with $n \geq 3$.

Predicates on Polygons

xMin(R) of type Polygon $\mapsto$ Angle returns the minimum value of x coordinates of *all* points which make up the polygon.

yMin(R) of type Polygon $\mapsto$ Angle returns the minimum value of y coordinates of *all* points which make up the polygon.

xMax(R) of type Polygon $\mapsto$ Angle returns the maximum value of x coordinates of *all* points which make up the polygon.

yMax(R) of type Polygon $\mapsto$ Angle returns the maximum value of y coordinates of *all* points which make up the polygon.

points(R) of type Polygon $\mapsto$ List returns the ordered list of points which make up the polygon.

Topological Predicates on Polygons

DC(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the two polygons R and S are disconnected.

EC(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the two polygons R and S are externally onnected.

PO(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the two polygons R and S overlap partially.

TPP(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the polygon R is a tangential proper part of the polygon S.

NTPP(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the polygon R is a non-tangential proper part of the polygon S.

TPPinverse(R,S) of type Polygon $*$ Polygon $\mapsto$ Bool checks whether the polygon S is a tangential proper part of the polygon R.

NTPPinverse(R,S) of type Polygon∗Polygon ↦ Bool checks whether the polygon S is a non-tangential proper part of the polygon R.

EQ(R,S) of type Polygon∗Polygon ↦ Bool checks whether the two polygons R and S are equal.

A.9. Lists

emptyList(T) of type T ↦ List constructs an empty list for objects of type T.

$\{T_1,\ldots,T_n\}$ of type T∗...∗T ↦ List constructs a list containing objects $T_1,\ldots,T_n$ of type T.

Predicates on Lists

head(L) of type List<T> ↦ T returns the first element of the list L, it is of type T.

tail(L) of type List<T> ↦ List<T> return the list L without the first element, i.e. without the head.

length(L) of type List<T> ↦ Integer returns the number of elements of the list as an integer.

sublist(L,N,M) of type List<T>∗Integer∗Integer ↦ List<T> returns the sublist from at a specific index N to a specific index M.

prefix(L,N) of type List<T>∗Integer ↦ List<T> returns the sublist from the first element of the list L to a specific index N.

suffix(L,N) of type List<T>∗Integer ↦ List<T> returns the sublist from a specific index N to the end of the list L.

append(T,L) of type T∗List<T> ↦ List<T> appends an element of type T as the first element of the list L.

append(L,L) of type List<T>∗List<T> ↦ List<T> concatenates two lists L and M.

split(L,N) of type List<T>∗Integer ↦ List<List<T>> splits a list L at a given position into two lists returning a list containing both lists.

filter(*Condition*,L) of type (T ↦ Bool)∗List<T> ↦ List<T> returns a list of all elements of type T in L for which condition(T) holds true.

map(*Function*,L) of type (T ↦ S)∗List<T> ↦ List<S> returns a list $\{Function(T_1),\ldots,Function(T_n)\}$ of type S.

A.10. Reference Systems

ReferenceSystem*cartesian*/*geospherical*,N,N,N,N,B,B) of type RSType∗Integer∗Integer∗Integer∗Integer∗ Bool∗ Bool ↦ ReferenceSystem

accepts a reference system type RSType, four Float or Integer coordinates which define minimum x, maximum y, maximum x and minimum y coordinates and two Bool values which mark vertical and horizontal wraparound.

ReferenceSystem($cartesian/geospherical, P, P, B, B$) of type RSType * Point * Point * Bool * Bool $\mapsto$ ReferenceSystem accepts a reference system type RSType, two Point variables defining upper left and lower right coordinates, and two Bool values which mark vertical and horizontal wraparound.

Predicates on Reference Systems

isCartesian(RS) of type ReferenceSystem $\mapsto$ Bool returns whether the reference system is of RSType cartesian.

isGeospherical(RS) of type ReferenceSystem $\mapsto$ Bool returns whether the reference system is of RSType geospherical.

isOfType($cartesian/geospherical, RS$) of type RSType * ReferenceSystem $\mapsto$ Bool returns whether the reference system is of the given RSType (cartesian or geospherical).

hasVerticalWrap(RS) of type ReferenceSystem $\mapsto$ Bool returns whether the reference system features vertical wraparound.

hasHorizontalWrap(RS) of type ReferenceSystem $\mapsto$ Bool returns whether the reference system features horizontal wraparound.

xMin(RS) of type ReferenceSystem $\mapsto$ Bool returns the minimum value of x coordinates which are allowed within the reference system.

yMin(RS) of type ReferenceSystem $\mapsto$ Bool returns the minimum value of y coordinates which are allowed within the reference system.

xMax(RS) of type ReferenceSystem $\mapsto$ Bool returns the maximum value of x coordinates which are allowed within the reference system.

yMax(RS) of type ReferenceSystem $\mapsto$ Bool returns the maximum value of y coordinates which are allowed within the reference system.

upperLeft(RS) of type ReferenceSystem $\mapsto$ Point returns the topmost leftmost point of the reference system in form of a Point.

lowerRight(RS) of type ReferenceSystem $\mapsto$ Point returns the lowest rightmost point of the reference system in form of a Point.

A.11. Intervals

[] of type Interval (empty interval)

[t1,t2] of type Interval (new crisp interval from t1 to t2)

pushback(I,*time*,*value*) of type Interval∗Time∗Float ↦ Void adds (*time*, *value*) to the membership function of the interval (Def. 4.36).

Set Operations on Intervals

complement(I)
  of type Interval ↦ Interval
complement(I,λ)
  of type Interval∗Float ↦ Interval
complement(I,*negation_function*)
  of type Interval∗(Float ↦ Float) ↦ Interval
(see Def. 4.37)

union(I,J)
  of type Interval∗Interval ↦ Interval
union(I,J,β)
  of type Interval∗Interval∗Float ↦ Interval
union(I,J,*co_norm*)
  of type *Interval*∗Interval∗(Float∗Float ↦ Float) ↦ Interval
(see Def. 4.38)

intersection(I,J)
  of type Interval∗Interval ↦ Interval
intersection(I,J,γ)
  of type Interval∗Interval∗Float ↦ Interval
intersection(I,J,*norm*))
  of type Interval∗Interval∗(Float∗Float ↦ Float) ↦ Interval
(see Def. 4.39)

setdifference(I,J)
  of type Interval∗Interval ↦ Interval
setdifference(I,J,*version*)
  of type Interval∗Interval∗SDVersion ↦ Interval
setdifference(I,J,*intersection*,*complement*)
  of type Interval∗Interval∗(Interval∗Interval ↦ Interval)∗
    (Interval ↦ Interval) ↦ Interval
(see Def. 4.40)

Predicates on Intervals

isCrisp(I)	Interval ↦ Bool
isCrisp(I,left/right)	Interval∗Side ↦ Bool
isEmpty(I)	Interval ↦ Bool
isConvex(I)	Interval ↦ Bool
isMonotone(I)	Interval ↦ Bool
isInfinite(I)	Interval ↦ Bool
isInfinite(I,left/right)	Interval∗Side ↦ Bool

(see Def. 4.41)

`during`(*time*,*I*,`core/kernel/support`) of type `Time*Interval*IntvRegion` $\mapsto$ `Bool` checks whether *time* is in the corresponding region of the interval *I* (Def. 4.43).

`isSubset`(*I*,*J*,`core/kernel/support`) of type `Interval*Interval*IntvRegion` $\mapsto$ `Bool` checks whether the corresponding region of *I* is a subset of the corresponding region of *J* (Def. 4.43).

`doesOverlap`(*I*,*J*,`core/kernel/support`) of type `Interval*Interval*IntvRegion` $\mapsto$ `Bool` checks whether the corresponding region of *I* overlaps with the corresponding region of *J* (Def. 4.43).

`member`(*time*,*I*) of type `Time*Interval` $\mapsto$ `Float` (membership function) (Def. 4.44).

`size`(*I*) of type `Interval` $\mapsto$ `Time` (size of the interval) (Def. 4.46)

`size`(*I*,*region*) of type `Interval*IntvRegion` $\mapsto$ `Time` (size of the corresponding region of the interval) (Def. 4.46)

`size`(I,t_1,t_2) of type `Interval*Time*Time` $\mapsto$ `Time` (size of the interval between t_1 and t_2) (Def. 4.46)

`point`(*I*,*side*,*region*) of type `Interval*Side*PointRegion` $\mapsto$ `Time` (position of the corresponding end of the region) (Def. 4.47).

`centerPoint`(I,n,m) of type `Interval*Integer*Integer` $\mapsto$ `Time` (*n*-*m* centre point) (Def. 4.48).

Manipulation of Intervals

`shift`(I,t) of type `Interval*Time` $\mapsto$ `Interval` shifts the interval by the given time (Def. 4.49).

`cut`(I,t_1,t_2) of type `Interval*Time*Time` $\mapsto$ `Interval` (extracts the part of *I* between t_1 and t_2) (Def. 4.50).

`hull`(*I*,`core/support/kernel/crisp/monotone/convex`) of type `Interval* Hull` $\mapsto$ `Interval` (construction of the corresponding hull) (Def. 4.51).

`invert`(*I*) of type `Interval` $\mapsto$ `Interval` inverts the membership function (Def. 4.52).

`scaleup`(*I*) of type `Interval` $\mapsto$ `Interval` scales the membership function up to maximal value 1 (Def. 4.53).

`times`(*I*,*f*) of type `Interval*Float` $\mapsto$ `Interval` multiplies the membership function of *I* with *f* (Def. 4.54).

`exp`(*I*,*e*) of type `Interval*Float` $\mapsto$ `Interval` exponentiates the membership function of *I* with *e* (Def. 4.54).

`extend`(I,`positive/negative`) of type `Interval*PosNeg`$\mapsto$`Interval` extends I to the infinity (Def. 4.55).

`extend`(I,*length*,*side*) of type `Interval*Time*Side`$\mapsto$`Interval` extends or shrinks I (Def. 4.56).

`integrate`(I,`positive/negative`) of type `Interval*PosNeg`$\mapsto$`Interval` integrates the membership function (Def. 4.57).

`fuzzify`(I,`linear/gaussian`,`left/right`,*increase*,*offset*)
of type `Interval`,`Fuzzify`,`Side`,`Float`,`Float`$\mapsto$`Interval`
`fuzzify`(I,`linear/gaussian`,`left/right`,$x1$,$x2$,*offset*)
of type `Interval`,`Fuzzify`,`Side`,`Time`,`Time`,`Time`$\mapsto$`Interval` (Def. 4.58)
fuzzifies the interval at the given side with the given fuzzification function.

`integrateSymmetric`(I,J,*simple*)
of type`Interval*Interval*Bool`$\mapsto$`Float` and
`integrateAsymmetric`(I,J)
of type `Interval*Interval`$\mapsto$`Float`
symmetric or asymmetric integration of the membership function of I over the membership function of J (Def. 4.59).

`MaximizeOverlap`(I,J,EJ,D) of type `Interval*Interval*Interval*(Interval*Interval`$\mapsto$`Float)`$\mapsto$`Float` (Def. 4.60)

B. Application Programming Interface Reference

The C++ API of the MPLL language is as follows:

MPLL functions are realised as a class `Function` in a namespace `MPLL`. They can be defined, they can be applied to arguments, and some information about them can be retrieved.

Definition:
A new MPLL function can be created with an ordinary constructor:

`fct = new Function(definition).`

The `definition` is a string representation of the definition, optionally followed by the keyword `explanation` and some text. The explanation can be retrieved just by `fct->explanation`.

The definition is parsed and compiled. Parsing or compilation errors can be obtained by `fct->getError()`. The function `fct->noError()` checks whether there was a parsing or compilation error.

Information about Functions:
The function definitions can be obtained in different versions:

`fct->callString()` returns the function call as string

`fct->typeString()` returns the function type as string

`fct->definitionString()` returns the function definition with line numbering as string.

`fct->codeString()` returns the abstract machine code as string.

The following example illustrates the use of the function `codeString()` and shows the output that is generated. This example pertains to the MPLL function `isPolygon`.

Example B.1 (for `codeString()`) The code string for the function

```
isPolygon(Multiline(newList(Configuration(1,1),
                            Point(2,1), Point(2,2),
                            Point(1,2), Point(1,1))))
```

is shown in the following listing:

```
0: 1[-1,1,Integer]
1: 1[-1,1,Integer]
2: Configuration(Integer*Integer->Configuration)
3: 2[-1,2,Integer]
4: 1[-1,1,Integer]
5: Point(Integer*Integer->Point)
6: 2[-1,2,Integer]
7: 2[-1,2,Integer]
8: Point(Integer*Integer->Point)
9: 1[-1,1,Integer]
10: 2[-1,2,Integer]
11: Point(Integer*Integer->Point)
12: 1[-1,1,Integer]
13: 1[-1,1,Integer]
14: Point(Integer*Integer->Point)
15: List(Configuration*Point*Point*Point*Point->
         List<Point>)
16: Multiline(List<Point>->Multiline)
17: isPolygon(Multiline->Bool)
```

It should be fairly obvious what this means. For example, line 0, `1[-1,1,Integer]` means that the parameter `I` at parameter position 1 of type `Integer` is pushed onto the stack. Line 2: `Configuration(Integer*Integer->Configuration)` means that the `Configuration` constructor pops its (two) arguments from the stack and pushes the result onto the stack again. Lines 3 to 14 are treated analoguously. Line 15: pops the arguments from the stack and constructs a list (performing an implicit cast from `Configuration` to `Point` in the process. The list ist then pushed back onto the stack. `Multiline` then pops the list, constructs a polygon from it, and pushes the polygon back onto the stack. Finally, `isPolygon` performs the check, whether the argument on top of the stack is a valid polygon in MPLL terms, i.e. it must consist of more than 3 points, the first and the last of which have to have identical coordinates. ■

Note that the actual computations, are done with compiled machine code. The commands of the MPLL abstract machine are only used to control the invocation of this machine code.

