

Masters Program in **Geospatial Technologies**



***Framework development for providing accessibility
to Qualitative spatial calculi***

Sahib Jan

Dissertation submitted in partial fulfilment of the requirements
for the Degree of *Master of Science in Geospatial Technologies*

Masters Program in **Geospatial Technologies**



Dissertation Supervised

By

PhD Professor Angela Schwering
PhD Professor Marco Painho
PhD Student Malumbo Chaka Chipofya

Date of Submission: 28th February 2011

Dissertation submitted in partial fulfilment of the requirements
for the Degree of *Master of Science in Geospatial Technologies*

ACKNOWLEDGEMENT

I would like to thank the North-Rhine-Westphalian Ministry for Innovation, Science, Research and Technology (MIWFT), Germany for making it possible for me to pursue my higher education in dynamic learning environments through their prestigious grants. I would like to thank all the professors and other staff working on the Master of Science in Geospatial Technologies for their time and knowledge. Thanks to my three supervisors for their positive comments and support. Special thanks go to Mr. Malumbo Chipofya for his interesting idea and support that I perused in this thesis and thanks to my friend Mr. Amjad Saleem for his support and guidance. Thanks to all my colleagues in the master program for their friendship and support. I acknowledge the moral support of my fiancé during these studies.

Framework Development for Providing Accessibility to Qualitative Spatial Calculi

ABSTRACT

Qualitative spatial reasoning deals with knowledge about an infinite spatial domain using a finite set of qualitative relations without using numerical computation. Qualitative knowledge is relative knowledge where we obtain the knowledge on the basis of comparison of features within the object domain rather than using some external scales. Reasoning is an intellectual facility by which, conclusions are drawn from premises and is present in our everyday interaction with the geographical world. The kind of reasoning that human being relies on is based on commonsense knowledge in everyday situations. During the last decades a multitude of formal calculi over spatial relations have been proposed by focusing on different aspects of space like topology, orientation and distance.

Qualitative spatial reasoning engines like SparQ and GQR represent space and reasoning about the space based on qualitative spatial relations and bring qualitative reasoning closer to the geographic applications. Their relations and certain operations defined in qualitative calculi use to infer new knowledge on different aspects of space.

Today GIS does not support common-sense reasoning due to limitation for how to formalize spatial inferences. It is important to focus on common sense geographic reasoning, reasoning as it is performed by human. Human perceive and represents geographic information qualitatively, the integration of reasoner with spatial application enables GIS users to represent and extract geographic information qualitatively using human understandable query language.

In this thesis, I designed and developed common API framework using platform independent software like XML and JAVA that used to integrate qualitative spatial reasoning engines (SparQ) with GIS application. SparQ is set of modules that structured to provides different reasoning services. SparQ supports command line instructions and it has a specific syntax as set of commands. The developed API provides interface between GIS application and reasoning engine. It establishes connection with reasoner over TCP/IP, takes XML format queries as input from GIS application and converts into SparQ module specific syntax. Similarly it extracts given result, converts it into defined XML format and passes it to GIS application over the same TCP/IP connection.

The most challenging part of thesis was SparQ syntax analysis for inputs and their outputs. Each module in SparQ takes module specific query syntax and generates results in multiple syntaxes like; error, simple result and result with comments. Reasoner supports both binary and ternary calculi. The input query syntax for binary-calculi is different for ternary-calculi in the terms of constraint-networks. Based on analysis I, identified commonalities between input query syntaxes for both binary and ternary calculi and designed XML structures for them. Similarly I generalized SparQ results into five major categories and designed XML structures. For ternary-calculi, I considered constraint-reasoning module and their specific operations and designed XML structure for both of their inputs and outputs.

KEYWORDS

QSR -	Qualitative Spatial Reasoning
SparQ -	Spatial Reasoning Done Qualitatively
API -	Application Programming Interface
GQR -	Generic Qualitative Reasoner
JEPD -	Jointly Exhaustive and Pair wise Disjoint
GIS -	Geographic Information Systems
ALLEN -	Allen's Time Interval Calculus.
DRA-24 -	Dipole Relation Algebra
CARDIR -	Cardinal Direction Calculus
RCC-8 -	Region Connected Calculus-8
DCC -	Double Cross Calculus
CASL-	Algebraic Specification Language
RDBMS-	Relational Database Management System
CAMA-	Computer Assisted Mass Appraisal

NOMENCLATURES

1. Calculi's Variables and relations/inverse relations

#	Calculi	Variable	Relations /Inverse Relations
1	Allen's Time Interval Calculus	X, Y	Before(<), overlap(o), meet(m), during(d), start(s), finish(f), equal(=), After(>), meet inv(mi), during inv(di), start(s), finish inv(fi)
2	Dipole Relation Algebra	A, B, C	rrrr, rrrl, rrlr, rrl, rllr, rllr, rlll, lrrr, lrrl, lrl, llrr, llr, lll, ells, errs, lere, rele, slsr, srsl, lsel, rser, sese, eses
3	Cardinal Direction Calculus	A, B, C	north(N), north-east(NE), east(E), south-east(SE), South(S), south-west(SW), west(W), north-west(NW)
4	Region Connection Calculus	A, B, C	disconnected(DC), externally-connected(EC), part-of(P), proper-part(PP), proper-part inv(PPi), identical(EQ), overlap(O), partially-overlay (PO), tangential proper-part(TPP), tangential proper-part inv(TPPi), non-tangential proper-part(NTPP), non-tangential proper-part inv(NTPPi).
5	Double Cross Calculus	A, B, C	7-3, 6-3, 5-3, 5-2, 5-1, 0-4, b-4, 4-4, 4-a, 4-0, 1-5, 2-5, 3-5, 3-6, 3-7, dou tri

2. Calculi Operations

#	Operations	Description
1	union	$R \cup S = \{x x \in R \vee x \in S\}$
2	intersection	$R \cap S = \{x x \in R \wedge x \in S\}$
3	complement	$\bar{R} = U \setminus R = \{x x \in U \wedge x \notin R\}$
4	composition	$r \circ s := \{(A, C \in D \exists B \in D : (A, B) \in r \wedge (B, C) \in s\}$
5	converse	$\check{r} = \{(A, B (A, B) \in D \wedge (B, A) \in r\}$
6	inverse	$A, BrC \rightarrow B, Ainv(r)C$

TABLE OF CONTENTS

ACKNOWLEDGEMENT	i
ABSTRACT.....	ii
KEYWORDS.....	iii
NOMENCLATURES	iv
Allen's Time Interval Calculus.....	iv
1. Introduction.....	1
1.1 Qualitative Reasoning Background and Reasoning Engines.....	1
1.1.1 Motivation.....	2
1.1.2 Research Objectives.....	3
1.1.3 Outline of the Thesis.....	3
2. Literature Review	4
2.1 Understanding of QSR Calculi	4
2.1.1 Allen's Time Interval Algebra.....	4
2.1.2 Dipole Relation Algebra	5
2.1.3 Cardinal Direction Calculus.....	5
2.1.4 Region Connection Calculus	7
2.1.5 Double Cross Calculus.....	9
2.2 Application Areas of Qualitative Spatial Reasoning	11
2.2.1 Reasoning Application in GIS	12
2.2.2 Reasoning Application in Navigation.....	13
2.2.3 Reasoning application in Artificial Intelligence	14
3. Qualitative Spatial Reasoner.....	15
3.1 SparQ	15
3.1.1 Compute-relation Module.....	15
3.1.2 Qualify Module.....	16
3.1.3 Constraint-reasoning Module	16
3.1.3.1 Algebraic-closure.....	16
3.1.3.2 Scenario-consistency	16
3.1.3.3 Refine Operation.....	16
3.1.3.4 Extend Operation	16
3.1.4 Algebraic Reasoning Module	17
3.1.4.1 Consistency-checking	17
3.1.4.2 Qualification	17
3.2 GQR.....	17
3.3 Calculi Analysis Using SparQ Reasoner	18
3.3.1 Allen's Time Interval Algebra (AI) Analysis	18
3.3.1.1 Qualify Module.....	18
3.3.1.2 Compute-relation Module.....	19
3.3.1.3 Constraint-reasoning Module	20
3.3.1.4 Algebraic-reasoning Module	21
3.3.2 Dipole Relation Algebra (DRA-24) Analysis.....	21
3.3.2.1 Qualify Module.....	21
3.3.2.2 Compute-Relation Module	22

3.3.2.3	Constraint-reasoning Module	23
3.3.2.4	Algebraic-reasoning Module	23
3.3.3	Cardinal Direction Calculus Analysis.....	24
3.3.3.1	Qualify Module.....	24
3.3.3.2	Compute-relation Module.....	24
3.3.3.3	Constraint-reasoning Module	24
3.3.3.4	Algebraic-reasoning Module	25
3.3.4	Region Connection Calculus (RCC) Analysis.....	25
3.3.4.1	Compute-relation Module.....	25
3.3.4.2	Constraint-reasoning Module	25
3.4	Overview of Proposed Architecture	26
3.4.1	OpenJUMP Plug-In.....	26
3.4.2	Application Programming Interface (API)	27
4.	Application program Interface (API) design	28
4.1	Usecase Diagram	28
4.1.1	Plug-in Usecase Diagram.....	28
4.1.2	API Usecase Diagram.....	29
4.2	Process Diagram	30
4.3	SparQ Query Analysis	31
4.4	XML Design for SparQ Modules	31
4.4.1	Qualify Module.....	31
4.4.2	Compute-relation Module.....	32
4.4.3	Constraint-reasoning Module	32
4.4.3.1	Algebraic-Closure and Scenario-consistency Operation	32
4.4.3.2	Refine and Extend Operation.....	32
4.4.4	Algebraic-reasoning Module	34
4.4.4.1	Consistency Operation.....	34
4.4.4.2	Qualify Operation	34
4.4.5	Constraint-reasoning for Ternary Calculi	35
4.4.5.1	Algebraic-closure.....	35
4.4.5.2	Scenario-consistency	35
4.4.5.3	Refine and Extend.....	35
4.5	XML Conversion to SparQ Syntax.....	37
4.5.1	Qualify Query (XML) to SparQ Syntax	37
4.5.2	Constraint-reasoning Query (XML) to SparQ Syntax	38
4.5.3	Compute-relation Query (XML) to SparQ Syntax	40
4.5.4	Algebraic-reasoning Query (XML) to SparQ Syntax.....	41
4.6	XML Parsing.....	43
4.7	SparQ Result Analysis	43
4.7.1	Simple Text.....	44
4.7.2	Simple-text and Constraint-network.....	44
4.7.3	Constraint-network	45
4.7.4	Simple-relations	46
4.7.5	Syntax-errors.....	46
4.8	SparQ Result Conversion into XML	46
4.8.1	Syntax-errors.....	47
4.8.2	Simple-relations	47
4.8.3	Simple-text and Constraint-network.....	48

4.8.4	Simple Text.....	49
4.8.5	Constraint-network	49
5.	API Implementation.....	51
5.1	API Class Diagram	51
5.1.1	Plug-In Class Diagram.....	51
5.1.1.1	Extension Class.....	51
5.1.1.2	MyExtension Class	51
5.1.1.3	PlugInUI Class.....	52
5.1.1.4	userResult Class.....	53
5.1.1.5	AbstractPlugIn Class.....	54
5.1.2	API Class Diagram	54
5.1.2.1	EngineConnector Class.....	56
5.1.2.2	IQueryGenerator Interface.....	57
5.1.2.3	QueryGenerator Class.....	58
5.1.2.4	QueryReceiver Class.....	61
5.1.2.5	QuerySender Class.....	67
6.	Case Study and Demonstration.....	70
6.1	Case-Study	70
6.1.1	Integration of Reasoner with GIS	70
6.2	Demonstration on Spatial Data.....	73
7.	Conclusion, Shortcomings and Future Work.....	80
7.1	Conclusion	80
7.2	Shortcomings	81
7.3	Future Work.....	82
8.	Bibliographic Reference	84
	Student Declaration.....	86

INDEX OF FIGURES

Figure 1: Cardinal directions defined by projection and Cardinal Directions as cones .	6
Figure 2: RCC relations defined by Randell, Cui and Cohn, 1992 with interpretation..	8
Figure 3: Lattice of the subsumption hierarchy of the basic binary RCC relations (reproduced from Randell, Cui, Cohn, 1994	8
Figure 4: (a) Orientation line (ab) and (bc). (b) : qualitative orientation relations define by Freksa, 1992.....	9
Figure 5: (a): Single Cross Calculus. (b): Double Cross Calculus defined by Freksa, 1992. (c): possible qualitative orientation relations by combining (14-a) and (14- b).	10
Figure 6: (a): Qualitative orientations of given object “d” with respect to orientation line ab and cb. (b): position of object “d” is right-back (3) wrt. orientation line cd.....	10
Figure 7: Compute Qualify operation on Allen’s Time Interval Calculus relations and possible outcomes as relation between object A, B, C	19
Figure 8: Composition operation on Time Interval relations (m d) using SparQ.....	19
Figure 9: Possible time interval relation between (A and C) using Composition operation	20
Figure 10: Representation of qualify operation results (A rllr B), (A rllr C) and (B llrl C)	22
Figure 11: Relationship between first two objects (A rllr B) (A rllr C) defined by qualify operation.....	22
Figure 12: Result of Converse operation on relation (A rlll B), which is (llrl)	22
Figure 13: Composition operation between objects (A llrl B) (B rlll C) and possible relations between A and C	22
Figure 14: Composition Operation on RCC-8 relations (EC (A, B), EQ (B,C) and TPP (C,D)) and resultant relation between A and C (EC).....	25
Figure 15: Composition operation of RCC-8 relations TPP (A, B) and PO (B, C) and possible relations as result between A and C are (DC, EC, NTPP, PO, TPP).....	25
Figure 16: Framework Architecture with required components.....	27
Figure 17: Usecase diagram for Plug-in represents set of activates at application (JUMP) side.	28
Figure 18: Use-Case diagram for Application Programming Interface (API).....	29
Figure 19: API process diagram, major components and processes with flow directions.....	30
Figure 20: XML document and there integration with JAVA application H. Maruyama, 2002	43
Figure 21: Plug-in UML Class Diagram.....	52
Figure 22: API UML Class Diagram.....	55
Figure 23: Projection Based Model and Cone-based Model of Cardinal Direction introduced by Frank 1991	71
Figure 24: Orientation representation of objects P1, P2 and P3 using projection-based model Frank 1991in Open Street Map.....	72
Figure 25: OpenJUMP plug-in to send and receive data	73
Figure 26: Munster Street data with road-intersections as points (A, B and C)	74
Figure 27: Reasoning result in XML format on given road-intersections as points (A, B and C).	75
Figure 28: Line segment (A, B and C) representing Munster road, used for reasoning	77

INDEX OF TABLES

Table 1: Symbolic representation of relations defined in Allen Time interval calculus, 1983.....	5
Table 2: Cardinal directions composition table introduced by Frank, 1991.....	7
Table 3: Composition table based on possible elations between R1 (a, b) and R2 (b, c) of the basic binary RCC relations (reproduced from Randell, Cui, Cohn, 1994).	9
Table 4: Classification of modules specific results for both binary and ternary calculi	44

1. Introduction:

1.1 Qualitative Reasoning Background and Reasoning Engines

Qualitative spatial reasoning (Cohn and Hazarika, 2001) is the subfield of knowledge representation and symbolic reasoning that deals with then knowledge about an infinite spatial domains using a finite set of qualitative relations. For human, spatial reasoning is particularly powerful and accessible mode of cognition as human can perceive the space directly through various channels conveying distinct modalities.

Reasoning is an approach for dealing with commonsense knowledge without using numerical computation. Qualitatively models are used to model commonsense reasoning in the spatial domain and leads to a better interpretation of the final result. Spatial reasoning is presented in our everyday's interaction with the geographical world. In particular, we use orientation information or approximated distance to locate places in space. Spatial reasoning is the most common and basic form of intelligence. Qualitative knowledge is relative knowledge, where we obtain this knowledge on the basis of comparison of the features within the object domain rather than using some artificial external scales. It is considered to be closer to how human represents and reason about commonsense and incomplete knowledge of real world entities. Reasoning is an intellectual capacity, by which conclusions are drawn from premises. The kind of reasoning that human being rely on, is based on commonsense knowledge in everyday situations, as well as in very specialized domains is called commonsense reasoning (Stepankova, 1992).

Spatial reasoning is required for a comprehensive GIS and several research efforts have been underway to address this need (Abler 1987; NCGIA, 1989) and it is important that a GIS can carry out spatial tasks, including specific inferences based on spatial properties in a manner similar to a human expert and that there are capabilities that explain the conclusion to users in terms they can follow (Try and Bento, 1988). Today GIS does not support common-sense reasoning due to the limitation of how to formalize spatial inferences and the gap between GIS user's interests and GIS itself.

Reasoner like SparQ and GQR are open source applications that are used for representing space and reasoning about the space based on qualitative spatial relations and bring qualitative reasoning closer to application. These relations and certain operations like (composition, union, intersection and converse, etc) on them constitute a qualitative calculus. Using these operations, GIS users can be able to infer new knowledge (Renz and Nebel, 2007). During the last two decades, a multitude of formal calculi over sets of spatial relations like (Overlaps, left-of, north-of) have been proposed , focusing on different aspect of space (topology, orientation and distance, etc.) by introducing different kind of spatial objects (point, line and region). Integration of these reasoning engines with Geographic applications will enable users for common sensing geographic reasoning and reasoning as it is performed by human particularly in GIS, where user can use defined set of calculi to infer knowledge qualitatively based on given spatial data.

As a case study, I took three different areas of Munster city as points like (A, B and

C) to apply qualitative reasoning calculi like cardinal direction with the help of developed API. An API supports GIS users to integrate reasoning engine (SparQ) with spatial application like OpenJUMP. Based on coordinate values of given points, I generated XML query to extract its qualitative description with the help of the connected reasoner.

1.1.1 Motivation

Qualitative spatial reasoning engines like SparQ and GQR are toolboxes for representing space and reasoning about the space based on qualitative spatial relations. During last two decades, a multitude of formal calculi over sets of spatial relations have been proposed, focusing on different aspects of space and dealing with different kinds of objects by QSR community. SparQ aims at making these qualitative spatial calculi and developed reasoning techniques available in a single homogenous framework. It is designed for researcher's working on qualitative spatial reasoning, making experimental analysis and inferring knowledge qualitatively based on defined operations introduced in the calculi.

The main motivation behind this framework development is to provide accessibility to spatial calculi. A huge list of such spatial calculi has been discussed in literatures, examples include the Point Algebra (Vilain and Kautz, 1986) and Allen's Time Interval Algebra (Allen, 1983), the various Region Connection Calculi (Randell, Cui, and Cohn, 1992; Duntsch, Wang and McCloskey, 1999), the Intersection Calculi (Egenhofer, 1991; Egenhofer and Franzosa 1991), Cardinal Direction Calculi (Frank 1991; Skiadopoulos and Koubarakis, 2004), the Double Cross Calculus (Freksa, 1992), the OPRA calculi (Moratz, Dylla and Frommberger, 2005) and many more. Research in QSR is motivated by a wide variety of possible application areas including GIS, robotic navigation, high level vision, common-sense reasoning about physical system and specifying visual.

Integration of reasoning engines particularly SparQ that support both binary and ternary calculi mentioned above, with spatial applications like, (ArcGIS) provides basis for the design of intelligent GIS that supports spatial and temporal reasoning, modeling natural languages for representing spatial or temporal aspects in human machine interaction, query writing, data integration and inferring new qualitative knowledge. The data in GIS is generally represented quantitatively either in vector or raster formats and users often want to abstract away from this mass of numerical data and obtain a high level symbolic description of the data or want to specify a query in natural language like "Is Munster in Germany?", or qualitative approach in representation of spatial knowledge in navigational task where the problem is to find a route between the given starting and ending points. In everyday communication, orientation of spatial entities with respect to other spatial entities is usually given in the terms of a qualitative category like "to the left of" or "north-east" rather than numerical expression like 53 degree.

Although integration of spatial calculi with GIS provides imprecise data but provides verbal descriptions that support natural language queries like "find all the university departments north of Town Munster", or "A is close to B then to C". By developing

the framework that provides common platform to access the reasoning engines from spatial applications provides solution to interact spatial data qualitatively.