Auxiliary Classes and Types

The data types of MPLL are represented as a class `Type` in the namespace `MPLL`. They can be basic data types or compound types. The most important API method for types is `toString`. Most other methods are for internal use.

The data which are manipulated by a MPLL function are comprised into a union type. Without further explanation we just list the definition.

```
union MPLLValue {
  Function*              lFunction;
  int                    Integer;
  float                  Float;
  bool                   Bool;
  string*                String;
  MPLL::Interval*        Interval;
  MPLL::Angle*           Angle;
  MPLL::Point*           Point;
  MPLL::Configuration*   Configuration;
  MPLL::Line*            Line;
  MPLL::List*            List;
  MPLL::Polygon*         Polygon;
  };
```

Application:
There are two application functions:

```
pair<Type*, MPLLValue>
   apply(vector<pair<Type*, MPLLValue> >\& values)
```

can be used to apply the function to a vector of parameters. The result is a pair consisting of the result type and the result value.

The other method:

```
pair<Type*, MPLLValue>
   apply(const string& arguments,
         const vector<FuTIRe::Interval*>& intervals)
```

can be used to apply the function to a string representation of the parameters. Intervals are represented as non-negative integers. The integers are used as indices in the given vector of interval pointers. The result is again a type-value pair.

C. Selected Code Samples

The first section of this chapter contains a listing of all constructs in the complete MPLL Standard Library. In order to improve readability and to provide a concise overview of the Standard Library, the author refrained in most cases from giving full flexed function explanations which are primarily important for the practical application of MPLL.

The second section shows some selected code samples of the MPLL implementation to provide an exemplary understanding of the implementation of the scanner and parser of the language and of different data types and functions.

C.1. The MPLL Standard Library

This section contains the complete sources of the MPLL Standard Library, as they are loaded at start time.

C.1.1. Types

```
MPLL Standard Library - Types;

Constants
************************************************************************
'Real = 0
explanation:
    (put explanation here);'

'Grd = 1
explanation:
    (put explanation here);'

'Deg = 2
explanation:
    (put explanation here);'

'Rad = 3
explanation:
    (put explanation here);'

'Pi = 3.141592653
```

```
22 explanation:
     (put explanation here);'
24
   'defCore = 25
26 explanation:
     (put explanation here);'
28
   'defSupport = 50
30 explanation:
     (put explanation here);'
32
   'defMod(type) = case (type==Real): 0,
34                                     (type==Grd): 400,
                                       (type==Deg): 360,
36                                     (type==Grd): (2*Pi),
                               else 0
38 explanation:
     (put explanation here);'
40
   'carDir(Dir) = case (Dir==north): 0,
42                                   (Dir==northeast): 50,
                                     (Dir==east): 100,
44                                   (Dir==southeast): 150,
                                     (Dir==south): 200,
46                                   (Dir==southwest): 250,
                                     (Dir==west): 300,
48                                   (Dir==northwest): 350,
                              else 0
50 explanation:
     (put explanation here);'
52
   'carDir(Dir) = case (Dir==N): 0,
54                                   (Dir==NE): 50,
                                     (Dir==E): 100,
56                                   (Dir==SE): 150,
                                     (Dir==S): 200,
58                                   (Dir==SW): 250,
                                     (Dir==W): 300,
60                                   (Dir==NW): 350,
                              else 0
62 explanation:
     (put explanation here);'
64
   'carDir(Dir) = case (Dir==front): 0,
66                                   (Dir==right_front): 50,
                                     (Dir==right): 100,
68                                   (Dir==right_back): 150,
                                     (Dir==back): 200,
70                                   (Dir==left_back): 250,
                                     (Dir==left): 300,
72                                   (Dir==left_front): 350,
                              else 0
74 explanation:
     (put explanation here);'
76
   'carDir(Dir) = case (Dir==behind): 0,
78                                   (Dir==right_of): 100,
```

```
                                    (Dir==in_front_of): 200,
                                    (Dir==left_of): 300,
                                else 0
explanation:
    (put explanation here);'

'distance(Dist) = case (Dist==): 10,
                                       (Dist==): 30,
                                       (Dist==): 60,
                                       (Dist==): 100,
                                       (Dist==): 150,
                                  else 0
explanation:
    (put explanation here);'

Math
*************************************************************************
'root(Float b, Float e) =
    pow(b, 1.0/e)
explanation:
    (put explanation here);'

'sqr(Float f) =
    pow(f, 2.0)
explanation:
    (put explanation here);'

'sqrt(Float f) =
    root(f,2.0)
explanation:
    (put explanation here);'

Angles (Constructors)
*************************************************************************
'Angle(Float v, Float a, Float b) =
    newAngle(v,a,b)
explanation:
    (put explanation here);'

'Angle(Float f, Integer type) =
    newAngle(f,-defMod(type),defMod(type))
explanation:
    (put explanation here);'

'Angle() = Angle(0.0,Grd())'
'AngleReal(Float f) = Angle(f,Real())
explanation:
    (put explanation here);'

'AngleReal(Float angle, Float mod) =
    Angle(angle, -mod, -mod)
explanation:
    (put explanation here);'

'AngleGrd(Float f) = Angle(f,Grd())
explanation:
    (put explanation here);'
```

```
136
   'AngleGrd(Float angle, Float mod) =
138    AngleReal(angle, mod)
   explanation:
140    (put explanation here);'

142 'AngleDeg(Float f) = Angle(f,Deg())
   explanation:
144    (put explanation here);'

146 'AngleDeg(Float angle, Float mod) =
       AngleGrd(degToGrd(angle), degToGrd(mod))
148 explanation:
       (put explanation here);'
150
   'AngleRad(Float f) = Angle(f,Rad())
152 explanation:
       (put explanation here);'
154
   'AngleRad(Float angle, Float mod) =
156    AngleGrd(radToGrd(angle), radToGrd(mod))
   explanation:
158    (put explanation here);'

160 'Angle(Configuration C, Configuration D, Configuration E) =
       Angle(bearing(D,E)-bearing(D,C))
162 explanation:
       (put explanation here);'
164
   Angles (Predicates)
166 ************************************************************************
   'isPositive(Angle A) =
168    not(isNegative(A))
   explanation:
170    (put explanation here);'

172 'isRightTurn(Angle A) =
       not(isNegative(A))
174 explanation:
       (put explanation here);'
176
   'isLeftTurn(Angle A) =
178    isNegative(A)
   explanation:
180    (put explanation here);'

182 'minMax(Angle A) =
       Interval(min(A),max(A))
184 explanation:
       (put explanation here);'
186
   'isNegative(Angle a) = (grad(a) < 0)
188 explanation:
       (put explanation here);'
190
   'degToGrd(Float i) = i*10/9
192 explanation:
```

```
    (put explanation here);'
194
  'degToRad(Float i) = i*PI()/180
196 explanation:
    (put explanation here);'
198
  'grdToDeg(Float i) = i*9/10
200 explanation:
    (put explanation here);'
202
  'grdToRad(Float i) = i*PI()/200
204 explanation:
    (put explanation here);'
206
  'radToGrd(Float i) = i*200/PI()
208 explanation:
    (put explanation here);'
210
  'radToDeg(Float i) = i*180/PI()
212 explanation:
    (put explanation here);'
214
  'degree(Angle a) = grdToDeg(grad(a))
216 explanation:
    (put explanation here);'
218
  'radian(Angle a) = grdToRad(grad(a))
220 explanation:
    (put explanation here);'
222
  'modulus(Angle a) = max(a)-min(a)
224 explanation:
    (put explanation here);'
226
  'equals(Angle a, Angle b) =
228   (grad(a)==grad(b)) and (min(a)==min(b)) and (max(a)==max(b))
  explanation:
230   (put explanation here);'

232 Points (Constructors)
  ************************************************************************
234 'Point() =
      Point(Angle(),Angle())
236 explanation:
    (put explanation here);'
238
  'Point(Angle x, Angle y) =
240   newPoint(x,y)
  explanation:
242   (put explanation here);'

244 'Point() =
      Point(Angle(),Angle())
246 explanation:
    (put explanation here);'
248
  'Point(Float x, Float y) =
```

```
250    Point(AngleGrd(x),AngleGrd(y))
   explanation:
252    (put explanation here);'

254 Points (Predicates)
   ************************************************************************
256 'distance(Point p1, Point p2) =
          let deltaX = grad(getX(p2)) - grad(getX(p1)) in
258       let deltaY = grad(getY(p2)) - grad(getY(p1)) in
          sqrt(deltaX + deltaY)'
260 'vpl(Point p1, Point p2, Point p3) = (p2-p1)*(p3-p1)
   explanation:
262    (put explanation here);'

264 'isLeft    (Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) >  0)
   explanation:
266    (put explanation here);'

268 'isRight   (Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) <  0)
   explanation:
270    (put explanation here);'

272 'isColinear(Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) == 0)
   explanation:
274    (put explanation here);'

276 'equals(Point a, Point b) =
      equals(getX(a),getX(b)) and equals(getY(a),getY(b))
278 explanation:
      (put explanation here);'
280
   Configurations (Constructors)
282 ************************************************************************
   'Configuration() =
284    Configuration(Angle(),Angle(),Angle(),false)
   explanation:
286    (put explanation here);'

288 'Configuration(Angle a, Angle b, Angle c, Bool d) =
      newConfiguration(a,b,c,d)
290 explanation:
      (put explanation here);'
292
   'Configuration(Angle b, Angle x, Angle y) =
294    Configuration(b,x,y,true)
   explanation:
296    (put explanation here);'

298 'Configuration(Angle x, Angle y) =
      Configuration(Angle(),x,y,false)
300 explanation:
      (put explanation here);'
302
   'Configuration(Angle b, Point p, Bool t) =
304    Configuration(b,getX(p),getY(p),t)
   explanation:
306    (put explanation here);'
```

```
'Configuration(Angle b, Point p) =
   Configuration(b,p,true)
explanation:
   (put explanation here);'

'Configuration(Angle b) =
   Configuration(b,Point(),true)
explanation:
   (put explanation here);'

'Configuration() =
   Configuration(Angle(),Point())
explanation:
   (put explanation here);'

'Configuration(Point p) =
   Configuration(getX(p),getY(p))
explanation:
   (put explanation here);'

'Configuration(Float x, Float y) =
   Configuration(Point(x,y))
explanation:
   (put explanation here);'

Configurations (Predicates)
**************************************************************************

Lines (Constructors)
**************************************************************************
'elem(List<T> list, Integer i) = element(list, i)
explanation:
   (put explanation here);'

'Multiline(List<Point> list) = newMultiline(list)
explanation:
   (put explanation here);'

'isPolygon(Multiline ml) =
   (size(ml) > 3) and equals(head(ml),elem(ml,size(ml)-1))
explanation:
   (put explanation here);'

Lines (Predicates)
**************************************************************************
'upperLeftBB(Line L) =
   Point(Angle(xMin(P)),Angle(yMin(P)))
explanation:
   (put explanation here);'

'lowerRightBB(Line L) =
   Point(Angle(xMax(P)),Angle(yMax(P)))
explanation:
   (put explanation here);'

Polygons (Constructors)
```

```
364 ************************************************************************
    'Polygon(List<Point> list) = newPolygon(list)
366 explanation:
       (put explanation here);'
368
    Polygons (Predicates)
370 ************************************************************************
    'upperLeftBB(Point P) =
372    Point(Angle(xMin(P)),Angle(yMin(P)))
    explanation:
374    (put explanation here);'

376 'lowerRightBB(Point P) =
       Point(Angle(xMax(P)),Angle(yMax(P)))
378 explanation:
       (put explanation here);'
380
    'isTriangle(Polygon p) = (size(p)==4)'
382
    'Float:area2(List<Polygon> lp)
384 explanation:
       (put explanation here);'
386
    'area(Polygon p) =
388         if (isTriangle(p))
            then vpl(head(p), element(p,1), element(p,2))/2.0
390         else let tri=triangulate(p) in area2(tri)
    explanation:
392    (put explanation here);'

394 'area2(List<Polygon> lp) =
            if empty(lp)
396         then 0.0
            else area(head(lp)) + area2(tail(lp))
398 explanation:
       (put explanation here);'
400
    'Point:com2(List<Polygon> lp)
402 explanation:
       (put explanation here);'
404
    'Point:com3(List<Polygon> lp)
406 explanation:
       (put explanation here);'
408
    'com(Polygon p) =
410         if (isTriangle(p))
            then (head(p) + element(p,1) + element(p,2))*(1.0/3.0)
412         else let tri=triangulate(p) in com2(tri)
    explanation:
414    (put explanation here);'

416 'com2(List<Polygon> lp) = com3(lp)*(1.0/size(lp))
    explanation:
418    (put explanation here);'

420 'com3(List<Polygon> lp) =
```

```
         if empty(lp)
422      then Point()
         else com(head(lp)) + com3(tail(lp))
424 explanation:
    (put explanation here);'
426
  Circular Intervals (Constructors)
428 ************************************************************************
  'CircularInterval() =
430    CircularInterval(Angle(0),defMod(Grd))
  explanation:
432    (put explanation here);'

434 'CircularInterval(Float Fvalue) =
    CircularInterval(Angle(0),defMod(Grd),Fvalue)
436 explanation:
    (put explanation here);'
438
  Circular Intervals (Predicates)
440 ************************************************************************

442 Lists (Constructors)
  ************************************************************************
444 'List<T>:prefix(List<T> list, Integer index) =
         if (index == 0)
446      then newList(head(list))
         else append2(head(list), prefix(tail(list),index-1))
448 explanation:
    (put explanation here);'
450
  'List<T>:suffix(List<T> list, Integer index) =
452      if (index == 0)
         then list
454      else suffix(tail(list), index-1)
  explanation:
456    (put explanation here);'

458 'List<T>:reverse(List list) =
    append(reverse(tail(list)),newList(head(list)))
460 explanation:
    (put explanation here);'
462
  Lists (Predicates)
464 ************************************************************************
  'head(List<T> x) =
466    element(x,0)
  explanation:
468    (put explanation here);'

470 'empty(List l) =
    (size(l) == 0)
472 explanation:
    (put explanation here);'
474
  'append2(T h, List<T> t) =
476    append(newList(h),t)
  explanation:
```

```
478    (put explanation here);'

480 'subList(List list, Integer from, Integer to) =
       suffix(prefix(list,to),from)
482 explanation:
       (put explanation here);'
484
    'remove(List list, Integer index) =
486          if (index==0)
             then tail(list)
488          else
                if (index==(size(index)-1))
490             then prefix(list, (size(list)-2))
                else append(prefix(list,(index-1)) , suffix(list,(index+1)))
492 explanation:
       (put explanation here);'
494
    'length(List list) =
496    if empty(list) then 0 else 1 + length(tail(list))
    explanation:
498    (put explanation here);'

500 'begin(List list, Integer b) =
       sublist(list, 0, b)
502 explanation:
       (put explanation here);'
504
    'end(List list, Integer a) =
506    sublist(list, a, 0)
    explanation:
508    (put explanation here);'

510 'append(List a, List b) =
       if empty(a) then b else append(head(a), append(tail(a), b))
512 explanation:
       (put explanation here);'
514
    'filter(Point->Bool cond, List list) =
516    if empty(list) then
          list
518    else
          if cond(head(list)) then
520          append(head(list), filter(cond, tail(list)))
          else
522          filter(cond, tail(list))
    explanation:
524    (put explanation here);'

526 'map(Point->Point trans, List list) =
       if empty(list) then
528       list
       else
530       append(trans(head(list)), map(trans, tail(list)))
    explanation:
532    (put explanation here);'
```

C.1.2. Functions

```
MPLL Standard Library - Functions;

Translation
************************************************************************
'translatePolar(Configuration C, Angle Adir, Angle Adist) =
   translateCartesian(C,sin(Adir)*Adist,cos(Adir)*Adist)
explanation:
   Moves the configuration C by the amount Adist into the direction Adir;'

'movePolar(Configuration C, Angle Adir, Angle Adist) =
   translatePolar(C,Adir,Adist)
explanation:
   Moves the configuration C by the amount Adist into the direction
   Adir;'

'moveX(Configuration C, Angle Adist) =
   translateCartesian(C,Adist,Angle(0))
explanation:
   Moves the configuration C by the amount A along the x-axis;'

'moveY(Configuration C, Angle Adist) =
   translateCartesian(C,Angle(0),Adist)
explanation:
   Moves the configuration C by the amount A along the y-axis;'

'move(Configuration C, Angle Ax, Angle Ay) =
   translateCartesian(C,Ax,Ay)
explanation:
   Moves the configuration C by the amount Ax along the x-axis and by the
   amount Ay along the y-axis;'

'move(Configuration C, Configuration D, Angle Adist) =
   translatePolar(C,bearing(C,D),Adist)
explanation:
   Moves the configuration C by the amount A into the direction of the
   bearing from C to D;'

'move(Configuration C, CarDir Dir, Angle Adist) =
   translatePolar(C,carDir(Dir),Adist)
explanation:
   Moves the configuration C by the amount A into the direction
   denoted by the enumeration type Dir;'

'moveTo(Configuration C, Configuration D) =
   Configuration(orientation(C),point(D))
explanation:
   Moves the configuration C to the position denoted by the
   coordinates of point P;'

'moveTo(Configuration C, Point P) =
   Configuration(orientation(C),P)
explanation:
   Moves the configuration C to the position denoted by the
   coordinates of point P;'
```

```
Rotation
*************************************************************************
'rotateLeft(Configuration C, Angle A) = rotate(C,-abs(A))
explanation:
   Rotates the orientation of the configuration C to the left by
   the amount A;'

'rotateCCW(Configuration C, Angle A) = rotate(C,-abs(A))
explanation:
   Rotates the orientation of the configuration C counterclockwise
   by the amount A;'

'rotateRight(Configuration C, Angle A) = rotate(C,abs(A))
explanation:
   Rotates the orientation of the configuration C to the right by
   the amount A;'

'rotateCW(Configuration C, Angle A) = rotate(C,abs(A))
explanation:
   Rotates the orientation of the configuration C clockwise by
   the amount A;'

'rotateLeft(Line L, Angle A) = rotate(L,-abs(A))
explanation:
   Rotates the all points of the line L around the origin of the
   reference system to the left by the amount A;'

'rotateCCW(Line L, Angle A) = rotate(L,-abs(A))
explanation:
   Rotates the all points of the line L around the origin of the
   reference system counterclockwise by the amount A;'

'rotateRight(Line L, Angle A) = rotate(L,abs(A))
explanation:
   Rotates the all points of the line L around the origin of the
   reference system to the right by the amount A;'

'rotateCW(Line L, Angle A) = rotate(L,abs(A))
explanation:
   Rotates the all points of the line L around the origin of the
   reference system clockwise by the amount A;'

'rotateLeft(Polygon R, Angle A) = rotate(R,-abs(A))
explanation:
   Rotates the all points of the polygon R around the origin of the
   reference system to the left by the amount A;'

'rotateCCW(Polygon R, Angle A) = rotate(R,-abs(A))
explanation:
   Rotates the all points of the polygon R around the origin of the
   reference system counterclockwise by the amount A;'

'rotateRight(Polygon R, Angle A) = rotate(R,abs(A))
explanation:
   Rotates the all points of the polygon R around the origin of the
   reference system to the right by the amount A;'
```

```
112
   'rotateCW(Polygon R, Angle A) = rotate(R,abs(A))
114 explanation:
      Rotates the all points of the polygon R around the origin of the
116    reference system clockwise by the amount A;'

118 Turning (rotation of orientation)
   ************************************************************************
120 'turnLeft(Configuration C, Angle A) = turn(C,-abs(A))
   explanation:
122    (put explanation here);'

124 'turnCCW(Configuration C, Angle A) = turn(C,-abs(A))
   explanation:
126    (put explanation here);'

128 'turnRight(Configuration C, Angle A) = turn(C,abs(A))
   explanation:
130    (put explanation here);'

132 'turnCW(Configuration C, Angle A) = turn(C,abs(A))
   explanation:
134    (put explanation here);'

136 'turnTo(Configuration C, Angle A) =
      Configuration(A,point(C))
138 explanation:
      (put explanation here);'
140
   'turnTo(Configuration C, Configuration D) =
142    Configuration(orientation(D),point(C))
   explanation:
144    (put explanation here);'

146 'turnAround(Configuration C) = if(hasOrientation(C))
      then Configuration(inverse(orientation(C)),point(C),true)
148    else C
   explanation:
150    (put explanation here);'

152 Bearing with allocentric cardinal direction (Dir is of type CarDir)
   ************************************************************************
154 'bearing(CarDir Dir, Configuration C, Configuration D) =
      (bearing(C,D) == direction(Dir))
156 explanation:
      (put explanation here);'
158
   'bearingF(CarDir Dir, Configuration C, Configuration D) =
160    member(bearing(C,D),directionF(Dir))
   explanation:
162    (put explanation here);'

164 'bearingF(CarDir Dir, Configuration C,
                        Configuration D, Float Fth) =
166    member(bearing(C,D),directionF(Dir),Fth)
   explanation:
168    (put explanation here);'
```

```
'bearing(CarDir Dir, Configuration C, Line L) =
   (bearing(C,L) == direction(Dir))
explanation:
   (put explanation here);'

'bearingF(CarDir Dir, Configuration C, Line L) =
   member(bearing(C,L),directionF(Dir))
explanation:
   (put explanation here);'

'bearingF(CarDir Dir, Configuration C, Line L, Float Fth) =
   member(bearing(C,L),directionF(Dir),Fth)
explanation:
   (put explanation here);'

Bearing with egocentric cardinal direction (Dir is of type EgoDir)
*************************************************************************
'bearing(EgoDir Dir, Configuration C, Configuration D) =
   (bearing(C,D) == direction(Dir,C))
explanation:
   (put explanation here);'

'bearingF(EgoDir Dir, Configuration C, Configuration D) =
   member(bearing(C,D),directionF(Dir,C))
explanation:
   (put explanation here);'

'bearingF(EgoDir Dir, Configuration C,
                      Configuration D, Float Fth) =
   member(bearing(C,D),directionF(Dir,C),Fth)
explanation:
   (put explanation here);'

Bearing with egocentric cardinal direction - deictic setting
(Dir is of type EgoDirD)
*************************************************************************
'bearing(EgoDirD Dir, Configuration C,
                      Configuration D, Configuration E) =
   (bearing(C,D) == direction(Dir,bearing(E,C)))
explanation:
   (put explanation here);'

'bearingF(EgoDirD Dir, Configuration C,
                       Configuration D, Configuration E) =
   member(bearing(C,D),directionF(Dir,bearing(E,C)))
explanation:
   (put explanation here);'

'bearingF(EgoDirD Dir, Configuration C,
                       Configuration D, Configuration E, Float Fth) =
   member(bearing(C,D),directionF(Dir,bearing(E,C)),Fth)
explanation:
   (put explanation here);'

Bearing Region
*************************************************************************
```

```
'bearingRegion(CarDir Dir, Configuration C, Angle Adist) =
   if (hasOrientation(C))
      then
         move(C,bearing(C,bearing(in\_front/behind,C)),unitlength(RS))
      else
         throwExcepton("bearingRegion: noOrientation")
explanation:
   (put explanation here);'

Inverse
************************************************************************
'inverse(Point P) =
   Point(-xCoordinate(P),-yCoordinate(P))
explanation:
   (put explanation here);'

'inverse(Configuration C) = if(hasOrientation(C))
   then Configuration(inverse(orientation(C)),
                      inverse(point(C)),true)
   else Configuration(0,inverse(point(C)),false)
explanation:
   (put explanation here);'
```

C.2. Implementation

This section show some excerpts of the MPLL implementation. The complete source code cannot, and need not, be listed here. The following excerpts sufficiently illustrate the essential interworkings of the different components.