1.1.2 Research Objectives

The main object of this study is to develop platform independent framework using XML and JAVA, which provides solution to integrate the qualitative spatial reasoning engines particularly SparQ. The framework will act as middleware application between spatial application and reasoner using TCP/IP connection. The input queries from GIS application will be in simple plain-text (XML) format, and response from reasoner will be converted into XML with the help of API. XML data structure for inputs and output data enables users to interact with other application to automate queries. Similarly conversion of result in XML format is to make result meaningful, and enable to apply further processes based on defined XML tags. The main objective will be achieved by pursuing the following sub-objectives.

- i. Study reasoning engines to understand the facilities that are provided for reasoning on spatial data.
- ii. Analyze the reasoner specific calculi, their relations and operations like (composition, union, intersection, etc.) to identify the commonalities.
- iii. Based on analysis, both reasoner and defined calculi in it, I need to develop XML structure that is used for writing queries that reasoner can easily understand.
- iv. Development of Java based API that supports XML format inputs and outputs. The qualitative queries will be extracting from spatial application (GIS) and pass it to reasoner via API for reasoning and extract result from reasoner.

1.1.3 Outline of the Thesis

The research topic Framework development for Providing Accessibility to Qualitative Spatial Calculi consists of seven chapters. Chapter-1 contains introduction of thesis in general, including motivation and objectives of the research work. Chapter-2 discusses about the knowledge used in this thesis including brief introduction of Qualitative Spatial reasoning, reasoning calculi and application areas of Reasoning and Case Study. Chapter-3 includes brief introduction of reasoning engines (SparQ and GQR), analysis of calculi their relations and operations using reasoner and API architecture overview. In Chapter-4, I have tried describe API design includes XML structure based on SparQ modules syntax analysis. Usecase diagrams to represent activates supported by plug-in and API. API and plug-in Processes and their representations using process diagram. Chapter-5 contain API implementation included with java codes for defined classes and their functionalities. Chapter-6 describes case-study, calculus used for case-study explanation, demonstration of developed API using GIS application (OpenJUMP). Last chapter describes overall conclusion of the research work, shortcomings and future work, related with API improvements.

2. Literature Review

2.1 Understanding of QSR Calculi

Qualitative representation of the space abstracts from physical world and enables computers to make predictions about spatial relations between existing objects in specified domain, even when precise quantitative information is not available. Different aspects of the space and temporal information can be represented by using qualitative descriptions. The two important concepts of commonsense knowledge are time and space. Time, being a scalar entity, is very well suited for a qualitative approach as temporal reasoning (J. Renz, 2002; B. Nebel, 2001). Space is much complex than time due to its inherent multi-dimensionality like topological information, orientation information that describes entities and relationships between entities. These relations are based on a set of jointly exhaustive and pair wise disjoint (JEPD) basic relations, which are closed under several operations. It is possible to apply constraint based methods for reasoning over these relations.

The most popular reasoning method used in qualitative spatial reasoning are constraint based techniques that contain different constraints based calculi like Point Algebra (Vilain and Kautz, 1986) and Allen's Time Interval Algebra (Allen, 1983), the various Region Connection Calculi (Randell, Cui, and Cohn, 1992; Duntsch, Wang, and McCloskey, 1999), the Intersection Calculi (Egenhofer, 1991; Egenhofer and Franzosa, 1991), Cardinal Direction Calculi (Frank, 1991; Skiadopoulos and Koubarakis, 2004), the Double Cross Calculus (Freksa, 1992), the OPRA Calculi (Moratz, Dylla, and Frommberger, 2005), and many more (Cohn, A. G. and Hazarika, S. M., 2001). Constraint defined in qualitative reasoning is knowledge about entities or about the relationships between entities in the given domain. A constraint satisfaction problem (CSP) consists of a set of variables (V) over a domain (D) and a set of constraints (Φ).

2.1.1 Allen's Time Interval Algebra

The problem of representing temporal knowledge and temporal reasoning arises in a wide range of disciplines including computer science, artificial intelligence philosophy and linguistic. Allen, 1983 introduced a calculus based on intervals representing events, qualitative relationships in hierarchical manners between these intervals and algebra for reasoning about these relations. It gives a temporal representation that takes the notion of a temporal interval as primitive. The reference intervals are defined intervals that are equipped with a special property that affects the amount of computation involved. It is used to group clusters of intervals for which temporal constraints between each pair of intervals in the cluster are fully computed. Allen introduced 13 base JEDP relations for temporal reasoning and union operation on these base relations can produce 2^{13} relations which are given below (J. Allen, 1983)

The composition operation is used for reasoning with these relationships. Given qualitative relationships between (X) and (Y) and (Y) and (Z) entities, such a composition is defined as possible relations between (X) and (Z) entity. Since

composition of base relations provide 169 possible compositions for reasoning.

Names from Allen 1983	Symbol	Symbol for inverse
X before Y	<	>
X equal Y	=	=
X meets Y	m	mi
X overlays Y	O	oi
X during Y	d	di
X starts Y	s	s
X finishes Y	f	fi

Table 1: Symbolic representation of relations defined in Allen Time interval calculus, 1983

2.1.2 Dipole Relation Algebra

Most approaches in qualitative representation and reasoning about orientation information deals with point as basic entities. A DRA-24 is oriented line segment calculi by (R. Moratz, J. Renz and D. Wolter, 2001) based on determining start and an end points as the basic entities. Dipoles are denoted by A, B, C etc. The starting point of line segment is denoted by sA and ending point as eA. By defining set of dipoles, it is possible to specify many different relations among given entities based on the length of dipole. These relations are JEPD, means any two dipoles hold exactly one relation. The DRA-24 is based on two dimensional continuous space to identify locations and orientations of the different dipoles based on point lies on the left (l), to the right (r) or on the straight line through the reference dipole. The possible relations between dipoles and point can be $R \{l \ o \ r\}$.

$$A \ (R) \ sB \ \wedge \ A \ (R) \ eB \ \wedge \ B \ (R) \ sA \ \wedge \ B \ (R) \ eA$$

There are 14 relations that holds between defined point $\{sB, eB, sA, eA\}$, if these points are distinct. In order to obtain a relational algebra they also consider those relations where two dipoles share common points at starting and ending on one dipole which is denoted as $\{s, e\}$ e.g. $(sA \ sB)$ and $(eA \ eB)$. By using these additional dipole-point relations, they obtain 10 more relations. Altogether obtain 24 atomic relations which refer D-24 (set of 24 atomic relations), which are given below.

{rrrr, rrll, rrlr, rrlr, rllr, rllr, rllr, lrrr, lrrl, lrrl, llrr, llrl, llr, lllr, ellr, errs, lere, rele, slsr, srsl, lsrl, rser, sese, eses}.

Applying qualitative reasoning, using dipole relations they introduce constraint-based reasoning techniques which handles defined set of relations, these must be from a relation algebra, covered under defined operations like composition (o), intersection, complement (-) and converse (~).

2.1.3 Cardinal Direction Calculus

Qualitative reasoning is widely used by humans to understand, analyze and draw conclusions about spatial environment available in qualitative form, as in the case of test documents (Tobler and Wineberg, 1971), commonly in navigational tasks like route finding between given starting point and ending point with certain route properties.

As it is clear that qualitative approach loses some information but that may simplify

reasoning. It uses less precise data and therefore yields less precise result then the quantitative one. Which is highly desirable (Kuipers, 1983; NCGIA, 1989) because

- 1 Precision is not always desirable.
- 2 Precise quantitative data is not always available

Frank, 1991 introduced the cardinal direction calculi using general algebra oriented style; consist of following properties and operations.

1. Directional symbol D , which describes directions as set of $\{N, S, E, W\}$ or more extensive $\{N, NE, E, SE, S, SW, W, NW\}$
2. Direction as function between two points $P1$ and $P2$ in the plane that maps to a symbolical direction

$$dir : p \times p \rightarrow D$$

3. Direction between two close points can determine identity element.

$$dir : p \times p \rightarrow 0$$

4. Cardinal direction is order dependent, if direction is given between two points $p1$ and $p2$ then we can deduce direction between $p2$ and $p1$ by introducing inverse function (inv), known as *inverse direction*

$$inv (dir (p1 , p2)) \rightarrow dir (p2 , p1)$$

5. Direction between two contiguous line segments can combine by introducing combination operation (∞)

$$dir (p1 , p2) \infty dir (p2 , p3) = dir (p1 , p3)$$

6. Combination of more then two directions must be independent of the order in which they combine known as *associative law*

$$a \infty (b \infty c) = (a \infty b) \infty c = a \infty b \infty c$$

7. Direction from a point to itself $d(p1, p1)$ does not effect other direction known as identity

$$d \infty 0 = 0 \infty d = d$$

Frank introduced two prototypical concept of cardinal direction, first one is cardinal direction as cones (related to angular direction between the observation's position and destination point) and second is cardinal direct defined by projections, based on (pair-wise opposition and each pair divided the plane in to two half-planes).

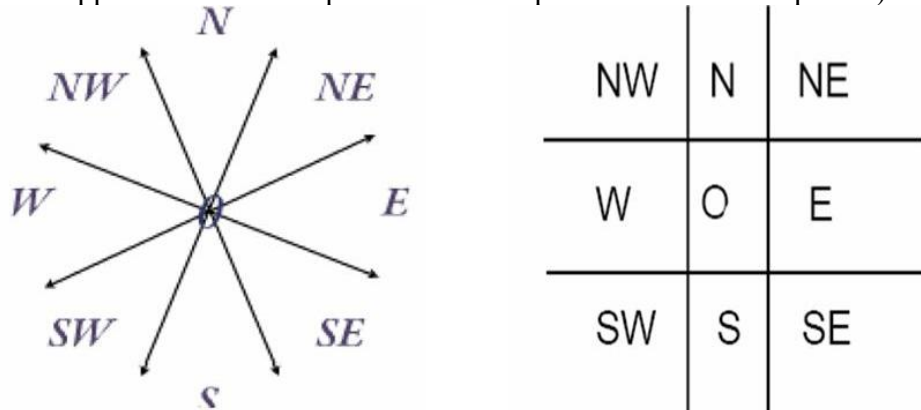


Figure 1: Cardinal directions defined by projection and Cardinal Directions as cones

According to cardinal direction as cones for every line segment, exactly one direction from the set of $\{N, NE, E, SE, SW, W, NW\}$ and identity element (0).

The operation quarter-turn anti-clockwise is defined as

$$q(N) = E, q(E) = S, q(S) = W, q(W) = N$$

Inverse operation in both 4 and 8 directions symbolically given as

$$e_8(d) = d$$

$$inv(d) = e_4(d)$$

In directions with natural zone plane is divided in 9 regions, central natural, 4 regions, the combination operation for each projection can be represent as

	N	NE	E	SE	S	SW	W	NW	0
N	N	n	ne	o	o	o	nw	nw	N
NE	n	ne	ne	e	o	o	o	n	NE
E	ne	ne	E	e	se	o	o	o	E
SE	o	e	e	SE	se	s	o	o	SE
S	o	o	se	se	S	s	sw	w	S
SW	o	o	o	s	s	SW	sw	w	SW
W	nw	o	o	o	sw	sw	W	w	W
NW	n	n	o	o	o	w	w	NW	NW
0	N	NE	E	SE	S	SW	W	NW	0

Table 2: Cardinal directions composition table introduced by Frank, 1991

2.1.4 Region Connection Calculus

The RCC calculus which has been developed at the University of Leeds over last few years, although the acronym “RCC” was originally derived from the last name initials of the authors of (Randell, Cui and Cohn, 1992). The fundamental approach of RCC is that extended entities i.e. regions in the space are taken as primary rather than the dimensionless points of traditional geometry and primitive relations between regions (Cohn, Brandon, J. Gooday and N. Mark, 1997). RCC theory is based on a single assumption of a primitive dyadic relation $C(x, y)$ where (x) connects with (y), in terms of points incident in regions, $C(x, y)$ holds when regions (x) and (y) share a common point. Using the relation $C(x, y)$ a basic set of dyadic relations are defined, which is reflective and symmetric.

$$\forall x C(x, x)$$

$$\forall xy [C(x, y) \rightarrow C(y, x)]$$

There are different degrees of connection between regions, from disconnected, equal, externally connected, partial overlapping, and one region being tangential part of the other or non-tangential part and so on. The defined theory by (Randell and Cohn, 1992) also supports a set of functions that define the Boolean composition of regions and a set of topological functions that allow for the explicit representation of interior, the closure and the exterior of particular regions (Randell, Cui and Cohn, 1992).

RCC-8 defines eight topological relations based on primitive relation $C(x, y)$, a basic set of dyadic relations which are JEPD, means that exactly one of these relations

holds between any two regions are defined $\{DC, EC, PO, TPP, NTPP, TPPi, NTPPi\}$.

Relation	interpretation	Definition of $R(x, y)$
(3) $DC(x, y)$	x is disconnected from y	$\neg C(x, y)$
(4) $P(x, y)$	x is a part of y	$\forall z[C(z, x) \rightarrow C(z, y)]$
(5) $PP(x, y)$	x is a proper part of y	$P(x, y) \wedge \neg P(y, x)$
(6) $EQ(x, y)$	x is identical with y	$P(x, y) \wedge P(y, x)$
(7) $O(x, y)$	x overlaps y	$\exists z[P(z, x) \wedge P(z, y)]$
(8) $DR(x, y)$	x is discrete from y	$\neg O(x, y)$
(9) $PO(x, y)$	x partially overlaps y	$O(x, y) \wedge \neg P(x, y) \wedge \neg P(y, x)$
(10) $EC(x, y)$	x is externally connected to y	$C(x, y) \wedge \neg O(x, y)$
(11) $TPP(x, y)$	x is a tangential proper part of y	$PP(x, y) \wedge \exists z[EC(z, x) \wedge EC(z, y)]$
(12) $NTPP(x, y)$	x is a nontangential proper part of y	$PP(x, y) \wedge \neg \exists z[EC(z, x) \wedge EC(z, y)]$

Figure 2: RCC relations defined by Randell, Cui and Cohn, 1992 with interpretation

The relations $\{P, PP, TPP$ and $NTPP\}$ are non-symmetrical and support inverses denoted by (Φ^{-1}) , where

$$\Phi \in \{P, PP, TPP, NTPP\}$$

The defined JEPD relations can be embedded in relational lattice, where symbol (T) interpreted as tautology and symbol (\perp) as contradiction. Ordering of these relations is on consumption that the weakest relations will connect directly to top and strongest will connect at the bottom.

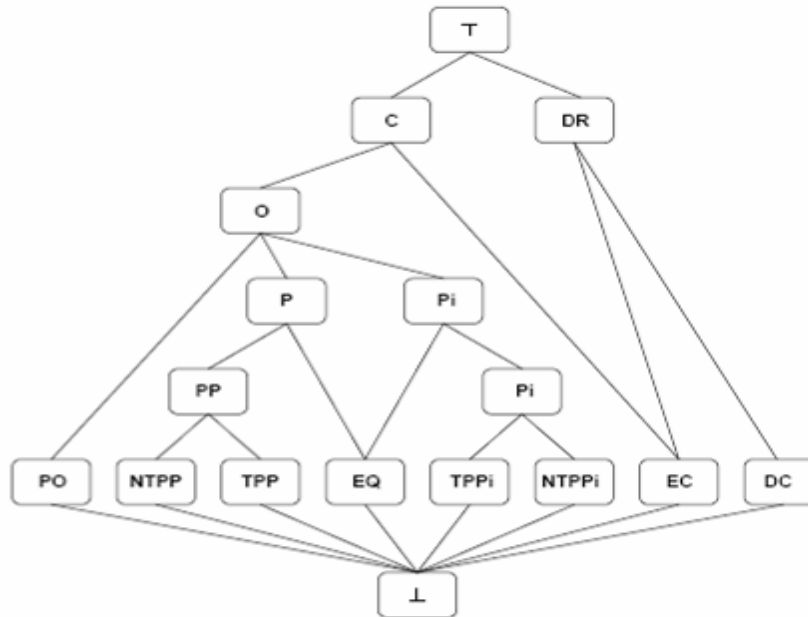


Figure 3: Lattice of the subsumption hierarchy of the basic binary RCC relations (reproduced from Randell, Cui, Cohn, 1994)

Below given compositional table is developed by (Cohn et al, 1993) for RCC-8. The composition table was developed from the initial idea of (Allen, 1983) is one of the decidable representations to support special decision procedures. Composition table for RCC-8 is developed by removing the relation (NTPI) from the original transitivity table and also replace relation (TPI) with equal relation ($=$), which is

represented as 8 by 8 matrix.

$R2(b,c) \backslash R1(a,b)$	DC	EC	PO	TPP	NTPP	TPPi	NTPPi	EQ
DC	no.info	DR,PO,PP	DR,PO,PP	DR,PO,PP	DR,PO,PP	DC	DC	DC
EC	DR,PO,PPi	DR,PO TPP,TPi	DR,PO,PP	EC,PO,PP	PO,PP	DR	DC	EC
PO	DR,PO,PPi	DR,PO,PPi	no.info	PO,PP	PO,PP	DR,PO,PPi	DR,PO PPi	PO
TPP	DC	DR	DR,PO,PP	PP	NTPP	DR,PO TPP,TPi	DR,PO PPi	TPP
NTPP	DC	DC	DR,PO,PP	NTPP	NTPP	DR,PO,PP	no.info	NTPP
TPPi	DR,PO,PPi	EC,PO,PPi	PO,PPi	PO,TPP,TPi	PO,PP	PPi	NTPPi	TPPi
NTPPi	DR,PO,PPi	PO,PPi	PO,PPi	PO,PPi	O	NTPPi	NTPPi	NTPPi
EQ	DC	EC	PO	TPP	NTPP	TPPi	NTPPi	EQ

Table 3: Composition table based on possible elations between $R1(a, b)$ and $R2(b, c)$ of the basic binary RCC relations (reproduced from Randell, Cui, Cohn, 1994).

2.1.5 Double Cross Calculus

Spatial orientation information, specifically directional information about the environment is directly available to animals and human beings through perception and is critical for establishing their spatial location and way finding, which is imprecise, partial and subjective (Freksa, 1992). Double Cross Calculus (DCC) can be seen as an extension of the Single Cross Calculus adding another perpendicular, is developed by Freksa, 1992, a new approach for qualitative temporal reasoning. It can be extended in order to presents and reason of orientation information of greater complexity and exploitation of conceptual neighborhood between related qualitative relations. The conceptual neighborhood information can bring computational advantages like incomplete knowledge handling and uncertainty control in case of fuzzy base knowledge.

In qualitative reasoning we relate entities of different dimensionality within a domain of a certain dimensionality (Freksa, 1992), e.g. one-dimensional domain length, which is spanned by two 0-dimensional entities (points with in the 1-dimensionssional domain). We can relate with two 0-dimensional entities on the basis of “less”, “equal” and “greater”. Directional orientation in 2-D space is a 1-D feature, which is determined by an oriented line, it is based on the order set of two points. Thus it describes the orientation of line (bc) with respect to the orientation of line (ab) and orientation can be describe as four distinct orientation relations like same, opposite, left and right.

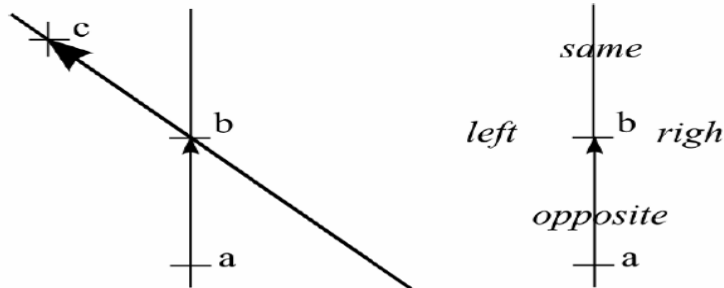


Figure 4: (a) Orientation line (ab) and (bc). (b) : qualitative orientation relations define by Freksa, 1992

By sub-segmenting the 2-D space into two semi-planes perpendicular to the orientation line, we can get eight augmenting qualitative orientation relations namely straight-front (0), right-front (1), right-natural(2), right-back (3), straight-back (4), left-back (5), left-natural (6) and left-front (7).