C.2.1. Scanner

```
%{
#include <iostream>

#include <string>
#include "MPLL.h"
#include "Parser.hh"

using namespace std;

YY_BUFFER_STATE MPLL_BUFFER;

 int* Special_Flag;
 int* MPLLRow;
 int* MPLLColumn;
%}

%option noyywrap

%%

"," {++*MPLLColumn; return ',';}
```

```
"(" {++*MPLLColumn; return '(';}
")" {++*MPLLColumn; return ')';}
"{" {++*MPLLColumn; return '{';}
"}" {++*MPLLColumn; return '}';}
"[" {++*MPLLColumn; return '[';}
"]" {++*MPLLColumn; return ']';}
"|" {++*MPLLColumn; return '|';}
"[]" {*MPLLColumn += 2; return EMPTYINTERVAL;}

"=" {++*MPLLColumn; return '=';}

"+" {++*MPLLColumn; return '+';}
"-" {++*MPLLColumn; return '-';}
"/" {++*MPLLColumn; return '/';}
"*" {++*MPLLColumn; return '*';}
"<" {++*MPLLColumn; return '<';}
">" {++*MPLLColumn; return '>';}
"<=" {*MPLLColumn += 2; return SMALLEREQUAL;}
">=" {*MPLLColumn += 2; return LARGEREQUAL;}
"==" {*MPLLColumn += 2; return EQUALS;}
"!=" {*MPLLColumn += 2; return NOTEQUALS;}
"min" {*MPLLColumn += 3; return MIN;}
"max" {*MPLLColumn += 3; return MAX;}
"%"   {++*MPLLColumn; return '%';}
":"   {++*MPLLColumn; return ':';}
"pow" {*MPLLColumn += 3; return POW;}
"sin" {*MPLLColumn += 3; return SIN;}
"cos" {*MPLLColumn += 3; return COS;}
"tan" {*MPLLColumn += 3; return TAN;}
"sind" {*MPLLColumn += 4; return SIND;}
"cosd" {*MPLLColumn += 4; return COSD;}
"asin" {*MPLLColumn += 4; return ASIN;}
":="  {*MPLLColumn += 2; return ASSIGN;}

"->"  {*MPLLColumn += 2; return MAPSTO;}

"if"            {*MPLLColumn += 2; return IF;}
"fi"            {*MPLLColumn += 2; return FI;}
"then"          {*MPLLColumn += 4; return THEN;}
"else"          {*MPLLColumn += 4; return ELSE;}
"case"          {*MPLLColumn += 4; return CASE;}

"or"            {*MPLLColumn += 2; return OR;}
"||"            {*MPLLColumn += 2; return OR;}
"and"           {*MPLLColumn += 3; return AND;}
"&&"            {*MPLLColumn += 2; return AND;}
"xor"           {*MPLLColumn += 3; return XOR;}
"^"             {*MPLLColumn += 1; return XOR;}
"not"           {*MPLLColumn += 3; return NOT;}

"lambda"        {*MPLLColumn += 6; return LAMBDA;}
"Let"           {*MPLLColumn += 3; return LET;}
"let"           {*MPLLColumn += 3; return LET;}
"in"            {*MPLLColumn += 2; return IN;}
"while"         {*MPLLColumn += 5; return WHILE;}

"complement"    {*MPLLColumn += 10; return COMPLEMENT;}
```

```
"union"         {*MPLLColumn += 5;
                 MPLL_lval.value = new string(yytext); return BIO;}
"intersection"  {*MPLLColumn += 12;
                 MPLL_lval.value = new string(yytext); return BIO;}
"setdifference" {*MPLLColumn += 13;
                 return SETDIFFERENCE;}
"cut"           {*MPLLColumn += 3; return CUT;}

"isEmpty"       {*MPLLColumn += 7;
                 MPLL_lval.value = new string(yytext); return PREDICATE;}
"isConvex"      {*MPLLColumn += 8;
                 MPLL_lval.value = new string(yytext); return PREDICATE;}
"isMonotone"    {*MPLLColumn += 10;
                 MPLL_lval.value = new string(yytext); return PREDICATE;}
"isInfinite"    {*MPLLColumn += 10;
                 MPLL_lval.value = new string(yytext); return PREDICATE;}

"size"         {*MPLLColumn += 4; return SIZE;}
"isInfinity"   {*MPLLColumn += 10; return isINFINITY;}

"components"   {*MPLLColumn += 10; return COMPONENTS;}
"component"    {*MPLLColumn += 9; return COMPONENT;}
"point"        {*MPLLColumn += 5; return POINT;}

"times"         {*MPLLColumn += 5;
                 MPLL_lval.value = new string(yytext); return TIMESEXP;}
"exp"           {*MPLLColumn += 3;
                 MPLL_lval.value = new string(yytext); return TIMESEXP;}

"extend"        {*MPLLColumn += 6;
                 MPLL_lval.value = new string(yytext); return EXTINT;}
"integrate"     {*MPLLColumn += 9;
                 MPLL_lval.value = new string(yytext); return EXTINT;}

"core"     {*MPLLColumn += 4;
  switch (*Special_Flag) {
   case 0:  MPLL_lval.value = new string("Icore"); return TOKEN;
   case 1:  MPLL_lval.value = new string("Hcore"); return TOKEN;
   default: MPLL_lval.value = new string("Pcore"); return TOKEN;}}

(I|H|P)core {*MPLLColumn += 5;
             MPLL_lval.value = new string(yytext); return TOKEN;}

"kernel"   {*MPLLColumn += 6;
  switch (*Special_Flag) {
   case 0:  MPLL_lval.value = new string("Ikernel"); return TOKEN;
   case 1:  MPLL_lval.value = new string("Hkernel"); return TOKEN;
   default: MPLL_lval.value = new string("Pkernel"); return TOKEN;}}

(I|H|P)kernel  {*MPLLColumn += 6;
                MPLL_lval.value = new string(yytext); return TOKEN;}

"support"  {*MPLLColumn += 7;
  switch (*Special_Flag) {
   case 0:  MPLL_lval.value = new string("Isupport"); return TOKEN;
   case 1:  MPLL_lval.value = new string("Hsupport"); return TOKEN;
   default: MPLL_lval.value = new string("Psupport"); return TOKEN;}}
```

```
136
   (I|H|P)support {*MPLLColumn += 8;
138                 MPLL_lval.value = new string(yytext); return TOKEN;}

140 "maximum"  {*MPLLColumn += 7;
               MPLL_lval.value = new string(yytext); return TOKEN;}
142


144
   "hull"         {*MPLLColumn += 4;  return HULL;}
146 "scaleup"      {*MPLLColumn += 7;
                   MPLL_lval.value = new string(yytext); return SCINV;}
148 "invert"       {*MPLLColumn += 6;
                   MPLL_lval.value = new string(yytext); return SCINV;}
150
   "fuzzify"      {*MPLLColumn += 7; return FUZZIFY;}
152 "shift"        {*MPLLColumn += 5; return SHIFT;}
   "member"       {*MPLLColumn += 6; return MEMBER;}
154 "integrateAsymmetric" {*MPLLColumn += 19; return INTEGRATEASYMMETRIC;}
   "integrateSymmetric"  {*MPLLColumn += 18; return INTEGRATESYMMETRIC;}
156 "centerPoint" {*MPLLColumn += 11; return CENTERPOINT;}

158 "componentwise" {*MPLLColumn += 13; return COMPONENTWISE;}

160 "newInterval"   {*MPLLColumn += 11; return NEWINTERVAL;}
   "pushBack"      {*MPLLColumn += 8; return PUSHBACK;}
162 "closed"        {*MPLLColumn += 6; return CLOSED;}
   "print"         {*MPLLColumn += 5; return PRINT;}
164 "round"         {*MPLLColumn += 5; return ROUND;}

166 "during"        {*MPLLColumn += 6;  return DURING;}
   "float"         {*MPLLColumn += 5;  return FLOAT;}
168 "NormalizeOverlaps" {*MPLLColumn += 17;  return NORMALIZEOVERLAPS;}
   "isSubset"      {*MPLLColumn += 8;  return isSUBSET;}
170 "doesOverlap"   {*MPLLColumn += 11; return doesOVERLAP;}
   "aILet"         {*MPLLColumn += 5;  return AILET;}
172 "aFLet"         {*MPLLColumn += 5;  return AFLET;}

174
   "newAngle"        {*MPLLColumn += 8; return MPLL_ANGLE;}
176 "grad"               {*MPLLColumn += 4; return MPLL_ANGLE_GRAD;}

178 "newList"        {*MPLLColumn += 7; return MPLL_LIST;}
   "newPolygon"       {*MPLLColumn += 7; return MPLL_POLYGON;}
180 "triangulate"       {*MPLLColumn += 8; return MPLL_POLYGON_TRIANGULATE;}
   "newMultiline"     {*MPLLColumn += 9; return MPLL_MULTILINE;}
182
   "newPoint"        {*MPLLColumn += 8; return MPLL_POINT;}
184 "getX"               {*MPLLColumn += 4; return MPLL_POINT_GETX;}
   "getY"               {*MPLLColumn += 4; return MPLL_POINT_GETY;}
186 "newConfiguration"        {*MPLLColumn += 16; return MPLL_CONFIGURATION;}
   "theAnswerToLifeTheUniverseAndEverything" {*MPLLColumn += 39;
188                                           return ULTIMATE_ANSWER;}
   "element"        {*MPLLColumn += 7; return MPLL_LIST_ELEMENT;}
190 "tail"         {*MPLLColumn += 4; return MPLL_LIST_TAIL;}
   "append"        {*MPLLColumn += 6; return MPLL_LIST_APPEND;}
192
```

```
"cast"                {*MPLLColumn += 4; return MPLL_CAST;}

[\+\-]?[0-9\.]+(e[\+\-])?[0-9]* {
   MPLL_lval.value = new string(yytext);
   *MPLLColumn += MPLL_lval.value->size(); return NUMBER;}

(::)?[a-zA-Z]((::)?[0-9a-zA-Z_\.](::)?)*
   {MPLL_lval.value = new string(yytext);
    *MPLLColumn += MPLL_lval.value->size(); return TOKEN;}

\"[^\"]*\"       {string s(yytext); string s1 = s.substr(1,s.size()-2);
                  unsigned int p = s1.find("\\n");
                  while(p != string::npos)
                     {s1 = s1.replace(p,2,"\n"); p = s1.find("\\n");}
                  MPLL_lval.value = new string(s1);
                  *MPLLColumn += MPLL_lval.value->size();
                  return STRING;}

\"([^\"]*\\\"[^\"]*)+\"       {string s(yytext);
                  string s1 = s.substr(1,s.size()-2);
                  unsigned int p = s1.find("\\n");
                  while(p != string::npos)
                     {s1 = s1.replace(p,2,"\n"); p = s1.find("\\n");}
                  p = s1.find("\\\"");
                  while(p != string::npos)
                     {s1 = s1.replace(p,2,"\""); p = s1.find("\\\"");}
                  MPLL_lval.value = new string(s1);
                  *MPLLColumn += MPLL_lval.value->size();
                  return STRING;}


"\n" {++*MPLLRow; *MPLLColumn = 1;}
"\t" {*MPLLColumn += 8;}
[ \r]+ {*MPLLColumn += string(yytext).size();} /* eat up whitespaces */

%%

void MPLL_Scan(const char* st, int* SpecialFlag) {
   Special_Flag = SpecialFlag;
   MPLLRow = &MPLL::Function::row;
   MPLLColumn = &MPLL::Function::column;
   MPLL::Function::ScannedText = &yytext;
   MPLL_BUFFER = yy_scan_string(st);}

void MPLL_ClearBuffer() {yy_delete_buffer(MPLL_BUFFER);}
```

C.2.2. Parser - Tokens

```
%union {
  string*       value;
  MPLL::Type*   type;
  int           fct;
}
```

```
%token <value>   TOKEN
%token <value>   NUMBER
%token <value>   ULTIMATE_ANSWER

%token <value>   MPLL_CAST

%token <value>   MPLL_LIST
%token <value>   MPLL_LIST_ELEMENT
%token <value>   MPLL_LIST_SIZE
%token <value>   MPLL_LIST_TAIL
%token <value>   MPLL_LIST_APPEND

%token <value>   MPLL_POLYGON
%token <value>   MPLL_POLYGON_TRIANGULATE

%token <value>   MPLL_MULTILINE

%token <value>   MPLL_ANGLE
%token <value>   MPLL_ANGLE_PREDICATE
%token <value>   MPLL_ANGLE_MIN
%token <value>   MPLL_ANGLE_MAX
%token <value>   MPLL_ANGLE_GRAD

%token <value>   MPLL_POINT
%token <value>   MPLL_POINT_GETX
%token <value>   MPLL_POINT_GETY

%token <value>   MPLL_CONFIGURATION

%token <value>   STRING

%token <value>   EQUALS
%token <value>   NOTEQUALS
%token <value>   SMALLEREQUAL
%token <value>   LARGEREQUAL
%token <value>   MIN
%token <value>   MAX
%token <value>   POW
%token <value>   SIN
%token <value>   COS
%token <value>   TAN
%token <value>   SIND
%token <value>   COSD
%token <value>   ASIN
%token <value>   FLOAT

%token <value>   MAPSTO

%token <value>   IF
%token <value>   FI
%token <value>   THEN
%token <value>   ELSE

%token <value>   SWITCH
%token <value>   CASE
%token <value>   ASSIGN
```

```
%token <value>   AND
%token <value>   OR
%token <value>   XOR
%token <value>   NOT

%token <value>   LAMBDA
%token <value>   LET
%token <value>   dLET
%token <value>   IN
%token <value>   WHILE
%token <value>   COMPLEMENT

%token <value>   BIO
%token <value>   UNION
%token <value>   INTERSECTION

%token <value>   SETDIFFERENCE
%token <value>   CUT

%token <value>   PREDICATE
%token <value>   isEMPTY
%token <value>   isCONVEX
%token <value>   isMONOTONE
%token <value>   isINFINITE
%token <value>   isSUBSET
%token <value>   doesOVERLAP

%token <value>   SIZE

%token <value>   isINFINITY
%token <value>   COMPONENTS
%token <value>   COMPONENT
%token <value>   POINT
%token <value>   EXTINT
%token <value>   EXTEND
%token <value>   INTEGRATE
%token <value>   TIMESEXP
%token <value>   TIMES
%token <value>   TIME
%token <value>   EXP
%token <value>   HULL
%token <value>   SCINV
%token <value>   SCALEUP
%token <value>   INVERT
%token <value>   FUZZIFY
%token <value>   SHIFT
%token <value>   MEMBER
%token <value>   INTEGRATESYMMETRIC
%token <value>   INTEGRATEASYMMETRIC
%token <value>   CENTERPOINT
%token <value>   EMPTYINTERVAL
%token <value>   NORMALIZEOVERLAPS
%token <value>   COMPONENTWISE
%token <value>   NEWINTERVAL
%token <value>   PUSHBACK
%token <value>   CLOSED
%token <value>   PRINT
```

```
%token <value>   ROUND
%token <value>   DURING
%token <value>   DEGREE
%token <value>   AILET
%token <value>   AFLET

%left MAPSTO
%left '-'
%left '+'
%left '/'
%left '*'
%left '%'

%left XOR
%left OR
%left AND

%type <value> definition
%type <value> parameters
%type <type>  expression
%type <type>  arithExpression
%type <type>  boolExpression
%type <type>  typeExpression
%type <type>  genericExpression
%type <type>  typeExpressions
%type <fct>   binaryCmpOperator
%type <type>  caseExpressions
%type <type>  SetDiffRest
%type <type>  printExpression
%type <type>  printExpressions
%type <type>  listExpression
%type <type>  listExpressions
%type <fct>   vars
```

C.2.3. Parser - Type Expressions

```
parameters:
  typeExpression TOKEN  {
  if($<type>1 != NULL) MPLL_function->addParameter($<type>1,$<value>2);}
| parameters ',' typeExpression TOKEN {
  if($<type>3 != NULL) MPLL_function->addParameter($<type>3,$<value>4);}
;

genericExpression:
 '<' typeExpression  '>' {$<type>$ = $<type>2;}
|  {$<type>$ = NULL;}
;

typeExpression:
TOKEN genericExpression {
  $<type>$ = MPLL::BasicType::getType(*$<value>1);
  if ($<type>2 != NULL) {
        if ($<type>$->hasGenericType()) {
        MPLL::Type* testType = MPLL::BasicType::copyBasicType(
```

```
            static_cast<MPLL::BasicType*>($<type>$));
        testType->setGenericType($<type>2);
        cout << testType->toString()<< "\n";
        $<type>$ = testType;
        } else {
                yyerror($<type>$->toString() + " is not a generic type. ");
        }
  }
  if($<type>$ == NULL) {
        $<type>$ = new MPLL::PolyType(*$<value>1);
  }
}


| typeExpression MAPSTO typeExpression {
  if($<type>1 && $<type>3) $<type>$ = new MPLL::CompoundType($<type>1,
        $<type>3);
  else $<type>$ = NULL;}

| '(' typeExpressions ')' MAPSTO typeExpression {
  vector<MPLL::Type*> types = MPLL_TypeVector.back();
  MPLL_TypeVector.pop_back();
  if($<type>5 == NULL) $<type>$ = NULL;
  else {
    $<type>$ = $<type>5;
    int tEnd = types.size();
    for(int i = 0; i < tEnd; ++i) if(types[i] == NULL)
        {$<type>$ = NULL; break;}
    if($<type>$) $<type>$ = new MPLL::CompoundType(types, $<type>5);}}

 | '(' typeExpression ')'{$<type>$ = $<type>2;}

 | error MAPSTO typeExpression {
   $<type>$ = NULL;}

 | '(' error ')'  MAPSTO typeExpression {
   $<type>$ = NULL;}

 | typeExpression MAPSTO error {
   $<type>$ = NULL;}

| '(' typeExpressions ')' MAPSTO error {
   $<type>$ = NULL;}
;

typeExpressions:
typeExpression {
  MPLL_TypeVector.push_back(vector<MPLL::Type*>());
  MPLL_TypeVector.back().push_back($<type>1);}
| typeExpressions '*' typeExpression {
  MPLL_TypeVector.back().push_back($<type>3);};


expression:
TOKEN {
  $<type>$ = NULL;
  MPLL::Token* Token = new MPLL::Token($<value>1,MPLL_function);
```

```
76   if(MPLL_function->noError()) {
       MPLL_function->push_back(Token);
78     $<type>$ = Token->resultType;
     } else {
80     delete Token;
     }
82 }

84 | MPLL_CAST '(' typeExpression ',' expression ')' {
           MPLL::Cast* Op = new MPLL::Cast($<type>3, $<type>5, MPLL_function);
86           if(MPLL_function->noError()) {
                 MPLL_function->push_back(Op);
88               $<type>$ = Op->resultType;
           } else delete Op;
90 }

92 | ULTIMATE_ANSWER {
           $<type>$ = NULL;
94         MPLL::Token* Token = new MPLL::Token(new string("42"),
                                                MPLL_function);
96         if(MPLL_function->noError()) {
                 MPLL_function->push_back(Token);
98               $<type>$ = Token->resultType;}
           else delete Token;
100 }

102 | MPLL_MULTILINE '(' expression ')' {
           MPLL::Multiline* Op = new MPLL::Multiline($<type>3, MPLL_function);
104          if(MPLL_function->noError()) {
                 MPLL_function->push_back(Op);
106              $<type>$ = Op->resultType;
           } else delete Op;
108 }

110 | MPLL_POLYGON '(' expression ')' {
           MPLL::Polygon* Op = new MPLL::Polygon($<type>3, MPLL_function);
112          if(MPLL_function->noError()) {
                 MPLL_function->push_back(Op);
114              $<type>$ = Op->resultType;
           } else delete Op;
116 }

118 | MPLL_LIST '(' {MPLL_ListVector.push_back(vector<pair<MPLL::MPLLValue*,
           MPLL::Type*> >());} listExpressions ')' {
120   MPLL::List* Op = new MPLL::List(MPLL_ListVector.back(),MPLL_function);
      MPLL_ListVector.pop_back();
122   MPLL_function->push_back(Op);
      $<type>$ = Op->resultType;}
124
    | MPLL_POLYGON_TRIANGULATE '(' expression ')' {
126         MPLL::PolygonPredicate* Op = new MPLL::PolygonPredicate(1,
              MPLL_POLYGON_TRIANGULATE, $<type>3, MPLL_function);
128           if(MPLL_function->noError()) {
                 MPLL_function->push_back(Op);
130              $<type>$ = Op->resultType;
           } else delete Op;
132 }
```

```
134 | MPLL_LIST_ELEMENT '(' expression ',' expression ')' {
        MPLL::ListPredicate* Op = new MPLL::ListPredicate(2,
136        MPLL_LIST_ELEMENT, $<type>3, $<type>5, MPLL_function);
          if(MPLL_function->noError()) {
138             MPLL_function->push_back(Op);
                $<type>$ = Op->resultType;
140     } else delete Op;
  }
142
  | MPLL_LIST_APPEND '(' expression ',' expression ')' {
144     MPLL::ListPredicate* Op = new MPLL::ListPredicate(2,
          MPLL_LIST_APPEND, $<type>3, $<type>5, MPLL_function);
146       if(MPLL_function->noError()) {
             MPLL_function->push_back(Op);
148          $<type>$ = Op->resultType;
        } else delete Op;
150 }

152 | MPLL_LIST_TAIL '(' expression ')' {
        MPLL::ListPredicate* Op = new MPLL::ListPredicate(1,
154        MPLL_LIST_TAIL, $<type>3, NULL, MPLL_function);
          if(MPLL_function->noError()) {
156             MPLL_function->push_back(Op);
                $<type>$ = Op->resultType;
158     } else delete Op;
  }
160
  | MPLL_LIST '(' error ')' {$<type>$ = NULL;  yyerror("list statement");}
162
  | NUMBER {
164   $<type>$ = NULL;
    MPLL::Token* Token = new MPLL::Token($<value>1,MPLL_function);
166 if(MPLL_function->noError()) {
      MPLL_function->push_back(Token);
168   $<type>$ = Token->resultType;}
    else delete Token;}
170
  | MPLL_ANGLE '(' expression ',' expression ',' expression ')' {
172     $<type>$ = NULL;
        MPLL::Angle* Op = new MPLL::Angle($<type>3, $<type>5,
174                                         $<type>7, MPLL_function);
          if(MPLL_function->noError()) {
176             MPLL_function->push_back(Op);
                $<type>$ = Op->resultType;
178     }
        else delete Op;
180 }

182 | MPLL_ANGLE_GRAD '(' expression ')' {
        MPLL::AnglePredicate* Op = new MPLL::AnglePredicate(MPLL_ANGLE_GRAD,
184                                         $<type>3, MPLL_function);
          if(MPLL_function->noError()) {
186             MPLL_function->push_back(Op);
                $<type>$ = Op->resultType;
188     } else delete Op;
  }
```

```
190
  | MPLL_POINT '(' expression ',' expression ')' {
192         $<type>$ = NULL;
            MPLL::Point* Op = new MPLL::Point($<type>3, $<type>5, MPLL_function);
194           if(MPLL_function->noError()) {
                  MPLL_function->push_back(Op);
196               $<type>$ = Op->resultType;
            }
198         else delete Op;
  }
200
  | MPLL_POINT_GETX '(' expression ')' {
202         MPLL::PointPredicate* Op = new MPLL::PointPredicate(MPLL_POINT_GETX,
               $<type>3, MPLL_function);
204           if(MPLL_function->noError()) {
                  MPLL_function->push_back(Op);
206               $<type>$ = Op->resultType;
            } else delete Op;
208 }

210 | MPLL_POINT_GETY '(' expression ')' {
            MPLL::PointPredicate* Op = new MPLL::PointPredicate(MPLL_POINT_GETY,
212            $<type>3, MPLL_function);
              if(MPLL_function->noError()) {
214               MPLL_function->push_back(Op);
                  $<type>$ = Op->resultType;
216         } else delete Op;
  }
218
  | MPLL_CONFIGURATION '(' expression ',' expression ',' expression ','
220                        expression ')' {
            $<type>$ = NULL;
222         MPLL::Configuration* Op = new MPLL::Configuration($<type>3,
               $<type>5, $<type>7, $<type>9, MPLL_function);
224           if(MPLL_function->noError()) {
                  MPLL_function->push_back(Op);
226               $<type>$ = Op->resultType;
            }
228         else delete Op;
  }
230
  | PRINT '(' {MPLL_PrintVector.push_back(vector<pair<string*,MPLL::Type*>
232              >());} printExpressions ')' {
    MPLL::Print* Op = new MPLL::Print(MPLL_PrintVector.back(),MPLL_function);
234 MPLL_PrintVector.pop_back();
    MPLL_function->push_back(Op);
236 $<type>$ = Op->resultType;}

238 | PRINT '(' error ')' {$<type>$ = NULL;  yyerror("print statement");}

240 | EMPTYINTERVAL {
    MPLL::emptyInterval* Op = new MPLL::emptyInterval();
242 MPLL_function->push_back(Op);
    $<type>$ = Op->resultType;}
```

C.2.4. C++ Sources

Angle.h

```
#include <iostream>
#include <sstream>

#ifndef ANGLE
#define ANGLE

using namespace std;

namespace DataType {
          class Angle;

class Angle {
        private:
        long int angle;
        long int min;
        long int max;
        static const long int ANGLE_MULTIPLIER = 1000000;
        void init(const long int, const long int, const long int);
        static double convert(long int);

        public:
        Angle (const Angle& );
        Angle (long int a = 0, long int b = 0, long int c = 0);
        string toString() const;
        long int getAngle() const;
        void setAngle(const long int);
        long int getMin() const;
        void setMin(const long int);
        long int getMax() const;
        void setMax(const long int);
        Angle* getNegative() const;
        Angle* addAngle(const Angle a);
        Angle* subtractAngle(const Angle a);
        Angle* multiply(const float f);
        bool equals(const Angle a) const;
        friend ostream& operator<<(ostream& s, const  Angle& r);
        friend ostringstream& operator<<(ostringstream& s,  Angle& r);
        friend istream& operator>>(istream& s, Angle& r);
        Angle& operator=(const Angle&);
};
}
#endif
```