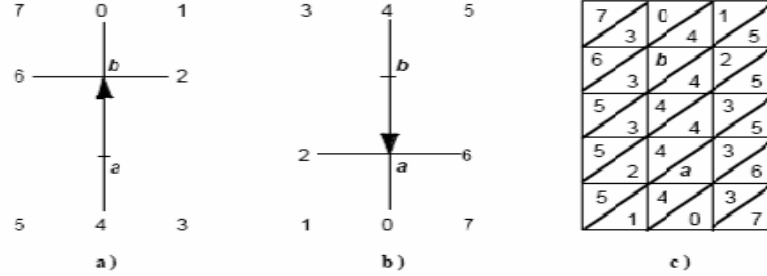


Figure 5: (a): Single Cross Calculus. (b): Double Cross Calculus defined by Freksa, 1992. (c): possible qualitative orientation relations by combining (14-a) and (14-b).

By combining both orientation labels from figures (a) and (b), we can produce 15 qualitative orientations and locations given in figure (c), each region corresponding to an orientation wrt. (B), is represented in the upper left of the corresponding matrix field and orientation wrt.(A), is represented as lower right of the corresponding matrix field, used to structure conceptual neighborhood schema. The neighborhood relation can be $\{5/3, 5/2, 5/1\}$ not relations like $\{3/5, 3/7\}$.

An introduced orientation-based representation framework is used for qualitative spatial reasoning, illustrated as an example. The task is based on knowing the qualitative spatial relations of vector (bc) to vector (ab) and the relations of vector (cd) to vector (bc), and inferring relations of vector (bd) to original reference vector (ab).

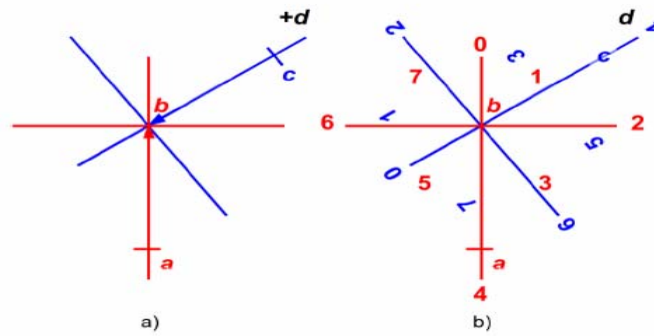


Figure 6: (a): Qualitative orientations of given object "d" with respect to orientation line ab and cb. (b): position of object "d" is right-back (3) wrt. orientation line cd.

By combining above mentioned two figures, we can infer location of (d) using the following notations: "*c is right-front (1) wrt. Vector (ab)*", "*d is right-back (3) wrt. Vector (cd)*" and finally "*d is left-front (7) wrt. Vector (ab)*" or "*d is straight-front (0) wrt. Vector (ab)*" or "*d is right-front (1) wrt. Vector (ab)*". Composition table of given 8*8 orientation relations is use to infer position of (d) with respect to vector (ab).

2.2 Application Areas of Qualitative Spatial Reasoning

The principle of qualitative reasoning is to represent not only our everyday commonsense knowledge about the physical world, but also the underlying abstractions used by engineers and scientists, when they create quantitative models. Using such knowledge and appropriate reasoning methods, a computer could make predication, diagnose and explain the behavior of physical system in qualitative manners, even a precise quantitative description is not available (A. G. Cohn, 1997). Most of the knowledge about time and space is qualitative in nature; specifically visual knowledge about space and the knowledge which we retrieved from memory. Qualitative descriptions are very powerful in this situation, when there are many items to be distinguished, there are many relations that can be established for distinguishing them, when there are few, only few are needed for their distinction (C. Freksa, 1991).

Representation and analysis of spatial information is an essential problem and the space has multidimensional aspects, which we can not represent by single scalar quantity. Spatial information about the space is available as qualitative information or as a large amount of quantitative data, which required efficient analysis in qualitative form. Qualitative spatial reasoning has wide variety of application areas including geographical information systems (GIS), robotic navigation , high level vision, the semantic of spatial preposition in natural language, engineering design, reasoning about physical situation and specifying visual language syntax and semantics (Cohn,1997).

QSR has concentrated on representation and extraction aspects of spatial information. Various computational Para-diagrams are investigated including constrain-reasoning, compute-relations based on composition table defined for specific calculus. Most of the qualitative spatial calculi are developed with respect to different aspects of the space like Allen's Time Interval Algebra based temporal reasoning, the Region Connection Calculi, used for representation and reasoning on topological relationship between given entities, Cardinal Direction calculi used for reasoning on position and orientation of entities and many more.

Most of the reasoning calculi have defined composition tables that represents possible relationships between entities, Relations are given as a set of JEPD relations which represents as $R1(a, b)$ and $R2(b, c)$. The composition table provides a useful and efficient way of reasoning and have certainly been the most commonly used form of qualitative spatial inference but they do not necessarily subsumes all forms of desired reasoning, therefore other more general form of reasoning known as constraint-based reasoning like algebraic-closure, scenario-consistency for network consistency verification and algebraic-reasoning has been introduced. Qualitative spatial reasoning can used in many application areas from everyday life in which spatial knowledge plays a role, particularly the areas in which uncertain and incomplete knowledge exists such as.

1. Geographic information system.
2. Cognitive maps and path finding, which might serve to control robots.
3. Computer aided systems for architectural design.
4. Design and user interfaces.

5. Natural language information to give directions.
6. Determination of the 3-D structure of molecules.

2.2.1 Reasoning Application in GIS

Geographic Information System (GIS) is a common platform that is used to represent and analyze geospatial information stored in both raster and vector data formats. Naïve Geography is the field of study that is concerned with formal modeling of common-sense geographic world. GIS users are interested to abstract away bulk of spatial data from the mass of numerical data, and obtain a high level symbolic description of the data or want to specify a query in a way which is essentially or at least qualitative. It comprises a set of theories upon which next generation GISs can be built based on qualitative spatial reasoning.

The concept of Naïve Geography is basis for the design of intelligent GISs that will act and response as a person would. Central to Naïve Geography is the area of spatial and temporal reasoning. Many concepts of spatial and temporal reasoning have become important research areas in a wide range of application domains such as physics, medicine, biology and geography and more specific on the reasoning about geographic space and time, subsequently called geographic reasoning (Egenhofer, D. Mark, 1995). There are different aspects of the space and it is very important for reasoning and representations to decide what kinds of spatial entities we will admit (commit to a particular ontology of space), and also need to consider different ways of describing the relationships between these entities like consideration of their topology, size, distance or their shape.

Naïve geography which is the basic form of human intelligence, people use spatiotemporal reasoning in daily common life and employ methods to infer information about their environment and about the consequences of changing over location in the space. It is very important to understand, how people handle their environment to incorporate naïve geographic knowledge and reasoning into GIS. The concepts and methods that people use to infer information about the space and time are important for interaction between human and computerized GISs (Egenhofer, David Mark, 1995). Today GIS does not support common-sense reasoning due to the limitations for how to formalize spatial inferences. In order to make GIS useful for wide range of people, it is important to focus on common-sensing geographic reasoning, reasoning as it is performed by people. Integration of named distance and qualitative orientation approach in GIS is appropriate example, where qualitative reasoning calculi like cardinal direction calculi and named distance approach is used to identify location of the entity in the terms of (N, S, W, E) and distance between entities in the terms of (Near, Medium, Far and very Far).

The scope of GIS applications can illustrated with respect to

1. Government and public service
2. Business and service planning
3. Logistics and transportation
4. Environment

Tex mapping and assessment is classical example of value of GIS in the local government level. A GIS is used to collect and manage the geographic boundaries

and associated information about properties with the help of computer Assisted Mass Appraisal (CAMA) system which is RDBMS and is responsible for sale analysis, evaluation, data management and administration and generating notice to owner. Both GIS and CAMA system are based on RDBMS technology and use a common identifier to effect linkage between a map features and a property records.

The tax assessment task involve a geographic database query to locate all scales of similar properties within the predetermined distance of a given property and to compare the values of all comparable properties with in the predetermined search radius. Integration of reasoner and geo-statistical features in GIS is useful for the assessment and comparison of the characteristic of these properties qualitatively and quantitatively, their properties, like plot size, scales price, neighborhood status can elevate on the basis of conceptual neighborhood and clustered properties. By integrating QSR with Geographical information system (GIS) user can infer new spatial knowledge using qualitative spatial query language i.e. by applying cardinal direction calculus on defined spatial data of plot; we can infer possible cardinal relations between given plots instead of defining all possible cardinal relation in the RDBMS.

2.2.2 Reasoning Application in Navigation

Application of orientation-based qualitative spatial reasoning is the process of determining a location in the space on the basis of our own location and the location which we know. Orientation information particularly directional information about environment is directly available to animals and human being through perception and is crucial for establishing their spatial location and way-finding (Freksa, 1992). In order to deal with such imprecise and partial information needs appropriate methods like exploration of conceptual neighborhood relations for presentation and processing. Two relations in the representation are conceptual neighbors, when an operation in the represented domain can result in a direct transition from one relation to the other.

The conceptual neighborhood structures are important since they intrinsically reflect the structure of the represented world with their operations (Furbach, 1985). Such presentations allow us to implement reasoning strategies. The Directional orientation in 2-dimensional space is 1-dimensional feature represented by an orientated line, which is specified by ordered set of two defined points. The specification of orientation can be described through qualitatively like same, opposite, left and right. On the other hand orientation-based inferences on the basis of ordered oriented line (vector) between defined set of points used to infer the directional relation of location from original reference line. Such inferences can be relevant for way-finding processes, like route description from a known location to unknown place in the terms of orientation information and also direct route to particular place.

Kuipers and his research group proposed several navigation and mapping systems for large scale space, based on four level semantic hierarchy of description for representation, namely sensorimotor interaction, procedural behavior, topological mapping and metric mapping (M. Escrig, F. Toledo, 1998). The sensorimotor and procedural levels are capable to solve navigation problem while topological and

metric levels provides the most powerful problem-solving capabilities (to drive the paths). Their proposed computational models are TOUR model (for structured outdoor environments), NX model (for structured indoor environments) and QUALNAV model (for open structured environments like mountainous terrain).

2.2.3 Reasoning application in Artificial Intelligence

Qualitative spatial reasoning is popular in Artificial Intelligence (AI) cause of following factors that influence.

1. Some time high-precision quantitative measurements are not useful for analysis of complex systems.
2. AI gains experience and confidence for representation and processing non-numerical knowledge.
3. Qualitative spatial knowledge requires less storage as compare to quantitative knowledge and easy to understand because qualitative knowledge is near to natural language.

In quantitative representation of space, the reference system is specified by a ruler which measures spatial objects and their locations. In a three-dimensional Cartesian reference system, for example, three rulers are arranged orthogonally, objects and their locations are specified by a measurement on each of these rulers (Christian Freksa, Ralf Rohrig, 2000). In contrast qualitative representations we do not required scales on the rulers, since they do not employ metrics, and comparisons are carried out among entities using same reference system.

Incomplete spatial information is one of the major problems in spatial reasoning, especially for robot navigation, where the location of objects and shape is not sufficiently determined. Qualitative approach to navigation can distinguish different kinds of partial information according to the degree of determination and can design specialized representation for them; metric information, for instance, is more determined then topological in the sense those matrix invariants put strong constraints on location and shape of objects then topological invariants. Most of formalisms devised for the qualitative representation of spatial information have some means for describing the linear ordering of points, like as ordering information is strictly stronger then topological information but strictly weaker then metrical information (Christoph Schlieder, 1991) as the isometric transformation of the plane either preserve the orientation of all point triples (e.g. rotation) and topological transformation generally do not preserve order information about given set of points.

The concept of landmark models particularly panorama is introduced as a solution to the qualitative spatial representation problem and specifically to the navigation problems, where a location is defined with reference to the landmarks around it and changes in panorama due to the transformation between regions. The navigation strategy in panorama, based exclusively on ordering information to find optimal solution as well as used as a guideline for efficient search in maps.

3. Qualitative Spatial Reasoner

Qualitative spatial reasoning (QSR) is an established field of research pursued by investigators from many disciplines including geography, philosophy, computer science and AI. Qualitative spatial representation techniques are especially suited for applications that involve interaction with human as they provide an interface based on human spatial concepts. A multitude of spatial calculi has been proposed during last two decades focusing on different aspects of the space but the amount of applications employing QSR techniques are comparatively small due to the following factors.

1. Choosing the right calculus for particular application is a challenging task, especially for the people not familiar with QSR.
2. Calculi are specified partially and no implementation is made available and investigated theoretically.
3. Integration of calculi with applications required serious efforts to identify appropriate calculus for application; this is time-consuming and error-prone process particularly writing down large composition table.

Qualitative spatial reasoner is toolbox that provides a platform for making the calculi and reasoning techniques developed in the QSR community available. Some of these reasoning engines provide both binary and ternary spatial calculi and capabilities to infer knowledge based on composition table.

3.1 SparQ

SparQ is a toolbox developed at university of Bremen (Dylla et al., 2006; Wallgrun et al., 2006) it supports both binary and ternary calculi, released under the GPL license for representing the space and reasoning about the space based on qualitative spatial relations. It is developed within the R3-(Q-Shape) project of the SFB/TR8 Spatial Cognition.

SparQ provides a platform for making binary and ternary calculi and reasoning techniques available. It is an application program that can be used directly, provides a broader range of services including capabilities of integration with the spatial applications like ArcGIS and OpenJUMP (over TCP/IP socket connectivity). It contains composition table and operations based on JEPD relations and supports the most common tasks including qualification, computing with relations, and constraint-based reasoning for an extensible set of spatial calculi. The calculi are defined in the algebraic specification language (CASL). The current version SparQ mainly focuses on calculi from the area of reasoning about orientation of point objects or line segments and designed as open framework of single program component with text-base communication, that support accessibility to integrate new calculi. SparQ introduced four modules like qualify, compute-relation, constraint-reasoning and algebraic-closure.

3.1.1 Compute-relation Module

Compute-relation in SparQ allows computation of operations defined in the specific calculus; it takes parameters as operations like (union, intersection, complement, composition and converse/inverse) and set of relations that holds between given entities.

Composition (N, N)

Example shows the composition operation on given relation north (N)

3.1.2 Qualify Module

Qualify module in SparQ takes quantitative scene descriptions as argument and returns qualitative descriptions of scene between the given entities. Each object description is a tuple consisting of the object identifier (A, B and C) and object parameters, which is depend upon type of object used for reasoning. Quantitative description of line segments (dipole) is represented as (Id, xs, ys, xe, ye). Here (xs)(ys) are starting point of X and Y and (xe)(ye) are ending points of entity X and Y.

Example:

((A -2 0 8 0) (B 7 -2 2 5) (C 1 -1 4.5 4.5)

Similarly 2-D Point can be represented as

(A X-axis Y-axis)(B X-axis Y-axis)

(A 25.344 0.9955)(B 25.6544 0.9933)

3.1.3 Constraint-reasoning Module

Constraint represents knowledge about entities or about the relationships between entities, it is used to restrict the defined domain of 2, 3, 4,....., n variables.

The constraint-reasoning module reads a description of a constraint network defined qualitatively that may includes disjunctions and may be inconsistence. It performs operation to identity network consistence on the basis of implemented operations like

3.1.3.1 Algebraic-closure

Algebraic-closure operation is used to enforce path-consistence on the constraint network. A constraint-networks and constraint satisfaction problem (CSP) is said to be path consistency if and only, for any partial instantiation of any two variables satisfying the constraints between the two variables. It is possible for any third variable to extend the partial instantiation to this third variable satisfying the constraints between the three variables

Example:

Algebraic-closure((A N B)(B E C))

3.1.3.2 Scenario-consistency

Operation scenario-consistence is provided as argument, the constraint-reasoning module check if the algebraically closed scenario exists for given networks. It use backtracking algorithm to generate all possible scenarios that are algebraic closed.

Example:

Scenario-consistency all ((A po B)(B tpp C))

3.1.3.3 Refine Operation

Refine operation in constraint-reasoning returns conjunction of two given constraint-networks. Basically it applies intersection operations on corresponding constraints.

3.1.3.4 Extend Operation

Extend operation in constraint reasoning module returns disjunction. It merges two constraint networks by uniting corresponding constraints.

3.1.4 Algebraic Reasoning Module

Algebraic reasoning module is used for reasoning about real-valued domain using techniques of algebraic geometry. The main service offer by this module is to providing a consistency checking mechanism for constraint network that is based on the relation semantics only. It provides possible results like

3.1.4.1 Consistency-checking

- a. Satisfiable
The network is proved to be satisfiable
- b. Not Satisfiable
The network is proven to be unsatisfiable
- c. Cannot Decide
Neither one of the above proofs succeeded

b. Operation analysis

The algebraic-reasoning module commend is used to verify the defined operations by iterating over all basic operations to analyze the given operation (e.g. composition, inverse) and summarize the result in the form of “OK” if operation verified, “CANNOT INCLUDE” if operation not agreeable with the calculus semantic, “MAY ALSO INCLUDE” if operation is not listed in table, may possible missing and “MUST ALSO INCLUDE” bases relation is not listed in the operation table.

3.1.4.2 Qualification

Qualification function is part of algebraic-reasoning used to identifying the scenario that qualifies with out supplying a designated qualification function. Each object description is a tuple consisting of the object identifier (A, B and C) and object parameters, which is depend upon type of object used for reasoning. Quantitative description of line segments (dipole) is represented as (Id, xs, ys, xe, ye). Here (xs)(ys) are starting point of X and Y , (xe)(ye) are ending points of entity X and Y.

Example:

a-reasoning dra-24 qualify all (A 25.344 0.9955)(B 25.6544 0.9933)

3.2 GQR

Generic Qualitative reasoner (GQR) is free software distribute under the terms of the GNU license. It was developed at the University of Freiburg as a solver for binary qualitative constraint-networks; it takes calculi descriptions and one or more constraint-networks as input to solve the path consistency method (basically returns a network that is (semantically) equivalent to the original one) and heuristic backtracking. The concept of path consistency method is not sufficient to decide consistency of constraint-networks, therefore GQR uses chronological backtracking with 2-way or n-way branching that trying out different instantiations of the constraints containing disjunctions of the base relations (cf. Ladkin and Reinefeid, 1997, Nebel 1997). By identifying tractable subclass of calculus (set of relations closed under intersection and composition, for which the path consistency method decides consistency) one can speed up the reasoning time.

The concept of consistency method and backtracking search may benefit from heuristics about which part of the constraint-network is to be processed next. GQR

supports arbitrary binary constraint calculi, new calculi can be added in it, by specifications in a simple text format or XML file format. Currently GQR supports arbitrary binary constraint calculi developed for spatial and temporal reasoning, such as calculi from the RCC family, the Intersection Calculi, Allen's Time Interval Algebra, Cardinal Direction Calculi and Calculi from the OPRA family.

Reasoning in GQR is based on purely syntactical definition of qualitative calculi. A calculus is defined by a non-empty finite set "B" of symbols and element of "B" are referred to the set of base relations. A unary function of "B" can be represent as Converse operation on each base relation can be represented as

$$: \tilde{B} \rightarrow B$$

Binary function is function based on the pair of basic relations their composition and a distinguished element (id) of (B) known as identity relations

$$\circ : B \times B \rightarrow B$$

$$(\tilde{\tilde{a}}) = a, id \circ a = a, a \circ id = a$$

GQR is written in object oriented programming language C++, there for user can add new heuristics quite easily. New qualitative calculi can define by writing in simple text or XML file to integrate in the reasoner.

3.3 Calculi Analysis Using SparQ Reasoner

3.3.1 Allen's Time Interval Algebra (AI) Analysis

Using SparQ reasoner, I analyzed operations on base relations defined by Allen in Time Interval Calculus. The purpose of analyzing Time Interval Algebra is to identify SparQ query syntax and their possible outcomes as result based on defined modules in reasoner like (qualify constraint-reasoning, compute-relation and algebraic-reasoning and their sub-operations). As Allen's Time Interval Algebra is a binary calculus therefore it supports only binary operations like (union, intersection, converse, complement and composition).

3.3.1.1 Qualify Module

Qualify module in SparQ is used to turn a given quantitative geometric scene descriptions into qualitative scene descriptions composed of base relations from a particular calculus. Each object description is a tuple, consists of objects (entities) identifiers like (A, B, C, D.... etc.) and its modules specific parameters, which is dependent upon entity type support by the reasoner. SparQ supports five basic entities including 1d-point, interval, 2d-point, 2d-oriented-point and dipole. The coordinate values of given entity are taken as entity type. These values can be specified as integer, float or rational numbers.