Angle.cpp

```
#include"Angle.h"
#include <math.h>
```

```
4 #define PI 3.14159265

6 using namespace DataType;

8 namespace DataType {
        class Angle;
10
        const long int Angle::ANGLE_MULTIPLIER;
12
        void Angle::init(const long int angle2, const long int min2,
14                       const long int max2) {
                setMin(min2);
16              setMax(max2);
                setAngle(angle2);
18      }

20      double Angle::convert(long int i)  {
                return ((double)i)/(double)(Angle::ANGLE_MULTIPLIER);
22      }

24      Angle::Angle(long int value, long int min, long int max) {
                init (value, min, max);
26      }

28      Angle::Angle(const Angle& a) {
                init (a.getAngle(), a.getMin(), a.getMax());
30      }

32      long int Angle::getAngle() const {
                return angle;
34      }

36      void Angle::setAngle(const long int a) {
                double helperA = (double)a;
38              double helperMin = (double)min;
                double helperMax = (double)max;
40              helperA = fmod((helperA - helperMin),
                               (helperMax-helperMin)) + helperMin;
42              helperA = -(fmod((-helperA -(-helperMax)),
                                 (helperMax-helperMin)) + (-helperMax));
44              angle = (long int) helperA;
        }
46
        long int Angle::getMin() const {
48              return min;
        }
50
        void Angle::setMin(const long int min) {
52              this->min = min;
        }
54
        long int Angle::getMax() const {
56              return max;
        }
58
        void Angle::setMax(const long int max) {
```

```
60              (*this).max = max;
        }
62
        ostream& operator<<(ostream& s, const Angle& a) {
64              s << "Angle(" << Angle::convert(a.getAngle())
                  <<  ',' << Angle::convert(a.getMin())
66                <<  ',' << Angle::convert(a.getMax()) << ')';
                return s;
68      }

70      ostringstream& operator<<(ostringstream& s, Angle& a) {
                s << "Angle(" << Angle::convert(a.getAngle())
72                <<  ',' << Angle::convert(a.getMin()) <<  ','
                  << Angle::convert(a.getMax()) << ')';
74              return s;
        }
76
        istream& operator>>(istream& i, Angle& a) {
78              float value = 0;
                char c = 0;
80
                i >> value >> c;
82
                if (c == 'R') {
84                      int v = (int)
                          (value*Angle::ANGLE_MULTIPLIER*200/PI);
86                      int max = (Angle::ANGLE_MULTIPLIER*400);
                        int min = -max;
88                      a.init(v,min,max);
                } else if (c== 'D') {
90                      int v = (int)
                          (value*Angle::ANGLE_MULTIPLIER*10/9);
92                      int max = (Angle::ANGLE_MULTIPLIER*400);
                        int min = -max;
94                      a.init(v,min,max);
                } else if (c=='G') {
96                      int v = (int)
                          (value*Angle::ANGLE_MULTIPLIER);
98                      int max = (Angle::ANGLE_MULTIPLIER*400);
                        int min = -max;
100                     a.init(v,min,max);
                } else {
102                     i.clear(ios_base::badbit);
                }
104
                return i;
106     }

108     Angle* Angle::addAngle(const Angle a) {
                Angle* ang =  new Angle(*this);
110             ang->setAngle(this->getAngle() + a.getAngle());
                return ang;
112     }

114     Angle* Angle::subtractAngle(const Angle a) {
                Angle* temp = a.getNegative();
116             return addAngle(*temp);
```

```
        }

        Angle* Angle::multiply(const float scalar) {
                Angle* a = new Angle(*this);
                a->setAngle((long int)
                            ((double)getAngle()*((double)scalar)));
                return a;
        }

        Angle* Angle::getNegative() const {
                Angle* result = new Angle(*this);
                result->setAngle(-getAngle());
                return result;
        }

        Angle& Angle::operator= (const Angle& a) {
                if (this != &a) {
                        init(a.getAngle(), a.getMin(), a.getMax());
                }
                return *this;
        }

        bool Angle::equals(const Angle a) const {
                return (getAngle() == a.getAngle() &&
                        getMin()   == a.getMin() &&
                        getMax()   == a.getMax());
        }
}
```

Configuration.h

```
#include <iostream>
#include <sstream>
#include <vector>

#include "Angle.h"
#include "Point.h"

#ifndef CONFIGURATION
#define CONFIGURATION

using namespace std;
using namespace DataType;

namespace DataType {
        class Angle;
        class Point;

        class Configuration : public Point {
                private:
                bool isOrientated;
```

```
22                  Angle orientation;

24                  public:
                    Configuration (Angle&, Angle&, Angle&, bool);
26
                    bool hasOrientation() const;
28                  double convert(int) const;

30                  Angle getOrientation() const;

32                  friend ostream& operator<<(ostream& s,
                                               const Configuration& p);
34                  friend ostringstream& operator<<(ostringstream& s,
                                                     Configuration& p);
36                  friend istream& operator>>(istream& s,
                                               Configuration& p);
38          };
    }
40  #endif
```

Configuration.cpp

```
    #include "Configuration.h"
2   #include "Angle.h"
    #include "Point.h"
4
    using namespace DataType;
6
    namespace DataType {
8           class Configuration;
            class Angle;
10          class Point;

12          Configuration::Configuration(Angle& angle1, Angle& angle2,
                                         Angle& angle3, bool isOrientated
14                                       ) : Point(angle2, angle3) {
                    orientation = angle1;
16                  Configuration::isOrientated = isOrientated;
            }
18
            bool Configuration::hasOrientation() const {
20                  return isOrientated;
            }
22
            Angle Configuration::getOrientation() const {
24                  return orientation;
            }
26
            ostream& operator<< (ostream& o, const Configuration& c) {
28                  o << "Configuration(";
                    if (c.hasOrientation()) {
```

```
30                          o  <<  c.getOrientation() << ", ";
                    }
32                  o << ((Point)c) << ')';
                    return o;
34          }

36          ostringstream& operator<< (ostringstream& o, Configuration& c) {
                    o << "Configuration(";
38                  o << "Configuration(";
                    if (c.hasOrientation()) {
40                          o  <<  c.getOrientation() << ", ";
                    }
42                  o << ((Point)c) << ')';
                    return o;
44          }

46          istream& operator>> (istream& i, Configuration& config) {
                    return i;
48          }
    }
```

List of Acronyms

AF Alternative Frequencies (term from RDS/TMC)

API Application Programming Interface

BFS Breadth-First Search

CT Clock Time (term from RDS/TMC)

DFS Depth-First Search

DL Description Logics

EBU European Broadcast Union

ECL Event Code List (term from RDS/TMC)

EON Enhanced Other Networks (term from RDS/TMC)

FM Frequency Modulation

GDF Geographic Data Format

GIS Geographic Information System

GML Geography Mark-Up Language

GPS Global Positioning System

GRS Geodetic Reference System

GSM Global System for Mobile Communications

GXL Graph Exchange Language

GeTS Specification Language for Geo-Temporal Notions

IEEE Institute of Electrical and Electronics Engineers

IERS International Earth Rotation and Reference Systems Service

ISO International Organization for Standardization

ITRS International Terrestrial Reference System

ITS Intelligent Transport Systems

JDRMA Japan Digital Roadmap Association

JUNG Java Universal Network/Graph Framework

KML Keyhole Mark-Up Language

L-DSMS Local Data Stream Management System

LCL Location Code List (term from RDS/TMC)

LPB Landmark Pair Boundary

MDA Model Driven Architecture

MPLL Multi-Paradigm Location Language

NER Named Entity Recognition

ODA Open Data Applications

OTN Ontology of Transportation Networks

OWL Web Ontology Language

PDA Personal Digital Assistent

PIN Programme Item Number (term from RDS/TMC)

PI Programme Identification (term from RDS/TMC)

PS Programme Service Name (term from RDS/TMC)

PTY Programme Type (term from RDS/TMC)

RBDS Radio Broadcast Data System

RCC Region Connection Calculus

RDF Resource Description Framework

RDS Radio Data System

RS Reference System

RT Radio Text (term from RDS/TMC)

SDTS Spatial Data Transfer Standard

SVG Scalable Vector Graphics

TIS Traffic Information System

TMC Traffic Message Channel

TP/TA Traffic Programme and Traffic Announcement (term from RDS/TMC)

TSP Travelling salesman problem

TTI Traffic and Travel Information

UTM Universal Transverse Mercator coordinate system

VHF Very High Frequency

VLBI Very Long Baseline Interferometry

VLSI Very Large Scale Integration

WGS World Geodetic System

WLAN Wireless Local Area Network

XLink XML Linking Language

XML Extensible Mark-Up Language

rton Relative Topological Orientation Node

Bibliography

[1] Charlie Abela and Matthew Montebello. PreDiCtS: A Personalised Service Discovery and Composition Framework. In *Proceedings of Semantic Web Personalization Workshop, Budva, Montenegro (12th June 2006)*, pages 1–10, 2006.

[2] Pragya Agarwal, Yongjian Huang, and Vania Dimitrova. Formal Approach to Reconciliation of Individual Ontologies for Personalisation of Geospatial Semantic Web. In M. Andrea Rodríguez, Isabel F. Cruz, Max J. Egenhofer, and Sergei Levashkin, editors, *GeoS*, volume 3799 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2005.

[3] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM (CACM)*, 26(11):832–843, 1983.

[4] *www.answers.com: Landmark Definition*. `http://www.answers.com`, accessed August 2006.

[5] Lora Aroyo, Ronald Denaux, Vania Dimitrova, and Michael Pye. Interactive Ontology-Based User Knowledge Acquisition: A Case Study. In York Sure and John Domingue, editors, *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 560–574. Springer, 2006.

[6] I. Budak Arpinar, Amit Sheth, Cartic Ramakrishnan, E. Lynn Usery, Molly Azami, and Mei-Po Kwan. Geospatial Ontology Development and Semantic Analytics. *Transactions in GIS*, 10(4):551–575, July 2006.

[7] Nicholas Asher and Laure Vieu. Toward a geometry for common sense: A semantics and a complete axiomatization for mereotopology. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 846–852, San Francisco, 1995. Morgan Kaufmann.

[8] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[9] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Laura Torasso. Verifying the compliance of personalized curricula to curricula models in the semantic web. In *Proceedings of Semantic Web Personalization Workshop, Budva, Montenegro (12th June 2006)*, pages 53–62, 2006.

[10] Chris Barrett, Riko Jacob, and Madhav V. Marathe. Formal language constrained path problems. In *Scandinavian Workshop on Algorithm Theory*, pages 234–245, 1998.

[11] Oliver Bender, Franz Josef Och, and Hermann Ney. Maximum entropy models for named entity recognition. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 148–151. Edmonton, Canada, 2003.

[12] M. Betke. Learning and Vision Algorithms for Robot Navigation. Technical Report MIT/LCS/TR-671, Massachusetts Institute of Technology, 1995.

[13] P. Bloom, M. A. Peterson, L. Nadel, and M. F. Garrett, editors. *Language and Space*. MIT Press, Cambridge, MA, 1996.

[14] Jean Daniel Boissonnat and Mariette Yvinec. *Algorithmique Geometry*. Cambridge University Press, 1998. BOI j 98:1 1.Ex.

[15] Johann Borenstein, H. R. Everett, and Liqiang Feng. *Navigating Mobile Robots: Systems and Techniques*. A. K. Peters, Ltd., Natick, MA, USA, 1996.

[16] Amy Briggs, Daniel Scharstein, Darius Braziunas, Cristian Dima, and Peter Wall. Mobile Robot Navigation Using Self-Similar Landmarks. In *IEEE International Conference on Robotics and Automation*, pages 1428–1434, San Francisco, April 2000.

[17] Levin Brunner, Klaus U. Schulz, and Felix Weigel. Organizing Thematic, Geographic and Temporal Knowledge in a Well-founded Navigation Space: Logical and Algorithmic Foundations for EFGT Nets. *Journal of Web Services Research, Special Issue "Bridging Communities: Semantically Augmented Metadata for Services, Grids, and Software Engineering"*, 2006.

[18] David J. Bryant. Human spatial concepts reflect regularities of the physical world and human body. In Patrick Olivier and Klaus-Peter Gapp, editors, *Representation and processing of spatial expressions*, pages 215–230. Lawrence Erlbaum Associates, Mahwah, New Jersey, 1998.

[19] Giorgio Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*. PhD thesis, University of Paderborn, 2002.

[20] Michael Buschmann and Markus Krieser. Protokollierung und statistische Auswertung von RDS/TMC Datenströmen. Projektarbeit/project thesis, Institute for Informatics, University of Munich, 2006.

[21] Adrijana Car, Henny Mehner, and George Taylor. Experimenting with hierarchical wayfinding, 1999.

[22] S. Chang and E. Jungert. Pictorial data management based upon the theory of symbolic projections. *Journal of Visual Languages and Computations*, 2(3):195–215, 1991.

[23] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C – World Wide Web Consortium, 1999. `http://www.w3.org/TR/xpath`.

[24] Bowman L. Clarke. A calculus of individuals based on 'connection'. *Notre Dame Journal of Formal Logic*, 22(3):204–218, 1981.

[25] Bowman L. Clarke. Individuals and points. *Notre Dame Journal of Formal Logic*, 26(1):61–67, 1985.

[26] Eliseo Clementini, Paolino Di Felice, and Daniel Hernández. Qualitative representation of positional information. *Artificial Intelligence*, 95(2):317–356, 1997.

[27] Anthony G. Cohn. The challenge of qualitative spatial reasoning. *ACM Computing Surveys*, 27(3):323–325, 1995.