\$./sparq qualify allen all "((A 4 8) (B 8 10) (C 3 10))"

Result: ((A m B) (A d C) (B f C))

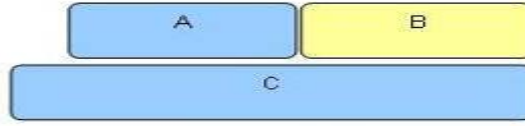


Figure 7: Compute Qualify operation on Allen's Time Interval Calculus relations and possible outcomes as relation between object A, B, C

In the above mentioned example, i analyzed qualify operation. Qualify module takes entity identifiers (A, B, C) and parameters of entity type, which is based on selection of entity-type. Here each entity takes two parameters as interval (start, end) values. Qualify module converts given quantitative description in qualitative description that represents possible relations ((A *meet* (m) B) (A *during* (d) C) (B *finish* (f) C)) between set of entities defined in Allen's Time Interval Algebra.

3.3.1.2 Compute-relation Module

The compute-relation module allows computing with the operations defined in the calculi. The operations are categorized as binary and ternary operations. The selection of operations depends upon calculus that supports. Compute-relation takes basic relations as input parameters, the number of input relations depend upon arity of operation like converse operation which takes single base relation as input parameter.

In the below mentioned example, I compute different operations like (UNION, INTERSECTION, COMPOSITION) on the set of base relations.

&. /sparq compute-relation allen union b m o

Result: (b m)

By applying compute-relation module, I computed union operation on given relations between entities like (A *before* (b) B), (B *meet* (m) C) and (C *overlap* (o) D). From the given result, i came to know the union of the given relations can be (*meet* (m), *before* (b)).

&. /sparq compute-relation allen intersection b b

Result: (b)

The result of intersection operation between two same base relations is same base relation, like intersection of two relation *before* (b) and *before* (b) itself relation *before* (b).

&. /sparq compute-relation allen composition m d

Result: (d o s)

$$r \circ s := \{ (A, C \in D \mid \exists B \in D : (A, B) \in r \wedge (B, C) \in S) \}$$

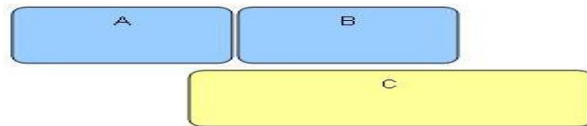


Figure 8: Composition operation on Time Interval relations (m d) using SparQ

The possible relationship between A and C is given as (d o s)

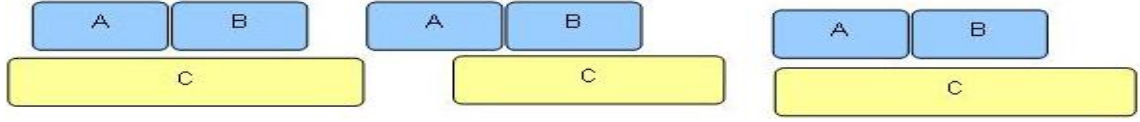


Figure 9: Possible time interval relation between (A and C) using Composition operation

The result of Composition operation on given base relations between entities like (*A meet (m) B*) and (*B during (d) C*) is (*d, o, s*), it means that the possible relations between entities “A” and “C” can be *during (d)*, *overlay (o)* and *start(s)*, which is represented as

(*A during (d) C*) or (*A overlay (o) C*) or (*A start(s) C*).

3.3.1.3 Constraint-reasoning Module

Constraint-reasoning module reads constraint-networks descriptions defined as qualitative scene. The module contains particular kind of operations like algebraic-closure and scenario-consistency and actions like refine and extend, that determines what kind of consistency check is performed.

a. Algebraic-closure:

Algebraic-closure operation is used in constraint-reasoning to enforce path-consistency on constraint-networks and detect consistency, algebraic closure and inconsistency of a given constraint-network.

\$./sparq constraint-reasoning allen algebraic-closure “((A s B) (B d C) (C m D))”

Result: Modified Network.

((C m D) (B b D) (B d C) (A b D) (A d C) (A s B))

In this example I applied constraint-reasoning module and algebraic-closure operation on the constraint-networks (*A start(s) B*) (*B during (d) C*) (*C meet (m) D*) to identify the path-consistency. As a result, reasoner modified given networks to make it algebraic-closed by adding extra base relations between entities like (*A before (b) D*), (*B before (b) D*) and (*A during (d)*

b. Scenario-Consistency:

Scenario-consistency is an operation of constraint-reasoning module. It use back-tracking and forward-tracking algorithms to check algebraically closed scenarios for the given constraint-networks. Scenario consistency operation takes additional parameters to identify and returns path-consistency scenario that found. The value “First” returns the first path-consistency scenario of given network and “ALL” returns possible path-consistency scenarios as set of disjunction of base relations. In case of inconsistency constraint network, reasoner will respond comments as “NOT CONSISTENT”

\$./sparq constraint-reasoning allen scenario-consistency first “((A s B) (B d C))”

Result: ((B s C) (A s C) (A s B))

In above SparQ query using scenario-consistency operation with return value “first” on constraint-networks (*A start(s) B*) (*B during (d) C*) to identify path-consistency. As result the reasoner returns path-consistency network by introducing additional base relation between entity A and C like, (*A start(s) C*), which is consistent scenario.

c. Refine:

Refine is action that returns the conjunction of two constraint-networks in constraint-reasoning.

\$./sparq constraint-reasoning allen refine “((A (d f) B))” “((A d B))”

Result: ((A (d) B))

The conjunction of two constraint-networks (A (during (d), finishes (f) B) (A during (d) B) is (A during (d) B).

d. Extend:

Extend action returns disjunction of two constraint-networks in the constraint-reasoning.

\$./sparq constraint-reasoning allen extend “((A (d f) B))” “((A d B))”

Result: ((A (d f) B))

Extend operation on constraint-networks (A during (d), finishes (f) B) (A during (d) B) returns disjunction relations as constraint on given networks, which is (A during (d), finishes (f) B).

3.3.1.4 Algebraic-reasoning Module

Algebraic-reasoning module in SparQ provides facilities of reasoning about real-valued domain using techniques of algebraic geometry. It provides two basic operations consistency and qualification. Consistency operation is used for consistency checking based on operation tables and algebraic relations specification. The reasoner returns result as string like, “SATISFIABLE, CAN NOT DECIDE or Not SATISFIABLE”

Operation in algebraic-reasoning provides functionality to qualify scenario without supplied a designated qualification function

a. Qualification:

\$./sparq a-reasoning allen qualify “((A 4 8) (B 4 10))”

Result: ((A (S) B))

The above mentioned qualify operation query using algebraic-reasoning module returns qualitative description (A start(s) B) of given quantitative description (A 4 8) (B 4 10), which are interval values (starting and ending) of entity “A” and “B”.

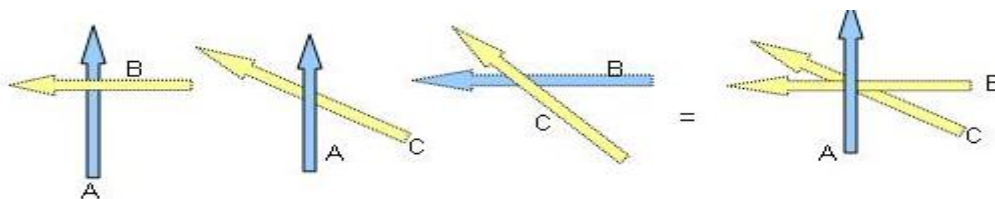
3.3.2 Dipole Relation Algebra (DRA-24) Analysis

3.3.2.1 Qualify Module

\$./sparq qualify dra-24 all “((A -2 0 8 0) (B 7 -2 2 5) (C 1 -1 4.5 4.5))”

Result: ((A rllr B)(A rllr C) (B lrrl C))

In above SparQ query, I applied qualify operation to get qualitative descriptions on given quantitative descriptions. It take two control modes represented as “ALL” and “FIRST2ALL” that returns all possible qualitative descriptions or first-two qualitative description on given 2-dimension dipole entities A, B and C. It takes four parameter values as (sx, sy, ex and ey). Here sx (starting value of x), sy (starting value of y), ex (ending value of x) and ey (ending value of y). The qualitative description represents basic relations between given entities.



3.3.2.3 Constraint-reasoning Module

a. Algebraic- Courser:

\$./sparq constraint-reasoning dra-24 algebraic-closure “((A rllr B) (A (rele rlll) C) (B rlll C))”

Result: Unmodified network

((B (rlll) C) (A (rele rlll) C) (A (rllr) B))

The algebraic-closure operation using constraint-reasoning module that returns unmodified network as result on given constraint-networks “((A rllr B)(A (rele rlll) C) (B rlll C)), means that the given network has path-consistency.

b. Scenario-consistency

\$./sparq constraint-reasoning dra-24 scenario-consistency all “((A rele C) (A ells B) (C errs B))”

Result: ((C (errs) B) (A (ells) B) (A (rele) C))

1 scenario found, no further scenarios exist

The scenario-consistency operation on constraint-network “((A rele C) (A ells B) (C errs B))” returns path-consistency as single scenario, which is ((C (errs) B) (A (ells) B) (A (rele) C)).

c. Refine:

\$./sparq constraint-reasoning dra-24 refine “ ((A (rele rllr) B))” “((A rllr B))”

Result: ((A (rllr) B))

Refine operation on constraint-networks “((A (rele rllr) B))” “((A rllr B))” returns conjunction of two given networks which is represented as relation between given entities ((A (rllr) B)).

d. Extend:

\$./sparq constraint-reasoning dra-24 extend “ ((A (rele rllr) B))” “((A rllr B))”

Result: ((A (rele rllr) B))

Extend operation returns disjunction of two given constraint-networks as possible relations between entities ((A (rele rllr) B)).

3.3.2.4 Algebraic-reasoning Module

a. Consistency Checking:

\$./sparq a-reasoning dra-24 consistency “((A rllr B) (A ells C) (B lrrl C))”

Result: Not Satisfiable

Consistency operation using algebraic-reasoning module on given constraint-networks “((A rllr B)(A ells C)(B lrrl C))” used to analysis path-consistency and it returns result as Not Satisfiable, which means given network is not consistent.

b. Qualification:

\$./sparq a-reasoning allen qualify all “((A -3 0 8 0) (B 8 0 4 8))”

Result: ((A (ells) B))

Qualify operation using algebraic-reasoning module returns qualitative description as a possible base relation (A (ells) B) between given entities.

3.3.3 Cardinal Direction Calculus Analysis

3.3.3.1 Qualify Module

\$./sparq qualify cardir all “((A 4 0) (B 4 8))”

Result: ((A (s) B))

Qualify operation on given entity “((A 4 0) (B 4 8))”. Entities are represented as 2d-point, that takes two parameters (ax, ay, bx, by) as quantitative description. The reasoner returns qualitative description as base relation between given entities (A south(s) B) defined in cardinal direction calculus.

3.3.3.2 Compute-relation Module

\$./sparq compute-relation CARDIR intersection N,N

Result : (n)

Intersection operation on base relation (North(N), North(N)) defined in cardinal direction calculus returns North(N) as result, which means intersection of relation (N,N) itself equal to relation North (N).

\$./sparq compute-relation CARDIR union N,NE

Result: ()

Similarly union of two base relations North (N) and North-east (NE) is empty relation.

\$./sparq compute-relation CARDIR composition S E

Result: (se)

Composition of two cardinal relations (South (S) and East (E) is South-East (se).

3.3.3.3 Constraint-reasoning Module

a. Algebraic- Closure:

\$./sparq constraint-reasoning cardir algebraic-closure “((A N B) (B W C))”

Result: Modified Network

((B (w) C) (A (nw) C) (A (n) B))

Algebraic-closure operation on constraint-networks “((A north(N) B) (B west(W) C))” returns modified network by adding additional cardinal direction relation between entities (A north-west(nw) C) to make given network algebraic-closed, The modified constraint-network as result given by reasoner is ((B west(w) C) (A north-west(nw) C) (A north(n) B)).

b. Scenario-consistency:

\$./sparq constraint-reasoning cardir scenario-consistency all “((A N B) (B E C) (A NE C))”

Result: ((B (e) C) (A (ne) C) (A (n) B)) 1 scenario found.

Scenario-consistency operation using return parameter “ALL” on constraint network “((A north(N) B) (B east(E) C) (A north-east(NE) C))” that returns one path-consistent scenario ((B east(e) C) (A north-east(ne) C) (A north(n) B)).

c. Refine:

\$./sparq constraint-reasoning cardir refine “((A N NE B))” “((A NE B))”

Result: ((A (ne) B))

Refine operation returns disjunction relation between two given constraint network,

which is $((A \text{ north-east}(ne) B))$

d. Extend:

$\$./sparq \text{constraint-reasoning cardir extend } "((A \text{ N NE } B))" "((A \text{ NE } B))"$

Result: $((A (n \text{ ne}) B))$

Extend returns conjunction relation between given constraint-network $"((A \text{ north } (N), \text{ north-east } (NE) B))"$ $"((A \text{ north-east } (NE) B))"$, which is $((A (\text{north } (n), \text{ north-east } (ne) B))$

3.3.3.4 Algebraic-reasoning Module

a. Qualification:

Cardinal direction calculus does not provide algebraic speciation.

b. Qualify

Return same result as defined in Qualify Module.

3.3.4 Region Connection Calculus (RCC) Analysis

3.3.4.1 Compute-relation Module

$\$./sparq \text{compute-relation rcc-8 composition ec eq tpp}$

Result: (ec)

The composition operation using compute-relation module on base relations between entities $(A \text{ externally-connected } (ec) B)$ $(B \text{ equal } (eq) C)$ $(C \text{ tangential-proper-part } (tpp) D)$ defined in RCC, returns relationship between given entities "A" and "D". Which is $(A \text{ externally-connected } (ec) D)$.

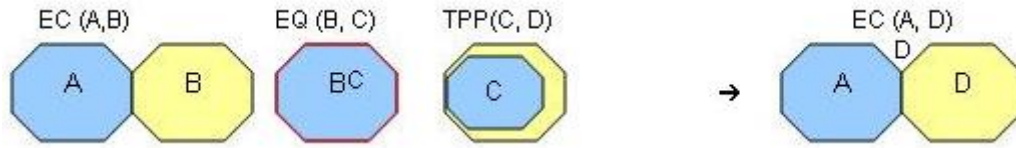


Figure 14: Composition Operation on RCC-8 relations $(EC(A, B), EQ(B, C) \text{ and } TPP(C, D))$ and resultant relation between A and C (EC)

$\$./sparq \text{compute-relation rcc-8 composition tpp po}$

Result: $(dc \text{ ec } ntp \text{ po } tpp)$

Similarly composition operation on relations $(A \text{ tangential-proper-part}(tpp) B)$ $(B \text{ partially-overlap}(po) C)$ returns possible set of base relations (dc, ec, ntp, po, tpp) between given entities "A" and "C". The possible set of relations between entity A and C are given below.

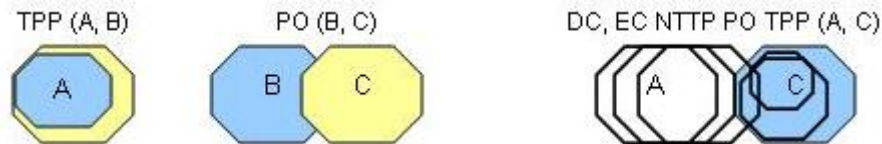


Figure 15: Composition operation of RCC-8 relations $TPP(A, B)$ and $PO(B, C)$ and possible relations as result between A and C are (DC, EC, NTP, PO, TPP)

3.3.4.2 Constraint-reasoning Module

a. Algebraic- Closure:

$\$./sparq \text{constraint-reasoning rcc8 algebraic-closure } "((A \text{ ec } B)(B \text{ TPP } C))"$

Result: Modified network

$((B (tpp) C) (A (ec ntpo po tpp) C) (A (ec) B))$

Algebraic-closure operation on given two constraint-networks $((A (n ne) B))$, returns modified network by adding additional possible set of relations between entities $(A (ec ntpo po tpp) C)$, that satisfied path-consistency.

b. Scenario-Consistency:

$\$./sparq constraint-reasoning rcc8 scenario-consistency first "((A ec B)(B TPP C))"$

Result: $((B (tpp) C) (A (tpp) C) (A (ec) B))$

Scenario-consistency operation check path-consistency on given constraint-networks $"((A ec B)(B TPP C))"$ and returns networks that is algebraically closed. Here reasoner added additional relation between entities $(A (tpp) C)$ to make given network path-consistent and algebraically closed.

c. Refine:

$\$./sparq constraint-reasoning rcc8 refine "((A (ec po) B))" "((A po B))"$

Result: $((A (po) B))$

Refine operation returns conjunction of given constraint-network $"((A (ec po) B))"$ $"((A po B))"$ which is $((A partially-overlap(po) B))$

d. Extend:

$\$./sparq constraint-reasoning rcc8 extend "((A (ec po) B))" "((A PO B))"$

Result: $((A (ec po) B))$

Extend returns disjunction by uniting corresponding constraints in given constraint-networks like $((A externally-connected (ec), Partially-overlay (po) B))$, that is obtained from two constraint-networks $"((A (ec po) B))" "((A po B))"$.

3.4 Overview of Proposed Architecture

Framework (API) architecture is based on three major components including.

1. OpenJUMP (Plug-in)
2. Application programming interface (API)
3. Reasoner (SparQ)

3.4.1 OpenJUMP Plug-In

OpenJUMP is an open source GIS application, developed by Canadian Companies Vivid Solutions and Refractions Research. The name JUMP is an abbreviation for Java Based Unified Mapping Platform. The application supports reading and writing shape files and simple GML file format as other GIS applications. It supports different data format including GML, SHP, DXF, JML, MIF, TIFF and postGIS etc. JUMP provides functionality to extend application by writing their own plug-ins, cursor tools, renderer etc. with the help of built-in extension class.

Java based Plug-in is used as extension, consists of different functionalities like *Input_Text_Field* (to insert XML format reasoning query), *Output_text_area* (display results in XML format received from reasoner), send buttons (send query to reasoner) connection button (establish connection with reasoning engine via TCP/IP) and disconnect button (break established connection). OpenJUMP provides facilities to convert spatial data in GML format that can easily converted in to SparQ specific

XML query for reasoning through developed plug-in. Plug-in application contains set of Java classes that are used to access API via http or can directly integrate within developed Plug-in application. Here in this thesis for demonstration, I integrated API with in Plug-in application, later with minor modification, it is possible to use it as web-based API to access over HTTP for reasoning.

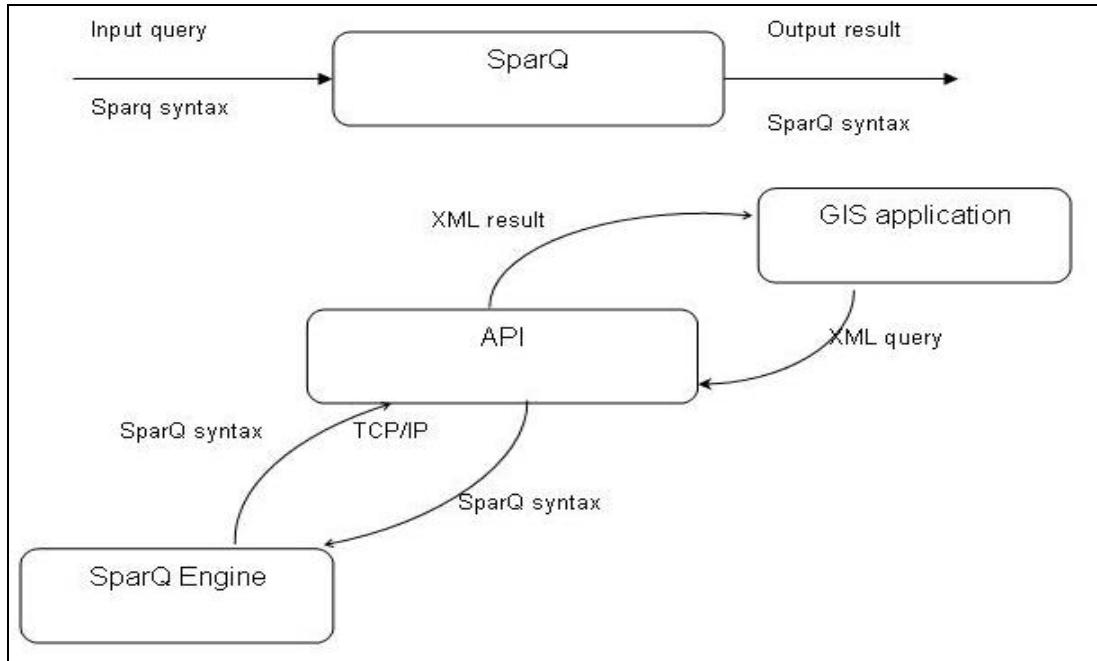


Figure 16: Framework Architecture with required components

3.4.2 Application Programming Interface (API)

API is middleware application, based on set of Java classes and XML files. XML is extremely portable language to the extent that it can be on large network with multiple platforms like internet and can be use on handhelds or palmtops. API facilitate GIS users to define own tags with respect to application needs. Development of web based applications using XML and Java is emerging technologies that facilitates the design and implementation of business-to-business (B2B) applications such as web application server, simple object access protocol (SOAP), web services and data binding

API contains particular set of rules and specifications that a software program can follow to access services and resources provided by another application that implements designed API. The purpose to design API in this study is to integrate qualitative spatial reasoning engines with GIS applications, particularly integration of reasoning engine (SparQ) with GIS application (openJUMP). It will serve as an interface between these two applications, similar to the way, that user interface provides facility to interaction between human and computers. The defined API used XML file as a common understandable language for both applications, where user can send query in XML format via application plug-in and retrieve results in XML format from reasoner. API process given XML file, validate query with defined XML schema, establishing connection with GIS application and reasoning engine through TCP/IP connection.

4. Application program Interface (API) design

4.1 Usecase Diagram

Usecase diagram is a description of a system's (API) behavior as it request, that originates from out that system. It is collection of diagram and text that together document how users expect to interact with the system. The given below usecase diagram represents overall structure of the API and plug-in as a simple usecase that representing system, actors, associations and dependencies of the API. The purpose of usecase diagram is to provide a high-level explanation of the relationship between the system (API) and the outside world.

4.1.1 Plug-in Usecase Diagram

Plug-in is small java based application that provides interface for OpenJUMP users to access reasoning engine. The usecase diagram represents general activates that performed by GIS users at application side by using plug-in includes.

Open application is general activity that performed by user to open and close application (openJUMP).

Load plug-in activity needs some events and functions that activates and load plug-in within application, plug-in provide interface, contains set of facilities including textbox (to write/generate query for reasoning), connection button (use to open connection with reasoning engine) and close button (use to close established connection with reasoner).

Report Error activity is used to display error messages related with the loading and reading activities.

Send Query activity forwards generated XML query based on spatial data to reasoner through API. API process XML data and convert it in to SparQ specific syntax.

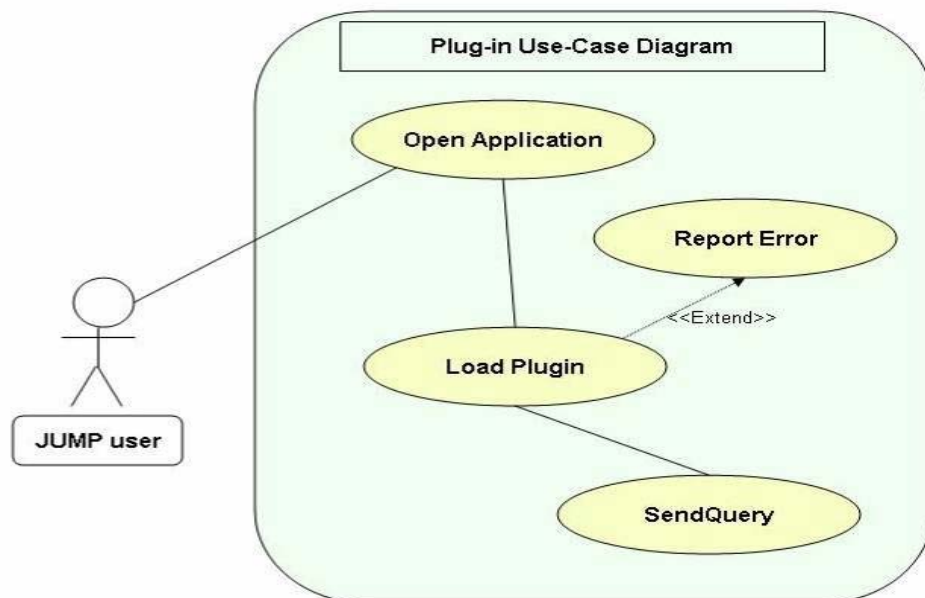


Figure 17: Usecase diagram for Plug-in represents set of activates at application (JUMP) side.

4.1.2 API Usecase Diagram

Open_Connection_With_Reasoner: activity establishing connection with reasoning engine (SparQ) through TCP/IP connection.

Process_XML_Query: converts given query in XML format with the help of defined XML tags. DOM parser is used to parse XML file and generate document tree that is used to access elements and their attributes.

Report Error: Activities used to generate reports related with define query syntax and socket connection.

Validate_XML_Query: It contains set of functions that are used to validate incoming parsed XML file with define schema.

XML_Schema: is a schema file contains set of rules with respect to reasoner's understandable syntax. During validation process API is requesting XML schema file syntax for validation of incoming queries.

Convert_Query_In_ReasoningSyntax: activity converts XML query in reasoner specific syntax and pass it over TCP/IP connect to the reasoner.

Send_Query_To_Reasoner: passes generated final query to specific reasoner through TCP/IP connection.

Extract_Result_And_Forward_to_Plugin: activity contains functions that extract results from reasoner and convert into the XML format. It passes XML format result to the end user.

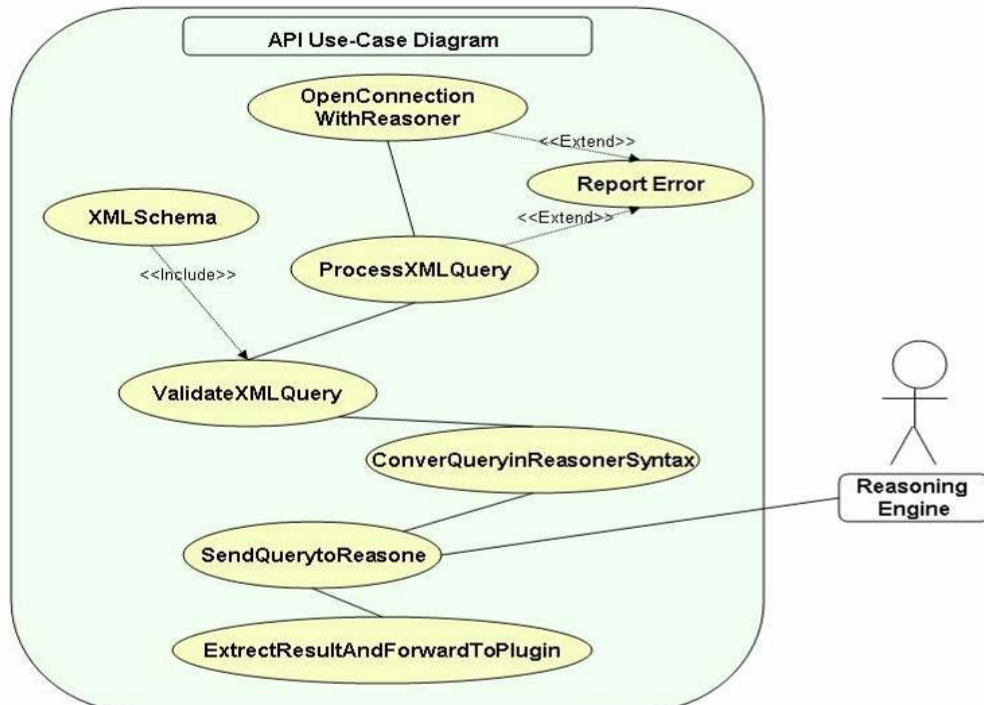


Figure 18: Use-Case diagram for Application Programming Interface (API)

4.2 Process Diagram

The process diagram, is diagram commonly used in software engineering to indicate the general flow of processes used in the system. The defined process diagram consists of four major components like (application, Plug-in, API and reasoning engine). The arrow line with defined labels indicates process-type, process direction and the process sequence.

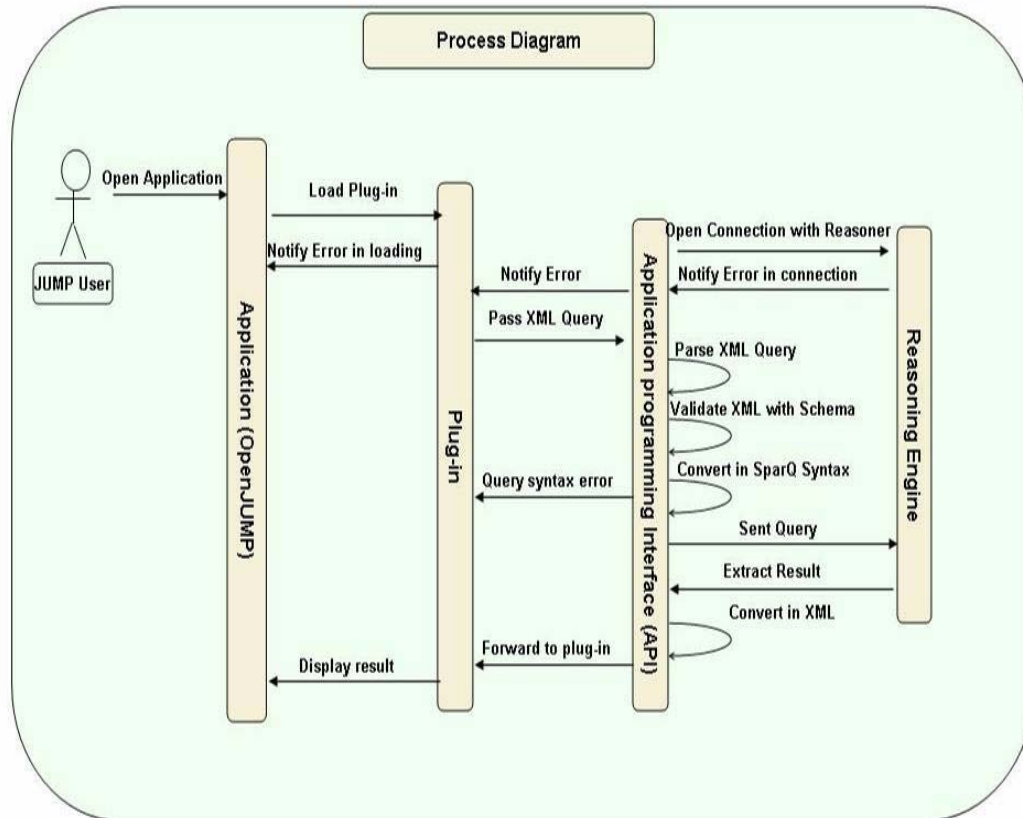


Figure 19: API process diagram, major components and processes with flow directions

4.3 SparQ Query Analysis

API is Java application, contains set of methods that provides functionalities to interact with the GIS application and reasoning engine without human interactions. API design requires analysis of reasoner to understand reasoner specific query syntaxes and result structures. In API, XML files are defined based on the analysis of reasoner's syntaxes. Reasoning calculi are analyzed using all modules and their specific operation introduced in SparQ, to identify their syntax commonalities. Each module in SparQ takes particular syntax, as a sequence of commands and module specific parameters. The general syntax of SparQ query is given below.

\$./sparq <module-name><calculus-name><module specifies parameters>

Module-Name: presently SparQ is supporting four types of modules that can be used to perform different type of reasoning including Qualify, Compute-Relation, Constraint-Reasoning and Algebraic-Reasoning.

Calculus-Name: parameter represents type of qualitative spatial calculus used for reasoning on given data. SparQ supports both binary and ternary calculi which are already defined as simple text format in it.

Module specific parameter: contains set of operations, relations and constraint-networks use for reasoning. Each defined operation takes module specific parameters.

The purpose of SparQ analysis is to understand each module and its specific syntax used for writing input queries. Based on the analysis, I categorized possible input queries and designed XML files for each module and module specific operation. XML data structure contains set of tags like module-name, calculi-name, operations, relations and modules specific parameters including control-modes, returns, entity types and constraint-networks etc. The standard tags in XML are used to generate reasoner specific queries and to validate it with defined XML schema.

4.4 XML Design for SparQ Modules

Based on analysis, I categorized all possible input queries for both binary and ternary calculi used for reasoning using SparQ and design standard XML structure for module specific queries.

4.4.1 Qualify Module

XML design for Qualify module contains sequence of tags including (*module_name*, *calculus_name* and *type*, *control-Mode*, *entity* and *entity type*) based on the qualify module specific syntax and required parameters for reasoning. Qualify Module takes quantitative descriptions of scene as input parameters and generates qualitative description, which is defined as entity tag. The entity type is forwarded as attribute and entity id as entity value.

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="qualify">
  <calculus type="binary" name="dra-24">
    <controlMode>all</controlMode>
    <entity type="-3 0 8 0">A</entity>
    <entity type="8 0 4 8">B</entity>
  </calculus>
</module>

```

4.4.2 Compute-relation Module

XML design for compute-relation module consists of standard tags including (module_name, calculus_name, operations and relations). It is designed as standard structure that reasoner accept to compute-relations by applying both binary and ternary operations. In compute-relation module, the defined operations take calculi specific relations as input parameter and infer possible relations between entities as output.

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="compute-relation">
  <calculus type="binary" name="rcc-8">
    <operation type="binary">composition</operation>
    <relation>dc</relation>
    <relation>ec</relation>
    <relation>tpp</relation>
  </calculus>
</module>

```

4.4.3 Constraint-reasoning Module

Module is used for constraint reasoning based on given constraint-networks. It contains four constraint reasoning specific operations like (Algebraic-reasoning, scenario- consistency, refine and extend). Each operation has specific syntax and takes certain type of parameters as input. Based on analysis I design XML for each operation that generates module and operation specific syntax.

4.4.3.1 Algebraic-Closure and Scenario-consistency Operation

Both Operations takes same syntaxes as sequence of tags like *module_name*, *calculus_name*, *operations* and *constraint-network* specific parameters (*entity_name* and *relations*) except return parameter. Return parameter (ALL/FIRST) is required by scenario-consistency operation to identify and extract result as scenario-consistency based on given constraint-networks or on first-two constraint-networks.

4.4.3.2 Refine and Extend Operation

Both operations takes same syntax as a sequence of tags like (*module_name*, *calculus_name*, *constraint-reasoning specific operation* and *constraint-networks as parameters*) to get conjunction or disjunction relation as result on given networks.

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="binary" name="cardir">
    <operation type="constraint-reasoning">algebraic-closure</operation>
      <entity>A</entity>
      <relation>N</relation>
      <entity>B</entity>
      <entity>B</entity>
      <relation>N</relation>
      <entity>C</entity>
    </calculus>
  </module>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="binary" name="cardir">
    <operation type="constraint-reasoning">scenario-
consistency</operation>
      <return>all</return>
      <entity>A</entity>
      <relation>N</relation>
      <entity>C</entity>
      <entity>A</entity>
      <relation>S</relation>
      <entity>B</entity>
    </calculus>
  </module>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="binary" name="cardir">
    <operation type="constraint-reasoning">refine </operation>
      <entity>A</entity>
      <relation>N</relation>
      <entity>B</entity>
      <entity>A</entity>
      <relation>S</relation>
      <entity>B</entity>
    </calculus>
  </module>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="binary" name="cardir">
    <operation type="constraint-reasoning">extend</operation>
    <entity>A</entity>
    <relation>N</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>S</relation>
    <entity>B</entity>
  </calculus>
</module>

```

4.4.4 Algebraic-reasoning Module

An algebraic reasoning module contains two operations like consistency and qualify. Each operation needs specific type of syntax that contains *module-name*, *calculus-name*, *operation*, *entities* and *relations*.

4.4.4.1 Consistency Operation

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="a-reasoning">
  <calculus type="binary" name="dra-24">
    <operation type="a-reasoning">consistency</operation>
    <entity>A</entity>
    <relation>rllr</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>ells</relation>
    <entity>C</entity>
  </calculus>
</module>

```

Consistency operation takes same constraint-network's specific parameters as constraint reasoning do.

4.4.4.2 Qualify Operation

Qualify operation in algebraic-reasoning module takes input parameters as quantitative description of the scene and generate quantitative description. The syntax of query required tags as a sequence of commands like *module_name*, *calculus_name*, *operation*, *controlMode*, *network parameter*. Network parameter contains set of entities with defined identifier and entity-type use for reasoning like

entity_Id and *entity_type*.

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="a-reasoning">
  <calculus type="binary" name="dra-24">
    <operation type="a-reasoning">qualify</operation>
    <controlMode>all</controlMode>
    <entity type="-3 0 8 0">A</entity>
    <entity type="8 0 4 8">B</entity>
  </calculus>
</module>
```

Similarly for ternary calculi I design XML query structure. XML structure for queries using ternary calculi and constraint-reasoning module in SparQ is given below. Each module specific operations in constraint-reasoning module take specific set of command as argument.

For ternary calculi, I considered *FlipFlop Calculus (FFC)* proposed by Ligozat (1993) to describe a position of Point “C” with respect to two points “A” (the origin) and “B” (the relatum) in the terms of basic nine relations. The defined base relation are left (l), right (r), front (f), back (b), inside (i), start (s), end (e), dou and tri.

4.4.5 Constraint-reasoning for Ternary Calculi

4.4.5.1 Algebraic-closure

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="ternary" name="ffc">
    <operation type="constraint-reasoning">algebraic-closure</operation>
    <entity>A</entity>
    <entity>B</entity>
    <relation>l</relation>
    <entity>C</entity>
    <entity>B</entity>
    <entity>C</entity>
    <relation>r</relation>
    <entity>D</entity>
  </calculus>
</module>
```

4.4.5.2 Scenario-consistency

For ternary calculi scenario-consistency operation of constraint-reasoning module takes parameters like *module-name*, *calculus-name* and its type, *operation-name* and its type, *return-type*, *entities* and their relations.

4.4.5.3 Refine and Extend

Both operations are used to identify conjunction and disjunction relations in the

given constraint-networks. It takes similar query structure as I mentioned above in scenario-consistency and algebraic-closure operation for ternary calculi.

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="ternary" name="ffc">
    <operation type="constraint-reasoning">scenario-consistency</operation>
      <entity>A</entity>
      <entity>B</entity>
      <relation>l</relation>
      <entity>C</entity>
      <entity>B</entity>
      <entity>C</entity>
      <relation>r</relation>
      <entity>D</entity>
    </calculus>
  </module>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="ternary" name="ffc">
    <operation type="constraint-reasoning">refine </operation>
      <entity>A</entity>
      <entity>B</entity>
      <relation>l</relation>
      <relation>r</relation>
      <entity>B</entity>
      <entity>C</entity>
      <relation>l</relation>
      <entity>D</entity>
    </calculus>
  </module>
```

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="ternary" name="ffc">
    <operation type="constraint-reasoning">extend </operation>
      <entity>A</entity>
      <entity>B</entity>
      <relation>l</relation>
      <relation>r</relation>
      <entity>B</entity>
      <entity>C</entity>
      <relation>l</relation>
      <entity>D</entity>
    </calculus>
  </module>

```

4.5 XML Conversion to SparQ Syntax

As we know that SparQ accept input queries in SparQ specific syntax and provides results in particular syntax based on used module for reasoning. Each SparQ modules takes specific syntax as set of parameters like *module-name* and *calculus-name* are common in all modules but network specify parameters are dependent upon module specific queries.

The developed API takes these XML queries as input and converts in to SparQ specific syntax by adding additional parameters like (parentheses, blank-spaces etc.). It forwards generated SparQ specific syntax in a string format over TCP/IP connection for reasoning.