[28] Anthony G. Cohn. Calculi for qualitative spatial reasoning. In Jacques Calmet, J. Campbell, and J. Pfalzgraf, editors, *Artificial Intelligence and Symbolic Mathematical Computation*, pages 124–143. Springer-Verlag, Berlin, 1996.

[29] Anthony G. Cohn, Brandon Bennett, John Gooday, and Nicholas M. Gotts. Representing and reasoning with qualitative spatial relations. In Oliviero Stock, editor, *Spatial and Temporal Reasoning*, pages 97–134. Kluwer Academic Publishers, Dordrecht, 1997.

[30] Anthony G. Cohn, Brandon Bennett, John Gooday, and Nicholas Mark Gotts. Qualitative spatial representation and reasoning with the region connection calculus. *GeoInformatica*, 1(3):275–316, 1997.

[31] Anthony G. Cohn and S. M. Hazarika. Qualitative spatial representation and reasoning: An overview. *Fundamenta Informaticae*, 46(1-2):1–29, 2001.

[32] Anthony G. Cohn and Achille C. Varzi. Connection relations in mereotopology. In *ECAI*, pages 150–154, 1998.

[33] Anthony G. Cohn and Achille C. Varzi. Modes of connection. In Freksa and Mark [61], pages 299–314.

[34] Anthony G. Cohn and Achille C. Varzi. Mereotopological connection. *Journal of Philosophical Logic*, 32:357–390, 2003.

[35] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, ISBN 0-262-03293-7, 2001. 525 pp.

[36] Matteo Cristani. The complexity of reasoning about spatial congruence. *Journal of Artificial Intelligence Research*, 11:361–390, 1999.

[37] S. Cucerzan and D. Yarowsky. Language independent named entity recognition combining morphological and contextual evidence, 1999.

[38] James R. Curran and Stephen Clark. Language independent ner using a maximum entropy tagger. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 164–167. Edmonton, Canada, 2003.

[39] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000. BER m2 00:1 1.Ex.

[40] Rina Dechter. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2003.

[41] J. Peter Denny. Locating the universals in lexical systems for spatial deixis. *In: Papers from the Parasession on the Lexicon [53]*, 1978.

[42] Nachum Dershowitz and Edward Reingold. *Calendrical calculations: The millennium edition*. Cambridge University Press, 2001.

[43] A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, 2001.

[44] Edsger W. Dijkstra. The Three Golden Rules for Successful Scientific Research. In *Selected Writings on Computing: A Personal Perspective*, pages 329–330. Springer-Verlag, 1982.

[45] *Microsoft Developer Network (MSDN): DirectX SDK Documentation*. `http://msdn.microsoft.com/library`, June 2006.

[46] Paul Dourish. What We Talk About When We Talk About Context. *Personal and Ubiquitous Computing*, 8(1):19–30, February 2004.

[47] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.

[48] Max J. Egenhofer. Reasoning about Binary Topological Relations. In O. Günther and H.-J. Schek, editors, *Proceedings of the Second International Symposium on Advances in Spatial Databases (SSD '91)*, pages 143–160, Heidelberg, 1991. Springer-Verlag.

[49] Max J. Egenhofer and Jayant Sharma. Topological Relations Between Regions in $\mathbb{R}^2$ and $\mathbb{Z}^2$. In *Proceedings of the Third International Symposium on Advances in Spatial Databases (SSD '93)*, pages 316–336, London, UK, 1993. Springer-Verlag.

[50] *EPA - U.S. Environmental Protection Agency – Glossary of mapping terms*. `http://www.epa.gov/ceisweb1/ceishome/atlas/learngeog/glossaryofmappingterms.html`, accessed June 2005.

[51] M. Teresa Escrig and Francisco Toledo. *Qualitative Spatial Reasoning: Theory and Practice - Application to Robot Navigation*, volume 47 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, ISBN: 90-5199-412-5, 1998.

[52] M. Theresa Escrig, Francisco Toledo, and Angel P. del Pobil. An overview to qualitative spatial reasoning. In *Current Trends in Qualitative Reasoning and Applications*, pages 43–60. International Center for Numerical Methods in Engineering, Barcelona, 1995.

[53] Donka Farkas, Wesley M. Jacobsen, and Karol W. Todrys, editors. *Papers from the Parasession on the Lexicon*. Chicago Linguistics Society, Chicago, 1978.

[54] Radu Florian, Abe Ittycheriah, Hongyan Jing, and Tong Zhang. Named entity recognition through classifier combination. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 168–171. Edmonton, Canada, 2003.

[55] The Free Software Foundation. *bison: A General-purpose Parser Generator*. `http://www.gnu.org/software/bison`, accessed August 2006.

[56] Lidia Fraczak. Generating "mental maps" from route descriptions. In Patrick Olivier and Klaus-Peter Gapp, editors, *Representation and processing of spatial expressions*, pages 185–200. Lawrence Erlbaum Associates, Mahwah, New Jersey, 1998.

[57] Andrew Frank. Qualitative Spatial Reasoning about Distance and Directions in Geographic Space. *Journal of Visual Languages and Computations*, 3(2):343–373, 1992.

[58] C. Freksa and K. Zimmermann. On the utilization of spatial structures for cognitively plausible and efficient reasoning. In FD. Anger HW. Güsgen and J. v.Benthem, editors, *Proc. of the Workshop on Spatial and temporal reasoning*, pages 61–66. Chambery, 1993.

[59] Christian Freksa. Temporal Reasoning Based on Semi-Intervals. *Applied Intelligence*, 54:199–227, 1992.

[60] Christian Freksa. Using orientation information for qualitative spatial reasoning, 1992.

[61] Christian Freksa and David M. Mark, editors. *Spatial Information Theory: Cognitive and Computational Foundations of Geographic Information Science, International Conference COSIT '99, Stade, Germany, August 25-29, 1999, Proceedings*, volume 1661 of *Lecture Notes in Computer Science*. Springer, 1999.

[62] Norbert E. Fuchs and Uta Schwertel. Reasoning in Attempto Controlled English. In *Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2003)*, Lecture Notes in Computer Science, Mumbai, India, 2003. Springer.

[63] *Geographic Data Files 3.0 (GDF) Documentation*. `http://www.ertico.com`, 1995.

[64] Alfonso Gerevini and Bernhard Nebel. Qualitative spatio-temporal resoning with rcc-8 and allen's interval calculus: Computational complexity. In *In Proccedings of the 15th European Conference on Artificial Intelligence (ECAI'02)*, June 2002.

[65] *GML - A Markup Language for Geography*. `http://opengis.net/gml`, accessed August 2006.

[66] *Google Earth*. `http://earth.google.com`, accessed August 2006.

[67] Nicholas Mark Gotts. Using the RCC Formalism to Describe the Topology of Spherical Regions. Technical report, University of Leeds, 1996.

[68] Kamal Gupta and Angel P. del Pobil, editors. *Practical Motion Planning in Robotics*. Wiley, 1998. ISBN: 0-471-98163-X.

[69] *Graph eXchange Language*. `http://www.gupro.de/GXL`, accessed August 2006.

[70] *Graph eXchange Language Project*. `http://gxl.sourceforge.net/index.html`, accessed August 2006.

[71] Christian Hänsel. TMC-KML Verkehrsdatenservice für den Google Earth Client. Projektarbeit/project thesis, Institute for Informatics, University of Munich, 2006.

[72] Andreas Heindel. Nutzung von Indoor-Positionierungsdaten zur mobilen Wegeplanung. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, March 2006.

[73] Nicola Henze. Personalisierbare Informationssysteme im Semantic Web. In Tassilo Pellegrini and Andreas Blumauer, editors, *Semantic Web. Wege zur vernetzten Wissensgesellschaft*. Springer, Berlin, 2006.

[74] Nicola Henze. Personalized e-Learning in the Semantic Web. *International Journal of Emerging Technologies in Learning (iJET)*, 1(1), 2006.

[75] Nicola Henze. Personalized e-Learning in the Semantic Web. In *Proceedings of First International Conference on Interactive Mobile and Computer Aided Learning, Amman, Jordan (19th–21st April 2006)*, 2006.

[76] Daniel Hernández. *Qualitative Representation of Spatial Knowledge*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994.

[77] Daniel Hernández, Eliseo Clementini, and Paolino Di Felice. Qualitative distances. In Andrew U. Frank and Werner Kuhn, editors, *Spatial Information Theory: A Theoretical Basis for GIS, International Conference COSIT '95, Semmering, Austria, September 21-23, 1995, Proceedings*, pages 45–57, 1995.

[78] Frank Ipfelkofer. Basisontologie und Anwendungs-Framework für Visualisierung und Geospatial Reasoning. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, 2004.

[79] *International Organization for Standardization (ISO)*. `http://www.iso.org`, accessed August 2006.

[80] ISO Standard 14819-1: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 1: Coding protocol for Radio Data System – Traffic Message Channel (RDS-TMC) using ALERT-C. `http://www.iso.org`, May 2003.

[81] ISO Standard 14819-2: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 2: Event and information codes for Radio Data System – Traffic Message Channel (RDS-TMC). `http://www.iso.org`, May 2003.

[82] *Intelligent Transport Systems (ITS)*. `http://www.iso.org`, accessed August 2006.

[83] ISO/TS Standard 14819-3: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 3: Location referencing for ALERT-C. `http://www.iso.org`, June 2000.

[84] ISO/TS Standard 14825: Intelligent transport systems – Geographic Data Files (GDF) – Overall data specification. `http://www.iso.org`, February 2004.

[85] *Japan Digital Roadmap Association (JDRMA)*. `http://www.drm.jp/drm/e_index.htm`, accessed August 2006.

[86] Ian Johnson. Mapping the Fourth Dimension: The TimeMap Project. In L. Dingwall, S. Exon, V. Gaffney, S. Laflin, and M. Van Leusen, editors, *Proceedings of the 25th Computer Applications in Archaeology Conference (CAA), Birmingham, April 1997*, BAR International Series 750, page 21 pp., Oxford, UK, 1999. Archaeopress.

[87] Simon J. Julier and Jeffrey K. Uhlmann. Consistent debiased method for converting between polar and cartesian coordinate systems. *Acquisition, Tracking, and Pointing XI*, 3086(1):110–121, 1997.

[88] *Java Universal Network/Graph Framework*. `http://jung.sourceforge.net`, accessed August 2006.

[89] Erland Jungert. The observer's point of view: An extension of symbolic projections. In Andrew U. Frank, Irene Campari, and Ubaldo Formentini, editors, *Spatio-Temporal Reasoning*, volume 639 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 1992.

[90] Roberta L. Klatzky. Allocentric and egocentric spatial representations: Definitions, distinctions, and interconnections. In C. Freksa, C. Habel, and K. F. Wender, editors, *Spatial cognition - An interdisciplinary approach to representation and processing of spatial knowledge*, volume 1404, pages 1–17, Berlin, 1998. Springer-Verlag.

[91] Dan Klein, Joseph Smarr, Huy Nguyen, and Christopher D. Manning. Named entity recognition with character-level models. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 180–183. Edmonton, Canada, 2003.

[92] Rolf Klein. *Algorithmische Geometrie*. Addison-Wesley, 1997. KLE r 97:1 1.Ex.

[93] *The Keyhole Markup Language (KML)*. `http://earth.google.com/kml/`, accessed August 2006.

[94] Dietmar Kopitz and Bev Marks. *RDS: The Radio Data System*. Artech House Publishers, 1998.

[95] Benjamin Kuipers. Modeling spatial knowledge. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 292–298, 1977.

[96] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, USA, 1991.

[97] A. Lazanas and J.-C. Latombe. Landmark-based Robot Navigation. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 816–822, San Jose, California, 1992. AAAI Press.

[98] Willem J. M. Levelt. Perspective taking and ellipsis in spatial descriptions. In P. Bloom, M.A. Peterson, M.F. Garrett, and L. Nadel, editors, *Language and space*, pages 77–107, Cambridge, MA, 1996. MIT Press.

[99] Stephen C. Levinson. *Pragmatics*. Cambridge University, Cambridge, England, 1983.

[100] Gérard Ligozat. Reasoning about Cardinal Directions. *Journal of Visual Languages and Computing*, 9:23–44, 1998.

[101] *Real-Time Travel Information – More Services ... with the Right Equipment*. `http://www.locationintelligence.net/articles/559.html`, accessed April 2004.

[102] G. D. Logan and D. D. Sadler. A Computational Analysis of the Apprehension of Spatial Relations. In P. Bloom, M.A. Peterson, M.F. Garrett, and L. Nadel, editors, *Language and space*, pages 493–529, Cambridge, MA, 1996. MIT Press.

[103] Bernhard Lorenz. Bewegungsplanung für nicht-holonome Vehikel. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, 2002.

[104] Bernhard Lorenz and Hans Jürgen Ohlbach. Ontology Driven Visualisation of Maps with SVG. Deliverable A1-D5, Institute for Informatics, Ludwig-Maximilians-Universität München, 2005.