4.5.1 Qualify Query (XML) to SparQ Syntax

Qualify module queries in XML format contains tags like *module-name*, *calculus-name* *entity-type* and *entity-id*. After parsing XML file, i accessed defined tags as tree nodes and their attributes by comparing tag names. We needed to set blank-spaces [“ ”] at the end of each tag value and parentheses[“(“] before starting of entity and ending of entity-type in the given network. By using below given logic, I generated qualify module specific syntax like.

<Module-name> “ ” <calculus-name> “ ” <controlMode> “ ” (<entity> “ ” <entityType>) (<entity> “ ” <entityType>).

General sudo-code to generate above SparQ syntax is given below.

Step1. If (calculus-Type matches ("binary")

Then

1. Access elementByTagName equal to "module" and add [" "]
2. Access elementByTagName equal to "calculus" and add [" "]
3. Access attribute define in elementByTagName equal to "controlMode" and add [" "]
4. Create array List for entities.
5. for-loop (set integer i=0; i<array-length(); i++)
 1. Add "("
 2. Get Entity at position (i) by comparing tag-name "entity"
 3. Get attribute of entity by comparing "type" to access entity-type
 4. add "(" before entity-name , add [" "] , add entity-type, and add ")"

Repeat step (4)

Close if condition

Step 2: Pass all values as string in sparqString.

4.5.2 Constraint-reasoning Query (XML) to SparQ Syntax

Constraint-reasoning module contains two operations algebraic-closure and scenario-consistency is used to identify scenario-consistency of given constraint-networks. Both operation takes same structure of constraint-networks like

<Module-name> " " <calculus-name> " " <operation> " " <return-name> " " (<entity-name> " " <relation> " " <entity-name> " ") (<entity> " " <relation> " " <entity-name>)

The major difference between algebraic-closure and scenario-consistency operation is return-types, scenario-consistency take extra command as return-type. Given XML query is converted in constraint-networks by add SparQ specific parameters with the sequence of entities and relations like (A N B) (B E C)

In XML Query we are just providing entities (A, B, C) and relations (N, E). To generate above mentioned SparQ syntax for constraint-networks, I added parentheses and spaces with the help of Java coding. Sudo-code for XML queries conversion is given below

```

Step1. If calculus-Type matches ("binary")
Then
    Step1. If module-name matches "constraint-reasoning"
    Then
        Step1. Access elementByTagName equal to "calculus" and add [" "]
        Step2. If operation-name matches "algebraic-closure"
        Then
            1. Set string =sparqString;
            2. sparqString=module-name; add [" "].+Caulus-name; add [" "]+
               operation-name, add[" "]+ add Nodes
            3. Close if
        Step3. If calculus-name matches "calculus"
            1. Set calculus Nodes as array List;
            2. Create nodeList of calculus Nodes
        Step5. For-loop (int i=0; i<calculusNode.length (); i++)
            1. get childNode at position (i);
            2. get nodeValue and save as value in constrant-networkString
            3. If (node-name matches "entity") and
               If (node-name is first entity)
               Then
                   1. add ["("]
                   2. add value
                   3. add character "R"
            5. Close if
            6. If (node-name is second entity)
            Then
                1. add nodeValue
                2. add [")"]
                3. replace "R" with relations
            7. Close if
            8. Else if (node-name matches "relation"
            Then
                1. add value of relation
                2. add [" "];
            9. Close else if
        Step6. Repeat Step-5
    Step2. Close if
    Step3. Return constraintNetworkString

```

Same logic is used to convert and generate constraint-networks from the given XML query in both operation of constraint-reasoning module. Queries using refine and extend operation in constraint-reasoning takes same syntax like ((A (N S) B))((B (N) C)). The sequence and number relations defined as constraint-networks are different from other constraint-reasoning operations. In refine and extend operation there is possibility of more then one relation between given entities and each network is covered with double closing and opening parentheses [“(”], [“)”] and blank-spaces [“ ”] between entities and relations. The SparQ syntax for refine and extend is given below.

```

<module-name> “ ” <calculus-name> “ ” < operation> “(” <entity> “ ”
<relation> “ ” <relation> “ ”<entity> “)” “(” <entity> “ ”<relation> “ ” <entity>

```

“)”

We can develop such a syntax form given XML query by converting it with the help of Java coding. Sudo code for conversion is given below.

```
Step1. If calculus-type matches ( "binary" )
Then
    Step1. Set String sparqString
    Step2. Get module-name [ " " ] get calculus-name[ " " ] + get operation-
            name + [ " " ] + refineExtendString
    Step3. Access elementByTagName equal to "calculus"
    Step4. Create Array NodeList
            1. Pass all calculus_ChildNodes in array
            2. Set String refineExtendString;
    Step5. For-loop (int i=0; i<calculusNode.length (); i++)
            1. get childNode at position (i);
            2. get nodeValue and save as value in refineExtendString
            3. If (node-name matches "entity") and
            4. If (node-name is first entity)
            Then
                    4. add [ "(" ]
                    5. add value
                    6. add character "R"

            5. Close if
            6. If (node-name is second entity)
            Then
                    4. add nodeValue
                    5. add [ ")" ]
                    6. replace "R" with relations
            7. Close if
            8. Else if (node-name matches "relation"
            Then
                    3. add value of relation
                    4. add [ " " ];
            9. Close else if
    Step6. Repeat Step-5
Step2. Close if
Step3. Return refineExtendString
```

The initial syntax like *module-name*, *calculus-name* and *operation-name* are same as other constraint-reasoning operations takes.

4.5.3 Compute-relation Query (XML) to SparQ Syntax

Like other SparQ modules, queries using compute-relation takes following sequence of parameters as input.

<module-name> " " <calculus-name> " " <operations-name> " " <relations>

Operation-name may contains nested operation-names with defined relations, relations are depends upon operations type, as binary-operation takes two relations as input parameters and ternary-operation takes three relations as input parameters.

<Operation-name> "(" <operation-name> " " <relation> " " <relations> ")"

Sudo-code to generate SparQ specific syntax for compute-relation module specific query is given below.

```

Step1. If calculus-type matches ("binary")
Then
    Step1. Get module-name [ " " ] get calculus-name[ " " ] +
    Step2. Create nodeList of operation by access with tag-name "operation"
        Set string operations
    Step3. For-loop (int i=0; i< nodeList.getLength (); i++)
        1. operation= add [ " " ]
        2. add operation-name value at position(i) is defined string and
        3. add [ " " ]
        4. add character R
    Step4. Close for-loop
    Step5. Save operations
    Step6. Get relation-name by TagName
    Step7. Replace R with relation-name
Step2. Close if
Step3. Return sparqString

```

4.5.4 Algebraic-reasoning Query (XML) to SparQ Syntax

In SparQ algebraic-reasoning module is represented as a-reasoning. In a-reasoning consistency operation used to identify network consistency and qualify operation is used to get qualitative description from given quantitative description of the scene.

In a-reasoning, consistency operation takes same syntax as constraint-reasoning do. Therefore XML conversion in a-reasoning use same logic a mentioned in constraint-reasoning module by adding parentheses [(), []] and blank-spaces [" "] after each command and between defined entities and relations .

<Module-name> " " <calculus-name> " " <operation> " " <return-name> " " (<entity-name> " " <relation> " " <entity-name> " ") (<entity> " " <relation> " " <entity-name>)

Similarly queries using a-reasoning module and qualify operation takes same syntax as qualify module except operation-name. Qualify operation takes extra command as operation-name in a-reasoning specific queries.

<Module-name> " " <calculus-name> " " <operation> " " <controlMode> " " (<entity> " " <entityType>) (<entity> " " <entityType>).

To generate SparQ specific syntax for ternary-calculi, I set a condition to validate either query contains calculus-type "binary" or "ternary". Ternary-calculi are used to reason on entities with respect to two given other entities. General syntax for queries using ternary-calculi is given below.

<Module-name> " " <calculus-name> " " <operation> " " <return-name> " " (<entity-name> " " <entity-name> " " (<relation>) " " <entity-name> " ") (<entity> " " <entity-name> " " (<relation>) " " <entity-name>)

The major differences in queries using binary and ternary calculi are extra entity-names with in given constraint-network. As we know that ternary-calculi are used to reason on entity (A) with respect to entity (B) and entity (C) Constraint-networks for

algebraic-closure and scenario-consistency operations takes the following structure as constraint-network in ternary-calculi ((A B North (n) C) (B C South(s) D). Sudo-code for generating constraint-network is given below.

```

Step1. If calculus-type matches ("ternary")
Then
  Step1. If module-name matches "constraint-reasoning"
  Then
    Step1. Access elementByTagName equal to "calculus" and add [" "]
    Step2. If operation-name matches "algebraic-closure"
    Then
      Step1. Set String sparqString
      Step2. Get module-name [" "] get calculus-name[" "] + get operation-
      name + [" "] + constraintNetworkString
      Step3. Access elementByTagName equal to "calculus"
      Step4. Create Array NodeList
        3. Pass all calculus_ChildNodes in array
        4. Set String refineExtendString;
      Step5. For-loop (int i=0; i<calculusNode.length (); i++)
        3. get childNode at position (i);
        4. get nodeValue and save as value in refineExtendString
        3. If (node-name matches "entity") and
        4. If (node-name is first entity)
        Then
          1. add ["("]
          2. add value
          3. ent++
        5. Close if
        6. If (node-name is second entity)
        Then
          1. add nodeValue
          2. add character "R"
          3. ent++
        7. If (node-name is third entity)
        Then
          1. add [")" ]
          2. replace "R" with relations
        7. Close if
        8. Else if (node-name matches "relation")
        Then
          5. add value of relation
          6. add [" "];
        9. Close else if
      Step6. Repeat Step-5
    Step2. Close if
  Step3. Return ConstraintNetworkString

```

Similarly for refine and extend operation using ternary-calculi takes same constraint-networks defined above. It takes extra parenthesis at the starting of constraint-network entity ["(("] and at the ending of constraint-network [")"). General structure for refine and extend operation using ternary calculi is given below.

<module-name> " " <calculus-name> " " <operation> " "((<entity-name> "

”<entity-name> “ ” (<relation>)“ ”<entity-name> “ ”)) “ ” ((<entity> “ ”<entity-name> “ ” (<relation> “ ”<entity-name>)).

4.6 XML Parsing

In computing, a parser is a program (or a piece of code that you can reference inside your own programs) which is used to analyze file to identify the components. All application that read input has a parser of some kind to identify the meaning of the information. XML applications are just same, they contains a parser which reads XML file and identifies the function of each the pieces of the document, and it makes that information available in memory to the rest of the program. There are two most popular APIs known by SAX (Simple API for XML) and DOM (Document Object Model) for processing XML document in JAVA.

Generally in DOM parser all elements and attributes in structure can be referenced by walking through the DOM tree. Their contents can be modified or delete and new elements can be created from subsequent insertion into the DOM tree.

In this API, I used DOM parser that contains set of methods to parse given XML file and generate Document Tree. The DOM parser is useful, when you need to know a lot about the structure of the document, to sort certain elements and when you need information in the document more then once.

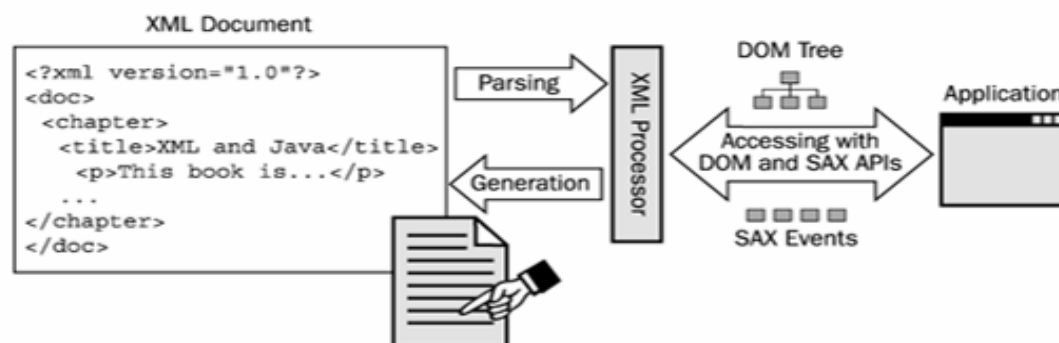


Figure 20: XML document and there integration with JAVA application H. Maruyama, 2002

In API parse () method takes generated XML file as argument and creates Document tree. All defined tags contains set of values as attributes, these attributes are accessible through method called *getAttributeByTagName()* and structured it into the SparQ specific syntax as string. SparQ syntax takes extra characters like quotations, parenthesis etc.

4.7 SparQ Result Analysis

SparQ results are analyzed to identify the possible output patterns as result for given input queries .I used all possible modules specific queries to find out commonalities between the results given by reasoner and type of errors that generates. I considered both binary and ternary calculi during my analysis. The purpose of result analysis is to design common standard XML data structure for given results. Based on result analysis, I generalized possible SparQ outputs in the following categories.

1. Simple-text

2. Simple-text and constraint-network
3. constraint-network
4. Simple relations
5. Errors

<i>Module/resultTypes</i>	<i>Qualify</i>	<i>Compute-relation</i>	<i>Constraint-reasoning</i>	<i>Algebraic-reasoning</i>
<i>Simple-text</i>	No	No	Yes	Yes
<i>Simple-text and constraint-network</i>	No	No	Yes	No
<i>constraint-network</i>	Yes	No	Yes	Yes
<i>Simple-relations</i>	No	Yes	No	No
<i>Syntax errors</i>	Yes	Yes	Yes	Yes

Table 4: Classification of modules specific results for both binary and ternary calculi

4.7.1 Simple Text

SparQ generates simple text result for some queries using constraint-reasoning module specific operation like (algebraic-closure, scenario-consistency) and Algebraic-reasoning module specific operation like (consistency). The given simple-text result has four categories. These types of results are usually occurred in constraint-reasoning module and remaining other results is given by consistency check operation in algebraic-reasoning module.

1. Not Consistent.
2. NOT SATISFIABLE
3. CANNOT DECIDE
4. SATISFIABLE

During conversion with the help of API the given Simple text result is passed in tag-name <comments></comments> that is already defined in XML structure for SparQ outputs.

```
<result type = "constraint-reasoning">
  <operation name= "algebraic-closure" type= "constraint-reasoning">
    <comments> Not Consistent</comments>
  </operation>
</result>
```

4.7.2 Simple-text and Constraint-network

Queries using Constraint-reasoning module, algebraic-closure and scenario-consistency operation with return parameters (all) generates results as composition of simple-text and constraint-networks. During conversion these results are split in to two set of arrays with the help of numerical value [0-9] and [.]. The array that contains simple text, is forwarded as comments in comments-tag. The second array that contains constraint-network is further processed and split it into substring based on common characters like] ([. The given substring is passed as attributes in defined tags like entities and relations with the help of API.

Similarly algebraic-closure and scenario-consistency with return parameter (first) generates results either in simple-text format as syntax error like (NOT

CONSISTENT) or returns constraint-network like ((A (N) B)(B (E) C)). The constraint-network represents specific pattern like (object1, relation, object2) (object2, relation, object3) based on the analysis, the given constraint-network is further processed to set these values as attributes of relations and entities with in defined entities tags and relation-tags.

```
<result type = "constraint-reasoning">
  <operation name= "algebraic-closure" type= "constraint-reasoning">
    <comments> Modified Network</comments>
    <entity>A</entity>
    <relation>rllr</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>ells</relation>
    <entity>C</entity>
  </operation>
</result>
```

```
<result type = "constraint-reasoning">
  <operation name = "scenario-consistency" type= "constraint-reasoning">
    <comments> 1 Scenario found: no further Scenarios exist</comments>
    <entity>A</entity>
    <relation>N</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>ne</relation>
    <entity>C</entity>
    <entity>B</entity>
    <relation>e</relation>
    <entity>C</entity>
  </operation>
</result>
```

4.7.3 Constraint-network

Most of queries using defined modules and their specific operations like qualify, algebraic-reasoning, constraint-reasoning (algebraic-closure, refine and extend operations) gives results in the form of constraint-network like ((A (N) B) (B (E) C)). The given networks is split based on common characters like ["(") ("] in to possible sub-networks with the help of API and generate XML structure by passing these element as attributes in defined tags like entity and relation. The operation-tag contains operation-name and its type. Qualify module doesn't contains any specific operations. Therefore in resultant XML for qualify module does not contains operation-name and its type represents module name used for reasoning.


```

<result type = "a-reasoning">
  <operation name="qualify" type= "a-reasoning">
    <entity>A</entity>
    <relation>rllr</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>ells</relation>
    <entity>C</entity>
  </operation>
</result>

```

4.7.4 Simple-relations

Queries using compute-relation module gives simple-relation as result. In compute-relation both binary and ternary operations takes relations as parameters and inferring possible relations between given entities, based on the defined operation in query. e.g.

(Composition (DC, EC and TPP) generates result as set of relations

(dc, ec, ntp, po, tpp)

These relations are converted in to XML by passing relation as attributes in defined relation-tag in XML.

```

<result type = "compute-relation">
  <operation name= "composition", type = "binary" >
    <relation0>dc</relation0>
    <relation1>ec</relation1>
    <relation2>ntpp</relation2>
    <relation3>po</relation3>
    <relation3>tpp</relation3>
  </operation>
</result>

```

4.7.5 Syntax-errors

Error generated by SparQ as result on given reasoning query is extracted with the help of API. In SparQ error is specified as string "An error occurred" Based on given string API converts error in to XML format, by passing it as value in the defined comments-tag

```

<result type = "constraint-reasoning">
  <operation type = "algebraic-closure">
    <comments> An error occurred: Error in module specification,.
    Object missing..... ,
  </comments>
  </operation>
</result>

```

4.8 SparQ Result Conversion into XML

The response received from SparQ as query result is starting with tag-name *sparq*>. Using sub-string method, I extracted required result and store as string. The final

result is further processed to convert into defined standard XML structure. Most of the SparQ modules and their operations provides module and operation specific result syntax. Therefore we need to use module specific conditions to convert given results in defined XML format.

4.8.1 Syntax-errors

All syntax errors related information is extracted based on condition that matches “An error occurred:” usually, all error messages in the SparQ are forwarded with above mentioned string e.g.

sparq> An error occurred: Error in module specification,. Objects are missing.....

The generated error message is extracted and passed between tag-name <comments> </comment> as value.

*Step1. Set string resp;
 Step2. Extract and trim final result, and store as string in resp
 Step3. If (resp matches “An error occurred”
 Then
 1. extract error and save as result
 2. close if
 Step4. pass result between tag-name <comments></comments>*

4.8.2 Simple-relations

Reasoning Queries using Compute-relation takes binary/ternary operations and relations as parameters. The result generated by SparQ contains simple set of possible relations like

Sparq> (ne, n s se)

I set a condition to match module-name with given string like “compute-relation”. Based on this condition, I extracted relations with the help of java sub-string method and remove all parentheses “(” “)” and forwarded these relations in string array. I accessed all relations one by one with the help of loop and passed it between tag-name <relation></relation>. If there is no relation as result, API will forward blank space with in parentheses like [()].