[105] Bernhard Lorenz and Hans Jürgen Ohlbach. Dynamic Data for Geospatial Reasoning - A Local Data Stream Management System (L-DSMS) and a Case Study with RDS-TMC. Deliverable A1-D6, Institute for Informatics, Ludwig-Maximilians-Universität München, 2006.

[106] Bernhard Lorenz, Hans Jürgen Ohlbach, and Laibing Yang. Ontology of Transportation Networks. Deliverable A1-D4, Institute for Informatics, Ludwig-Maximilians-Universität München, 2005.

[107] Kevin Lynch. *The Image of the City*. MIT Press, June 15, 1960.

[108] M. Fischer Manfred and Yee Leung. *Geocomputational Modelling: Techniques and Applications (Advances in Spatial Science)*. Springer, 2006.

[109] James Mayfield, Paul McNamee, and Christine Piatko. Named entity recognition using hundreds of thousands of features. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 184–187. Edmonton, Canada, 2003.

[110] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*. EATCS Monographs on Theoretical Computer Science. Springer, 1984. MEH k 84:3 2.P-Ex.

[111] Doreen Mizzi. A Mobile Navigational Assistance System Using Natural Language Generation. BSc FYP, Department of Computer Science & A.I., University of Malta, Malta, 2004.

[112] Martien Molenaar. *An Introduction to the Theory of Spatial Object Modelling for GIS*. CRC Press, 1998.

[113] Daniel R. Montello. Scale and multiple psychologies of space. In Andrew U. Frank and Irene Campari, editors, *Spatial Information Theory: A Theoretical Basis for GIS, International Conference COSIT '93, Marciana Marina, Elba Island, Italy, September 19-22, 1993, Proceedings*, volume 716 of *Lecture Notes in Computer Science*, pages 312–321. Springer, Heidelberg, 1993.

[114] Amitabha Mukerjee and Gene Joe. A qualitative model for space. In *AAAI*, pages 721–727, 1990.

[115] NAVTEQ, `http://www.navteq.com`. *Provider of digital map data*, accessed August 2006.

[116] Hans Jürgen Ohlbach. Calendar logic. In D.M. Gabbay, I. Hodkinson, and M. Reynolds, editors, *Temporal Logic: Mathematical Foundations and Computational Aspects*, pages 489–586. Oxford University Press, 2000.

[117] Hans Jürgen Ohlbach. Calendrical calculations with time partitionings and fuzzy time intervals. In H. J. Ohlbach and S. Schaffert, editors, *Proc. of PPSWR04*, number 3208 in LNCS. Springer Verlag, 2004.

[118] Hans Jürgen Ohlbach. Fuzzy Time Intervals and Relations – The FuTIRe Library. Forschungsbericht/research report PMS-FB-2004-4, Institute for Informatics, University of Munich, 2004.

[119] Hans Jürgen Ohlbach. Relations between fuzzy time intervals. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 44–51, Los Alamitos, California, 2004. IEEE.

[120] Hans Jürgen Ohlbach. The role of labelled partitionings for modelling periodic temporal notions. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 60–63, Los Alamitos, California, 2004. IEEE.

[121] Hans Jürgen Ohlbach. Computational Treatment of Temporal Notions – The CTTN-System. In *Proceedings of the Third Workshop on Principles and Practice of Semantic Web Reasoning, Dagstuhl, Germany (11th–16th September 2005)*. INRIA, 2005.

[122] Hans Jürgen Ohlbach. Fuzzy Time Intervals – The FuTI-Library. Forschungsbericht/research report PMS-FB-2005-26, Institute for Informatics, University of Munich, 2005.

[123] Hans Jürgen Ohlbach. GeTS - A Specification Language for Geo-Temporal Notions. Forschungsbericht/research report PMS-FB-2005-28, Institute for Informatics, University of Munich, 2005.

[124] Hans Jürgen Ohlbach. Implementation: GeTS - A Specification Language for Geo-Temporal Notions. Deliverable A1-D10-1, Institute for Informatics, Ludwig-Maximilians-Universität München, 2005.

[125] Hans Jürgen Ohlbach. Fuzzy Time Intervals – System Description of the FuTI-Library. In *Proceedings of the 4th Workshop on Principles and Practice of Semantic Web Reasoning , Budva, Montenegro (10th–11th June 2006)*, 2006.

[126] Hans Jürgen Ohlbach. GeTS - A Specification Language for Geo-Temporal Notions. In *Proceedings of 29th Annual German Conference on Artificial Intelligence, Bremen, Germany (14th–19th June 2006)*, 2006.

[127] Hans Jürgen Ohlbach and Dov Gabbay. Calendar logic. *Journal of Applied Non-classical Logics*, 8(4):291–324, 1998.

[128] Hans Jürgen Ohlbach, Klaus Schulz, and Felix Weigel. Geotemporal Reasoning: Basic Theory. Deliverable A1-D1, Institute for Informatics, Ludwig-Maximilians-Universität München, 2004.

[129] Dan Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, 2005. PhD Thesis, Institute for Informatics, University of Munich, 2005.

[130] Dan Olteanu, Tim Furche, and François Bry. An efficient single-pass query evaluator for XML data streams. In *SAC*, pages 627–631, 2004.

[131] David D. Palmer and David S. Day. A statistical profile of the named entity task. In *ANLP*, pages 190–193, 1997.

[132] Louis F. Pau. *Mapping and Spatial Modelling for Navigation*. Springer, Berlin, 1990.

[133] Axel Pinz. Consistent Visual Information Processing Applied to Object Recognition. In *The 6th International Fall Workshop on Vision, Modeling, and Visualization (VMV), Stuttgart, Germany*, 2001.

[134] David A. Randell, Zhan Cui, and Anthony Cohn. A spatial logic based on regions and connection. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 165–176. Morgan Kaufmann, San Mateo, California, 1992.

[135] Specification of the radio broadcast data system. `ftp://ftp.rds.org.uk/pub/acrobat/rbds1998.pdf`, April 1998.

[136] RBDS versus RDS – What are the differences and how can receivers cope with both systems? `ftp://ftp.rds.org.uk/pub/acrobat/rbds_vs_rds.pdf`, April 1998.

[137] Jochen Renz. *Qualitative Spatial Reasoning with Topological Information*. Springer, ISBN 3-540-43346-5, 2002. 13-15, 20p., 32-40.

[138] Jochen Renz and Bernhard Nebel. On the complexity of qualitative spatial reasoning: A maximal tractable fragment of the region connection calculus. Technical Report report00087, University of Freiburg, Germany, 23, 1997.

[139] Thomas Rickinger. Eine Client/Server-Architektur zur endgerätebasierten Ortung auf Basis von WLAN Scene Analysis. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, March 2006.

[140] H. Ritter. Self-Organizing Maps on non-euclidean Spaces. In S. Oja, E. & Kaski, editor, *Kohonen Maps*, pages 97–110. Elsevier, Amsterdam, 1999.

[141] Ralf Röhrig. Representation and Processing of Qualitative Orientation Knowledge. In *Proceedings of the 21st Annual German Conference on Artificial Intelligence (KI'97)*, pages 219–230, London, UK, 1997. Springer-Verlag.

[142] N. S. Ryan, J. Pascoe, and D. R. Morse. Enhanced reality fieldwork: the context-aware archaeological assistant. In V. Gaffney, M. van Leusen, and S. Exxon, editors, *Computer Applications in Archaeology 1997*, British Archaeological Reports, Oxford, October 1998. Tempus Reparatum.

[143] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[144] Bill Schilit and Marvin Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, 1994.

[145] Matthias Schmeisser. PlanML - A Markup Language for expressing Planning Results. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, December 2006.

[146] Alexander Scivos and Bernhard Nebel. Double-crossing: Decidability and computational complexity of a qualitative calculus for navigation. In Daniel R. Montello, editor, *COSIT*, volume 2205 of *Lecture Notes in Computer Science*, pages 431–446. Springer, 2001.

[147] *Spatial Data Transfer Standard (SDTS)*. `http://data.geocomm.com/sdts/`, accessed August 2006.

[148] J. Sharma, D. Flewelling, and M. Egenhofer. A qualitative spatial reasoner. In *The Proceedings of the 6th International Symposium on Spatial Data Handling*, 1994.

[149] Dan I. Slobin. From "thought and language" to "thinking for speaking". In J. J. Gumperz and S. C. Levinson, editors, *Rethinking linguistic relativity*, pages 70–96, Cambridge, MA, 1996. Cambridge University Press.

[150] Mark E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1(4):333–356, 1985.

[151] Edgar-Philipp Stoffel. A Research Framework for Graph Theory in Routing Applications. Diplomarbeit/diploma thesis, Institute for Informatics, LMU, Munich, 2005.

[152] Leonard Talmy. Lexicalization patterns: Semantic structure in lexical forms. In Timothy Shopen, editor, *Language typology and syntactic description, vol. 3*, pages 57–149, Cambridge, 1985. Cambridge University Press.

[153] Tele Atlas, `http://www.teleatlas.com`. *Provider of digital map data*, accessed August 2006.

[154] D. Tilbury, R. Murray, and S. Sastry. Trajectory Generation for the N-Trailer Problem Using Goursat Normal Form. Technical Report ERL-93-12, University of California, Berkeley, 1993.

[155] Sabine Timpf and Corinna Heye. Complexity of routes in multi-modal wayfinding. Technical report, University of Zurich, Department of Geography, 2002.

[156] Sabine Timpf and Werner Kuhn. Granularity transformations in wayfinding. In *Spatial Cognition*, pages 77–88, 2003.

[157] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 142–147. Edmonton, Canada, 2003.

[158] Barbara Tversky and Paul U. Lee. How space structures language. In Christian Freksa, Christopher Habel, and Karl Friedrich Wender, editors, *Spatial Cognition*, volume 1404 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 1998.

[159] Barbara Tversky and Paul U. Lee. Pictorial and verbal tools for conveying routes. In Freksa and Mark [61], pages 51–64.

[160] Laure Vieu. Spatial Representation and Reasoning in AI. In O. Stock, editor, *Spatial and Temporal Reasoning*, pages 3–40, Boston, MA, USA, 1997. Kluwer Academic Publishers.

[161] W3C, `http://www.w3.org/TR/owl-guide`. *OWL – The Web Ontology Language*, accessed August 2006.

[162] *flex: The Fast Lexical Analyzer*. `http://flex.sourceforge.net`, accessed August 2006.

[163] *OWLAPI – The OWL API Project*. `http://sourceforge.net/projects/owlapi`, accessed August 2006.

[164] *REWERSE - Reasoning on the Web with Rules and Semantics*. `http://rewerse.net`, accessed August 2006. Funded within the 6th Framework Programme project REWERSE number 506779.

[165] Felix Weigel. Enhancing User Interaction and Efficiency with Structural Summaries for Fast and Intuitive Access to XML Databases. In *Proceedings of the 3rd Ph. D. Workshop at the 10th International Conference on Extending Database Technology (EDBT)*, 2006.

[166] Felix Weigel, Klaus U. Schulz, Levin Brunner, and Eduardo Torres-Schumann. Integrated Document Browsing and Data Acquisition for Building Large Ontologies. In *Proceedings of the 10th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES), Invited Session "Engineered Applications of Semantic Web" (SWEA)*, 2006.

[167] Steffen Werner and Christopher Habel. Spatial Reference Systems. *Spatial Cognition and Computation*, 1(4):3–7, 1999.

[168] Steffen Werner, Bernd Krieg-Brückner, Hanspeter A. Mallot, Karin Schweizer, and Christian Freksa. Spatial cognition: The role of landmark, route, and survey knowledge in human and robot navigation. In *GI Jahrestagung*, pages 41–50, 1997.

[169] Alfred North Whitehead. *Process and Reality*. Macmillan, London, 1929. Corrected edition ed. by David Griffin and Donald Sherburne, New York: Free Press, 1978.

[170] Casey Whitelaw and Jon Patrick. Named entity recognition using a character-based probabilistic approach. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 196–199. Edmonton, Canada, 2003.

[171] *Universal Transverse Mercator coordinate system*. `http://en.wikipedia.org/wiki/UTM`, accessed August 2006.

[172] *World Geodetic System, WGS-84*. `http://en.wikipedia.org/wiki/World_Geodetic_System`, accessed August 2006.

[173] Kai Zimmermann. A Proposal for Representing Objects Sizes. In Pribbenow and Schlider, editors, *Proceedings of the Workshop on Spatial and Temporal Reasoning at the European Conference on Artificial Intelligence*, 1992.

[174] Kai Zimmermann. Enhancing Qualitative Spatial Reasoning Combining Orientation and Distance. In Andrew U. Frank and Irene Campari, editors, *Spatial Information Theory: A Theoretical Basis for GIS, International Conference COSIT '93, Marciana Marina, Elba Island, Italy, September 19-22, 1993, Proceedings*, volume 716 of *Lecture Notes in Computer Science*, pages 69–76. Springer, Heidelberg, 1993.

[175] Kai Zimmermann and Christian Freksa. Qualitative Spatial Reasoning Using Orientation. *Applied Intelligence*, 6(32):49–58, 1996.

Index