*Step1. Set string resp;
 Step2. Extract and trim final result, and store as string in resp
 Step3. If (resp contain “sparq> and “(
 If (match module-name with (“compute-relation”))
 Then
 1. remove [() and trim result
 2. pass result in array called relations[]
 3. for-look(int i=0; i<relation. length (); i++)
 1. get relation at position (i)
 2. Pass relation between tag-name <relation></relation>
 Close loop
 Close if
 Close if*

4.8.3 Simple-text and Constraint-network

For both binary and ternary calculi queries using Constraint-reasoning operations like algebraic-closure and scenario-consistency with return-type (all) generate results as combination of text and constraint-networks like

- i. *sparq>Modified Network. ((B (e) C)(A (ne) C)(A (n) B)) or*
- ii. *sparq> ((B (e) C)(A (ne) C)(A (n) B))1 scenario found, no further scenarios exist.*

Such a type of result is extracted based on conditions, that if given response contains both tag-name [**sparq>**] and string parentheses [()] then extract response from reasoner and store as string named “result”. The given result is further split using java regular expression in to two string arrays based on common characters in module specific result. In algebraic-closure operation, i split result based on character [.] and store in two arrays. Similarly in scenario-consistency, i used [0-9] expression to split result in to simple text and constraint-networks for further process like conversion in to XML structure. Sudo-code to convert engine result in XML format for both binary and ternary calculi is given below.

```
Step1. Set string resp;
Step2. Extract and trim final result, and store as string in resp
Step3. If (resp contain “sparq> and “(“
      If module-name matches (“scenario-consistency”)
      If (returnType matches” all)
      Then
        1. Set array split-network
        2. Split resp based on [0-9] and save in split-network at position (0)
        3. Pass text at split-network position (1) to <comments>
        4. Split sub-network based on “[)][(]”
        5. Save as array list
      Close if
      Close if
      Close if
Step4. Remove all parentheses [()][)] with blank-space [“ ”]
Step5. if (calculuType matches (“binary”)
Then
  1. Get relation using substring (1, result. Length ()-1 and pass relation in tag-
    name <relation>.
  2. Gets first entity using substring (0, 1) and pass entity in tag-name <entity>
  3. Get second entity using substring (result. Length ()-1, result. Length ()) and
    pass it in <entity>.
Step6. else if(calculusType matches “ternary”)
Then
  1. Get relation using substring (3, result. Length ()-1 and pass relation in tag-
    name <relation>.
  2. Gets first entity using substring (0, 2) and pass entity in tag-name <entity>
  3. Gets first entity using substring (2,3) and pass entity in tag-name <entity>
  4. Get second entity using substring (result. Length ()-1, result. Length ()) and
    pass it in <entity>.
Step 7. pass all tags between tag-name <result></result>
Step8. pass XML result in plug-in text-Area
```

4.8.4 Simple Text

Most of the cases using constraint-reasoning module, if the given constraint-network is inconsistent, queries using constraint-reasoning modules and their specific operations like a-reasoning module (consistency), constraint-reasoning module (algebraic-reasoning, scenario-consistency) generates results as simple text like.

sparq> Not Consistent.

These results can be “*NOT Consistent*”, “*Cannot decide*”, “*not satisfiable*”, “*satisfiable*”. Such a result are simple extracted based on starting tag *sparq>* and passed it as a string in defined tag-name *<comments>*. Sudo code to extract and convert simple text result is give below.

```
Step1. Set string resp;
Step2. Extract and trim final result, and store as string in resp
Step3. If (resp contain “sparq>”
      If (resp contains “NOT CONSISTENT.” or “NOT SATISFIABLE” or
          “SATISFIABLE” or “CANNOT DECIDE” )
      Then
          1. extract sub-string
          2. Save string as result
      Close loop
Step4. Pass result between tag-name <comments></comments>
```

4.8.5 Constraint-network

Queries using module specific operations like scenario-consistency with returnType (first), refine extend and qualify are generating result as a constraint-networks (object1 relation object2) (object2 relation object3) like

Sparq> ((A (N) B)(B (E) C)).

Such a result contains sequence entities and relations as constraint between the given entities. There is a possibility of empty relation between two given entities or more then one relation between them. I extracted SparQ response based on conditions like, if sparq response contains [*sparq>*] and starting parentheses “(”, then extract the result and split it into sub-networks using java regular express `[(.*)]`. General logic to convert sub-networks in XML tags are given below.

```

Step1. Set string resp;
Step2. Extract and trim final result, and store as string in resp
Step3. If (resp contain "sparq> and "("
    If module-name matches ("constraint-reasoning")
    If operation matches ("refine")
    Then
        1. Set array split-network
        2. Split result based on "[)]([" and save in list
        Close if
    Close if
Step5. For-loop (int i=0; i< list. length ();i++
    1. Replace All "(" with [" "]
    2. Close Loop
Step6. Get relation using substring (1, result. Length ()-1 and pass relation in tag-
name <relation>.
Step7. Gets first entity using substring (0, 1) and pass entity in tag-name <entity>
Step8. Gets second entity using substring (result. Length ()-1, result. Length ()) and
pass it in <entity>.
Step9. Pass all tags between tag-name <result></result>

```

5. API Implementation

5.1 API Class Diagram

Class diagram in the Unified Modeling Language (UML) is type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, behavior and the relationships between classes. A relationship is a general terms covering the specific type of logical connection between classes and instance of the classes, which can be inheritance, aggregation/association interface etc.

5.1.1 Plug-In Class Diagram

There are two main packages in openJUMP source code. *Com.vividsolutions.jump* and *org.openjump.core*, contains the original sources. All classes from these packages can be fixed or improved. To create OpenJUMP plug-in, i used four classes.

5.1.1.1 Extension Class

An extension is a collection of classes and supporting resources that provides additional functionality to JUMP application. Extensions are packaged as JAR file. It is used to add plug-ins and cursor tools to the workbench.

5.1.1.2 MyExtension Class

MyExtension class extends Extension class. It contains two main methods configuration () and initialize(). Configuration method configures an extension that used to add plug-in. in the JUMP workbench. Configuration method call on each Extension class it finds.

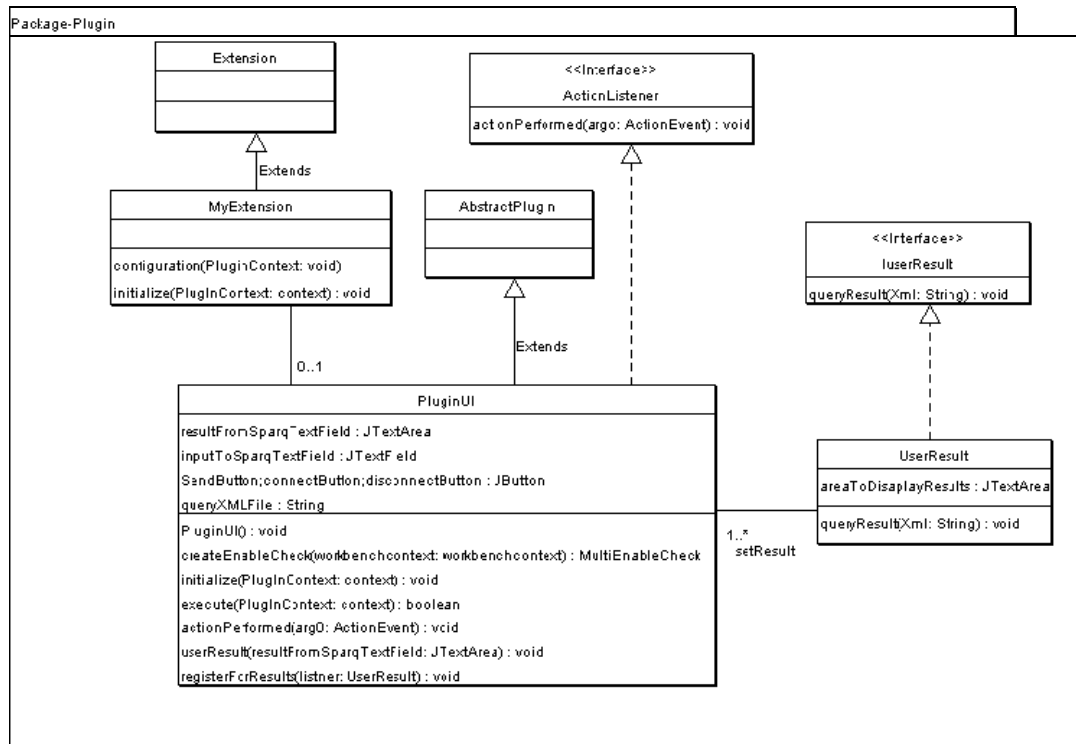


Figure 21: Plug-in UML Class Diagram

5.1.1.3 PlugInUI Class

Class extends *AbstractPlugIn* class. It has three methods including *initnialize()*, *excute()* and *createEnableCheck()*. PlugInUI Class override these method defined in *AbstractPlugIn* class. PlugInUI class also implements *ActionListner* interface, it define methods to add *ActionListner* on designed buttons like *sendButton.addActionListner(this)*;

```

public void initialize(PlugInContext context) throws Exception{
    FeatureInstaller fi=new
    FeatureInstaller((context.getWorkbenchContext()));
    If.addmainMenuItem(this,new
    string[] {"View"},this.getName(),false,null,creatEnableCkech(context.g
    etWorkbenchContext()));
}

```

Initialize() method is used to initialize plug-in that loads developed plug-in as menu item with given name.

```

public boolean excute(PlugInContext context) throws Exception{
    textArea =new JTextField (20);
    outputArea= new JTextArea(50);
    context.getworkbenchFram().getOutputFrame().setLayout(new
    GridLayout(4,4));
    context.getworkbenchFram().getOutputFrame().addText("Sparq
    Reasoner");
    context.getworkbenchFram().getOutputFrame().add(textArea);}

```

actionPerformed() method contains methods like *startConnection()* and *sendXMLQueryToReasoner()* based on the action performed by Plug-in user. Method provides interface between API and plug-in program.

```
public void actionPerformed(ActionEvent arg0) {
    JButton clickedButton = (JButton)arg0.getSource();
    if(clickedButton.getName().matches("send")){
        queryXMLFile = inputToSparqTextField.getText();
        System.out.println("input xml path : " + queryXMLFile);
        if(!queryXMLFile.isEmpty()){
            api.sendXMLQueryToReasoner(queryXMLFile);
        }else {
            resultFromSparqTextField.setText("Please input a proper
XML file path ");
        }else if(clickedButton.getName().matches("connect")){
            System.out.println("Connecting to the engine ....");
            api.startConnectionAt("localhost", 4444);
        } else if(clickedButton.getName().matches("disconnect")){
            api.closeTheConnection();
            api = null;
            listner = null; }}

```

excute () method takes argument *plugInContext* type of context and initialize *outputframe()* method to register result received from reasoner with the help of *rgisterForResult()*, set plug-in layout, plug-in title, and other fields related with user interface.

```
public boolean execute(PlugInContext context)throws Exception{
    resultFromSparqTextField = new JTextArea(10, 50);
    inputToSparqTextField = new JTextField(50);
    sendButton = new JButton("Send QUERY");
    connectButton = new JButton("Connect REASONER");
    disconnectButton =new JButton("Disconnect REASONER");
    Dimension size=new Dimension(600,400);
    api =new QuerySender();
    listner = new UserResult(resultFromSparqTextField);
    api.registerForResults(listner);
    return true;}

```

5.1.1.4 userResult Class

userResult class implements *IUserResult* interface. *userResult* class overrides defined method *queryResult(String xml)*. It is used to pass the generated result in textArea defined to display final result in the designed plug-in


```

public class UserResult implements IUserResult{
    private JTextArea areaToDisplayResults = null;
    public UserResult(JTextArea tArea) {
        this.areaToDisplayResults = tArea;}
    public void queryResult(String xml) {
        System.out.println("Result in XML format....." + "\n"+ xml);
        if(this.areaToDisplayResults!=null){
            this.areaToDisplayResults.setText("");
            this.areaToDisplayResults.setText(xml);
        }
    }
}

```

5.1.1.5 AbstractPlugIn Class

It is a built-in class contains set of methods that are used in *PlugInUI* class, provided by *vividsolution*. It is used to generate plug-in name from the class name.

5.1.2 API Class Diagram

API class diagram contains set of classes with two implemented interfaces, used to perform different processes related with the XML processing, validation with SparQ syntax, conversion of XML query in SparQ syntax and establishing connection with reasoner. API contain following classes

1. EngineConnector
2. IQueryGenerator
3. IUserResult
4. QueryGenerator
5. QueryReceiver
6. QuerySender

Here in this thesis, I used OpenJUMP application to demonstrate API functionality by integrate API application with developed plug-in. OpenJUMP Application activates both plug-in application and API at the same time, when extension is loaded for reasoning on spatial data. In future the developed API can access over network or can modify to access over http as web-based API.

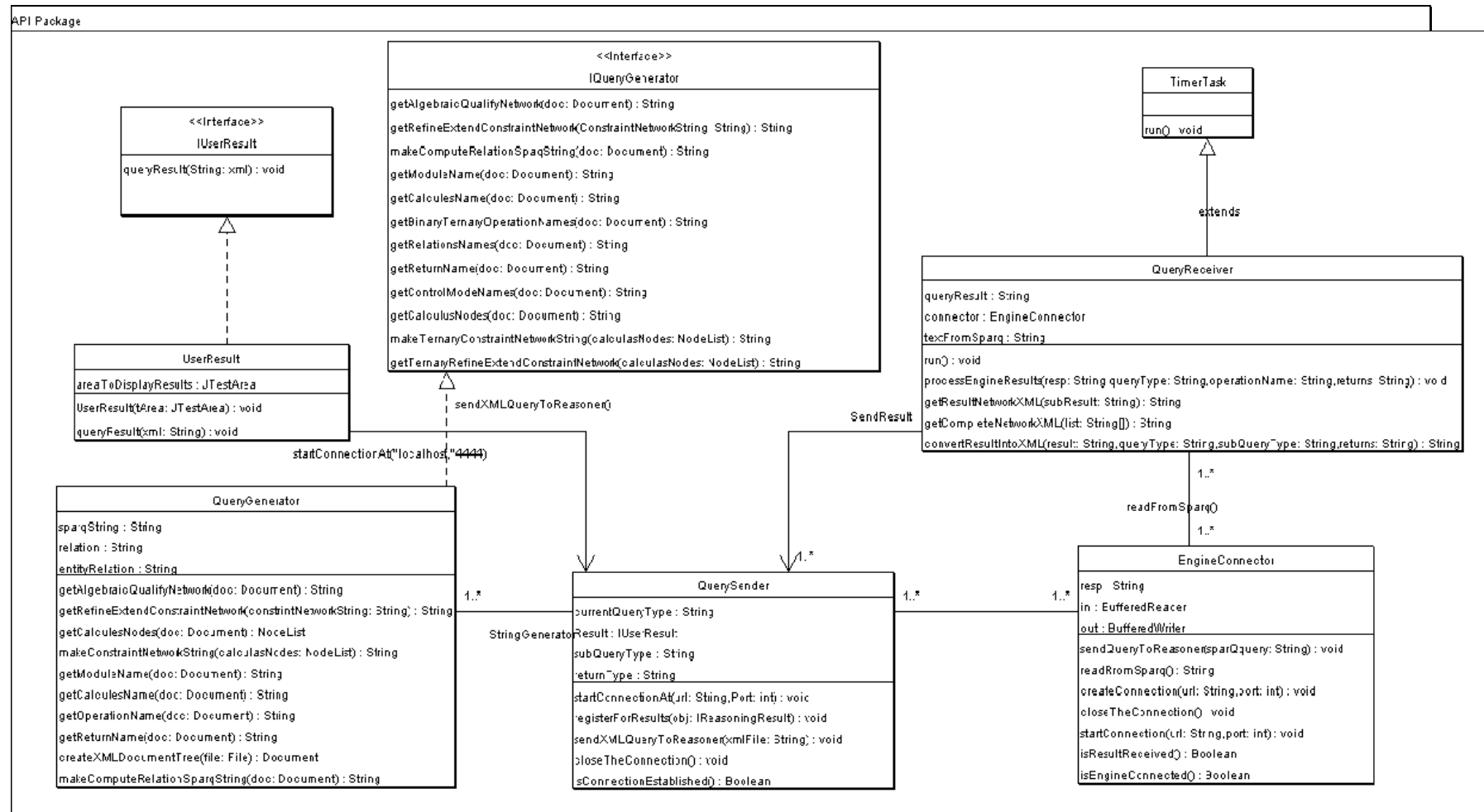


Figure 22: API UML Class Diagram

5.1.2.1 EngineConnector Class

A class *EngineConnector* contains set of methods that provides main functionalities including sending queries to reasoner, receiving results from reasoner, establishing connection with reasoner via TCP/IP and controlling connection with reasoner.

createConnection() method takes two arguments as *hostname* and *port_number* to establish connection with reasoner via TCP/IP. It creates Buffer reader and writer to read and write queries and result over TCP/IP after establishing connection with reasoner.

```
public void createConnection(String url, int port){
    try{
        Socket skt=new Socket(url,port);
        Skt.setKeepAlive(true);
        In=new BufferedReader(new InputStreamReader(skt.getInputStream()));
        Out=new BufferedWriter(new OutputStreamWriter(skt.getOutputStream()));
    } catch(Exception e){
        System.out.println("problem in connection"+e.getMessage());
    }
}
```

isResultReceived() method is used for verification, that result is received from reasoner. Result from reasoner contains Sparq-tag like *sparq*>. If the result contains tag-name, method will return true value otherwise it will return false.

```
public boolean isResultReceived(){
    return isSparqTagReceived;
}
```

closeTheConnection() method is used to control connection with reasoner as well as to control the buffers reader and writer.

```
public void closeTheConnection(){
    try{
        out.close();
        in.close();
    }
    }catch(Exception e){
    }
}
```

sendLastReceivedQueryFromUser() method is used to listen socket and forward query to the reasoner using established connection. *Out.flush()* method is used to forward the reasoning query.

```

public void sendLastReceivedQueryFromUser(String sparQuery){
try{
    Out.write(sparQuery);
    Out.newLine();
    Out.flush();
} catch(Exception e){
    e.printStackTrace();
}}

```

isEngineConnected() method is used to activate *BufferRead* that listen input stream and return Boolean value as true, if *BufferRead* is not ready it through IO exception as error.

```

public Boolean isEngineConnected(){
try{
    return in.ready();
} catch(IOException err){
    err.printStackTrace();
    return false;
}
}

```

5.1.2.2 IQueryGenerator Interface

IQueryGenerator is an interface implemented by *QueryGenerator* class. It defines all possible methods required by specific reasoner. These methods are implemented in *QueryGenerator* class to access attributes and values defined with in tags of given XML file.

```

public interface IReasoningQueryGenerator {
    public String getAlgebraicQualifyNetwork(Document doc);
    public String getRefineExtendConstraintNetwork(String conNetworkStr);
    public NodeList getCalculusNode(Document doc);
    public String getModuleName(Document doc);
    public String getCalculusName(Document doc);
    public String getBinaryTernaryOperationNames(Document doc);
    public String getRelationNames(Document doc);
    public String getOperationName(Document doc);
    public String getControlModeNames(Document doc);
    public String getReturnNames(Document doc);
    public String makeTernaryConstraintNetworkString(NodeList
calculusNodes);
    public String getTernaryRefineExtendConstraintNetwork(NodeList
calculusNodes);
}

```

5.1.2.3 QueryGenerator Class

QueryGenerator class implements all defined method in *IQueryGenerator* (interface). It contains set of methods that are used to access nodes and their values defined in Document tree like *module-name*, *calculus-name* and *constraint-networks*. Each module in SparQ has its own syntax to define network specific parameters. *getModuleName()*, *getCalculusName()*, *getBinaryTernaryOperationNames()*, *getOperationNames()*, *getReturnName()* and *getControlModeNames()*, these methods are used to access elements attributes and their values, Method *getElementByTagName()* returns element attribute using defined tag-name in XML file and returns its attribute as string.

```
Public String getModuleName(Document doc){
    Element
    module=(Element)doc.getElementByTagName("module").item(0);
    Return (module.getAttribute("name"));
}
```

```
Public String getOperationName(Document doc){
    Element module=(Element)doc.getElementByTagName("operation")
    .item(0).getTextContent;
    Return (module.getAttribute("name"));
}
```

createXMLDocumentTree() method takes defined XML file as argument and creates document in tree structure using *parse()* method. Tree structure developed during parsing and used to access elements defined as tags and their attributes.

```
Private Document createXMLDocumentTree(File file){
    Document doc=null;
    try{Doc=DocumentBuilderFactory.newInstance().newDocumentBuilder().p
    arse(file);
    }catch(SaxException e){
        e.printStackTrace();
    }catch(SaxException e){
        e.printStackTrace();
    } catch(SaxException e){
        e.printStackTrace();
    }return doc;}
}}}}
```

makeConstraintNetworkString() method is used to create constraint-network string for binary calculi using parameters like (*entity-name*, *relations*, *entity-name*). In constraint-reasoning module operations named (algebraic-closure) and (Scenario-consistency) takes arguments as constraint-networks for reasoning in the following structure“((A (N S) B)(B (S) C))”. *makeConstraintNetworkString()* creates string in above mentioned syntax for binary calculi and returns their value as set of string.

```

public String makeConstraintNetworkString(NodeList calculasNodes){
    String sparqString = "" ;sparqString += " " + "(";
    int ent = 1;String relation = "";
    String entityRelation = "";
    for(int i=0; i < calculasNodes.getLength(); i++){
        Node node = calculasNodes.item(i);
        String value =node.getTextContent();
        if( node.getNodeName().matches("entity")){
            if(ent==1){
                entityRelation += "(" + value + " (R) ";
                ent++;
            }else if(ent==2){
                entityRelation += value + ")" ;
                sparqString += entityRelation.replace("R", relation);
                ent = 1;relation = "";entityRelation = "";
            }else if(node.getNodeName().matches("relation")){
                relation += value + " " ;
            }
        }
    }return sparqString + ")" ; }
}

```

makeComputeRelationSparqString() method takes document as argument and creates string that contains *module-name*, *calculus name*, *operation* and *relations* . In compute-relation module SparQ accept string in the following sequence.

“(Operation (relation, relation))”

makeComputeRelationSparqString() method creates above mentioned syntax by accessing operations and relations defined in XML tags and returns values as string.

```

public String makeComputeRelationSparqString(Document doc){
    String sparQquery = getModuleName(doc) + " " +
        getCalculusName(doc) + " " +
        getBinaryTernaryOperationsNames(doc).replace("R",
        getRelationsNames(doc));
    System.out.println("[a comput-relation query: ]" + sparQquery);
    return sparQquery;
}
}

```

Similarly *makeTernaryConstraintNetworkString()* method is used to create constraint-network for ternary calculi in ternary calculi constraint-network contains three entities with given constraint like ((A B N C))((B C S D)).

```

public String makeTernaryConstraintNetworkString(NodeList calculasNodes){
    String sparqString = "";
    //char quot = "";
    try{
        sparqString += "(";
        int ent = 1;
        String relation = "";
        String entityRelation = "";
        for(int i=0; i < calculasNodes.getLength(); i++){
            Node node = calculasNodes.item(i);
            String value = node.getTextContent();
            if( node.getNodeName().matches("entity")){
                if(ent==1){
                    entityRelation += "("+ value ;
                    ent++;
                }else if(ent==2){
                    entityRelation += " "+value + "(R)";
                    ent++;
                }else if (ent==3){
                    entityRelation += " "+value + ")" ;
                    sparqString +=
                        entityRelation.replace("R", relation);
                    ent = 1;
                    relation = "";
                    entityRelation = "";
                }
            }else if(node.getNodeName().matches("relation")){
                relation +=value;
            }
        }
    }catch(Exception e){
        System.out.println("[problem to get constraint-network String...]
"+e.getMessage());
    }
    return sparqString + ")";
}

```

`getRelationsNames()` method creates list of all relations used in query. It is using document as argument in `getElementByTagName()`. Value defined in tag name "relation" is retrieve through `getTextContent()` and return as string relations.

```

public String getRelationsNames(Document doc){
    String relations = "";
    NodeList operationsInQuery =
    doc.getElementsByTagName("relation");
    for(int i=0; i <= operationsInQuery.getLength(); i++){
        String relation =
    operationsInQuery.item(i).getTextContent();
        relations += " " + relation ;
    }return relations;
}

```

getAlgebraicQualifyNetwork() method is used to create reasoner specific syntax used in algebraic-reasoning using qualify operation, it takes entity name and quantitative description as entity type and generates possible relations between entities of given network.

```

public String getAlgebiacQualifyNetwork(Document doc){
    NodeList entity = doc.getElementsByTagName("entity");
    String networkString = "(";
    for(int i = 0 ; i <= entity.getLength() ; i++){
        Element ele = (Element)entity.item(i);
        String entityName = ele.getTextContent() ;
        String entityType = ele.getAttribute("type");
        networkString += "(" + entityName + " " + entityType + ")";
    }
    return networkString + ")"; }

```

5.1.2.4 QueryReceiver Class

QueyReceiver class extends *TimerTask* used to schedule receiver that listen input steam up to 500 mile second. It contains method *run ()* to read incoming string as result. While-loop is used to keep program listen input stream until received all result strings, after receiving all data from input stream program, it store it in as string named result and terminate while-loop to process result. In next step *run()* call *processEngineResult()* that process generated result. Most of queries using SparQ forward results within tag-name<**sparq**> and send result with ending-tag like empty space with length (0) but some of the modules like algebraic-reasoning queries, SparQ generates result with out any end flag or with out any result ending information. Based on analysis, i set program that extract results from given tag-name <**Sparq**>, or by comparing module-name using ending flag information. Incase of error in input stream program through IOException as error.

SparQ provides different format of results. Based on analysis, I generalized all possible results, like query result base on compute-relation module contains set of relations within parentheses and “**sparq**” tag. To convert these relations in XML structure, I set a program that replace all parentheses with empty space and extract substring based on tag “**sparq**”. Simple-text as result contains string like *Not Consistent, Satisfiable, not satisfiable and cannot decide*, such a result are extracted using *contains method* that compare results with these define strings and forward

results in tag-name <comments>.

```

public void processEngineResults(String resp, String queryType, String
operationName, String returns,String binaryTernaryOperationType,String
binaryTernaryOperationName,String calculusType){
    String moduleOperationStart="\t"<operation" + " "+ "name =" +
    ""+ ""+operationName+ ""+ " type =" + "
    "+ ""+queryType+ ""+ ">"+ "\n";
    String moduleOperationEnd="\t"</operation>";
    try{
        if(resp.contains("sparq>")) {
            if(resp.contains("An error occured:")){
                int firstBraceIndex = resp.indexOf("sparq>") + 6;
                queryResult = resp.substring(firstBraceIndex);
                xml = "\n" + "\t"<operation" + " "+ "name =" +
                "</comments>"+ "\n" processResult.result.queryResult("\n"<result"+ "
                "+ "type =" + ""+ ""+queryType+ ""+ ">"+ "\n"
                +moduleOperationStart+ xml
                +moduleOperationEnd+ "\n" + "</result>");
            }else if(resp.contains("Not consistent.")|| resp.contains("SATISFIABLE.")||
            resp.contains("NOT SATISFIABLE.")|| resp.contains("CANNOT
            DECIDE.")){
                int firstBraceIndex = resp.indexOf("sparq>") + 6;
                queryResult = resp.substring(firstBraceIndex);
                xml = "<comments>"+ queryResult + "</comments>"+ "\n";
                processResult.result.queryResult("\n"<result"+ " "+ "type =" + ""+
                ""+queryType+ ""+ ">"+ "\n"
                +moduleOperationStart+xml+moduleOperationEnd
                + "\n" + "</result>");
            }else if( resp.contains("(")){
                int firstBraceIndex = resp.indexOf("sparq>") + 6;
                queryResult = resp.substring(firstBraceIndex);
                processResult.result.queryResult("\n"<result"+ " "+ "type =" + ""+
                ""+queryType+ ""+ ">"+ "\n"
                + convertResultIntoXML(queryResult, queryType, operationName,
                returns,binaryTernaryOperationType,binaryTernaryOperationName,calcu
                lusT   ype)
                + "\n" + "</result>");
                processResult.sendQuery = false;
            }
        }
        catch(Exception e){
            System.out.println("[problem in engine data processing ]" +
            e.getMessage());
        }
    }
}

```

Similarly general error report is as also extracted based on String “An error occurred” and represented as error with in comments tag. *PocessEngineResult()* is used to call other methods like *ConvertResultIntoXML()*, *getCompleteNetworkXML()* and *getResultNetworkXML()* that are used to remove black-spaces, quotations and split result in to sub-networks based on different java regular expressions. It is used to extract relations and entities. *QueyReceiver* class override *Run()* method defined in *TimerTask* class.

```

public void run() {
    String line = "smthing";
    String result = "";
    Boolean shouldStop = false;
    Boolean sparqTagReceived = false;
    try {
        while(!shouldStop){
            line = EngineConnector.in.readLine();
            System.out.println("lines read from sparq..." + line);
            if(line.contains("sparq>")){
                sparqTagReceived = true;
            }
            if(sparqTagReceived){
                String module = processResult.currentQueryType;
                result += line;
                if(module.matches("constraint-reasoning")||module.matches("qualify")){
                    if(line.length()==0){
                        shouldStop=true;
                    }
                }else if (module.matches("compute-relation")){
                    if(line!="null"){
                        shouldStop=true;
                    }
                }else if(module.matches("a-reasoning")){
                    shouldStop=true;
                }
            }
        }
    }catch (IOException e1) {
        System.out.println(" Problem in RUN method... ");
        e1.printStackTrace();
    }
    System.out.println(" orginal result: " + result);
    int indexOfSparqTag = result.indexOf("sparq>");
    String requiredResult = result.substring(indexOfSparqTag, result.length());
    System.out.println("complete result : " + requiredResult);
    try{
        processEngineResults(requiredResult, processResult.currentQueryType,
            processResult.subQueryType, processResult.returnsType ,processResult
            .binaryTernaryOperationType,
            processResult.binaryTernaryOperationName,processResult.calculusType);
    }catch(Exception e){
        System.out.println("[error in reading data from SparQ");
        e.printStackTrace();
    }
}
}

```

processEngineResults() takes arguments like *resp*, *queryType*. The operation-name and returns etc., basically these arguments are used to compare and process module specific result conversion. It extracts substring result from original required result by removing unwanted string like <sparq> and blank spaces. The method is used to call

set of other methods like *convertResultIntoXML()*, *getcompleteNetworkXML()* and *getResultNetworkXML()* that extracts final result by removing blank spaces and parentheses and converts in to XML structure. The final given XML result is passed as argument in *queryResult()* to display as query result at user interface.

```
private String getResultNetworkXML(String subResult){
    xml = "";
    String relationXML = "";
    try{
        String relation = subResult.substring(1, subResult.length()-1).trim();
        if(relation.contains(" ")){
            String[] allRelations = relation.split("[ ]");
            for(int r = 0; r <= allRelations.length; r++){
                System.out.println(allRelations[r]);
                relationXML += "<relation>" +
                    allRelations[r] + "</relation>" + "\n";
            }
        }else {
            relationXML = "<relation>" + relation + "</relation>" + "\n";
        }
        xml =
            "<entity>" + subResult.substring(0, 1) + "</entity>" + "\n" +
            relationXML + "\n" + "<entity>" +
            subResult.substring(subResult.length()-1, subResult.length()) +
            "</entity>" + "\n";
    }catch(Exception e){
        System.out.println("Problem in getResultNetworkXML() " +
            e.getCause());
    }return xml; }

```

convertResultIntoXML() method contains arguments like *result*, *module-name*, *operation-name*, *operation type*, *calculus-name*, *calculus-type*, *binaryTernayoperationName*, *binaryTernaryOperationType* and returns (*first/all*). It contains module based set of programming codes to convert given result in XML format.

Each module specific queries generates specific type of result based on analysis these results are categorized and programmed to display in XML format *getCompetleNetworkXML()* takes argument as list, that contains all characters defined in result. Method removes all parentheses from result by replacing with blank spaces and forwards as *subResult* to *getResultNetwrokXML()* method. It is calling other two defined method like *getTernaryResultNetworkXML()* and *getResultNetworkXML()* with the help of conduction if *calculuType* matches with binary or ternary as both binary and ternary calculi returns different constraint-networks.

```

private String getCompleteNetworkXML(String[] list){
    xml = "";try{ for(int r=0; r < list.length; r++){
        if(list[r].length() > 1){
            String subResult = list[r].trim().replaceAll("[()",
            "").trim();
            subResult = subResult.replaceAll("[[]]", "").trim();
            if(processResult.calculusType.matches("binary")){
                xml += getResultNetworkXML(subResult);
            }else if(processResult.calculusType.matches("ternary")){
                xml+=getTernaryResultNetworkXML(subResult);
            }
            xml += getResultNetworkXML(subResult);
        }}catch(Exception e){
            System.out.println(" [problem in getCompleteNetworkXML
            method...]" +e.getMessage());
        }
        return xml
    }

```

```

private String getResultNetworkXML(String subResult){
    xml = "";
    String relationXML = "";
    try{
        String relation = subResult.substring(1, subResult.length()-1).trim();
        if(relation.contains(" ")){
            String[] allRelations = relation.split(" ");
            for(int r = 0; r < allRelations.length; r++){
                System.out.println(allRelations[r]);
                relationXML += "\t"+"\t"+"<relation>" + allRelations[r] +
                "</relation>"+"\n" ;
            }
        }else {
            relationXML = "\t"+"\t"+"<relation>" + relation +
            "</relation>"+"\n" ;
        }
        xml =
        "\t"+"\t"+"<entity>" + subResult.substring(0, 1) + "</entity>"
        + "\n" + relationXML + "\t"+"\t"+"<entity>" +
        subResult.substring(subResult.length()-1, subResult.length()) +
        "</entity>"+"\n";
    }catch(Exception e){
        System.out.println("Problem in getResultNetworkXML() " + e.getCause());
    }return xml;
}

```

```

private String convertResultIntoXML(String result, String queryType, String
subQueryType, String returns){
    xml = "";
    if(queryType.matches("compute-relation")){
        String afterRemovingBraces = result.trim().substring(1, result.length()-2);
        String[] relations = afterRemovingBraces.split(" ");
        = for(int r=0; r <= relations.length ; r++){
            xml += "<relation>" + relations[r] + "</relation>" + "\n";
        }
    }else if(queryType.matches("qualify")){
        String[] list = result.split("[ ]");
        xml += getCompleteNetworkXML(list);
    }else if(queryType.matches("constraint-reasoning")){
        String[] splitNetwork = null;
        String subNetWork = "";
        String[] list = null;
        String subSubType = returns;
        if(subQueryType.matches("scenario-consistency")){
            if(subSubType.matches("all")){
                splitNetwork = result.split("[0-9]");
                int whereIsLastBraces = result.indexOf(")");
                subNetWork = splitNetwork[0].trim();
                list = subNetWork.split("[ ]");
                xml += "\n" + "<comments>" +
result.substring(whereIsLastBraces+2, result.length()) +
"</comments>" + "\n";
                xml += getCompleteNetworkXML(list);
            }else if(subSubType.matches("first")){
                result = result.replace(" ", "");
                list = result.split("[ ]");
                xml += getCompleteNetworkXML(list);
            }
        }
    }
}

```

5.1.2.5 QuerySender Class

A class QuerySender contains set of methods that provides functionalities like *connection over TCP/IP, register generated result, close connection and sending query to reasoner.*

isConnectionEstablished() method is used to verify the established connection, that returns true value on the successful connection with the reasoning engine, it calls method *isEngineConnected()* of class *EngineConnector*.

closeTheConnection() method is used to close connection with reasoner and close all opened socket like Input stream and output stream. It calls *closeTheConnection()* method define in *EngineConnector* class through connector object of *EngineConnector* type.

```
public void startConnectionAt(String url, int port){  
    connector.createConnection(url, port);  
    }public void closeTheConnection(){  
        connector.closeTheConnection();  
    }public Boolean isConnectionEstablished(){  
        return connector.isEngineConnected();  
    }
```

sendXMLQueryToReasoner() method takes given XML file as argument and passes file to *createXMLDocumentTree()* method that used for generating Document tree. Method *sendXMLQueryToReasoning()* is accessing all XML tags and there values to generate sequence of string based on the module specific syntax. Method access all getters defined in *IReasonQueryGenerator* interface using object named “*stringGenerator*”

```

public void sendXMLQueryToReasoner(String xmlFile) {
    File query = new File(xmlFile);
    Document doc =
stringGenerator.createXMLDocumentTree(query);
    String module = stringGenerator.getModuleName(doc);
    String calculusName = stringGenerator.getCalculusName(doc);
    String calculusType=stringGenerator.getCalculusType(doc);
    String sparqQuery = "";
    NodeList calculusNodes =
stringGenerator.getCalculusNodes(doc);
    If(calculusType.matches("binary")){
        if(module.matches("compute-relation")){
            sparqQuery =
stringGenerator.makeComputeRelationSparqString(doc);
        }else if(module.matches("constraint-reasoning")){
            String operationName =
stringGenerator.getOperationName(doc).toLowerCase();
            if(operationName.matches("algebraic-closure")){
                sparqQuery = module + " " + calculusName + " " +
operationName + " " +
.....
            }
        }
        if (calculusType.matches ("ternary")){
            if(module.matches("constraint-reasoning")){
                String operationName =
stringGenerator.getOperationName(doc).toLowerCase();
                if(operationName.matches("algebraic-closure")){
                    sparqQuery = module + " " + calculusName + " " +
operationName + " " + .....
                }
            }
        }
        System.out.println("[Sending to sparq: ]" + sparqQuery.trim());
        connector.sendQueryToReasoner(sparqQuery.trim());
        timer=new timer();
        reveiver=new QueryReceiver(connector,this);
        timer.schedule(receiver,500);
    }
}

```


6. Case Study and Demonstration

6.1 Case-Study

6.1.1 Integration of Reasoner with GIS

GIS is fundamentally about solving real-world problems, it is used to improve many of our day-to-day working and living arrangements. Today wider availability of GIS through the internet, as well as through organization-wide local area networks, more and more individuals and organizations find themselves using GIS to answer the fundamental question, *where?* And to solve complex problems that are of real-world concern. There are huge range of applications of GIS, integrated with corporate information system (IS) including topographic base mapping, socio-economic and environmental modeling, global(interplanetary) modeling and education (Longley, M. F. Goodchild, 2009).

Reasoner like SparQ contains qualitative reasoning calculi with defined vocabulary in the form of composition table, the transitivity table introduced by Allen in temporal calculus. It is a fixed vocabulary of relations defined in qualitative spatial reasoning calculi normally, this will constitute a JEPD set, such a table enables one to answer the following question like $R1(x, y)$ and $R2(y, z)$, what are the possible relations from the set (R_i) that can hold between (x) and (z) ?. The integration of reasoner with spatial application is possible through implementation of platform independent middleware framework (API) and plug-in as user interface. Java based plug-in as application interface will facilitate GIS users to write queries for reasoning on real data by selecting features directly. These spatial queries in XML format will pass to reasoner via TCP/IP connection and extract result from reasoner in XML format.

In case study, I considered integration of Cardinal Direction Calculus proposed by Frank, 1991. The purpose of study is to explore functionality of qualitative spatial reasoning calculi in GIS particularly orientation information using spatial data. Orientation information about the urban environment is directly available to human being through perception. People perceive the arrangement of entities in space, categorize them as spatial relationships and describe them as spatial expression in language. These Spatial representations need frame of reference, Franks, 1991 introduced methods to partition the orientation information, which are known as projection-based model, cone-based model and directions with natural zone, he introduce natural zone to solve the problem of how to determine direction when two point are too close together. Cardinal direction is a binary calculus that describes binary function between two objects in the space $(P1, P2)$ that map in to a symbolic direction. The set of symbol depends upon the granularity, usually human deals with two level of granularity for directions like $\{N, S, E, W\}$. Freksa, 1992, introduced the double cross model, which is based on the projection-based model of directions. These models are represented below.

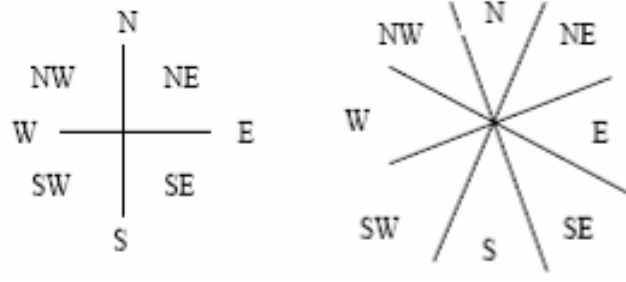


Figure 23: Projection Based Model and Cone-based Model of Cardinal Direction introduced by Frank, 1991

Cardinal Direction as Cone:

The angular direction between the observer's position and destination point is most often used prototypical concept of cardinal direction. This model of cardinal direction has a property that the area of acceptance for any given direction increases with respect to distance (Frank, 1991).

The quarter turn of the given direction can be defined as

$$q(N) = E, q(E) = S, q(S) = W, q(W) = N, q(0) = 0$$

The composition table for the given calculi is defined in chepter-2.

Cardinal Directions Defined by Projections:

A projection based cardinal direction defined by par-wise oppositions and each pair divides the plan onto two half-planes. By dividing space in four half-planes provides nine regions including natural zone, provides reliable solution to identify relative position of points on earth. For the (N-S) direction there are three values for direction are $d_{ns} = \{N, P, S\}$ and similarly for {E-W} direction, they are $d_{ew} = \{E, Q, W\}$. The intuitive properties of cardinal directions are describe in the form of algebra with two operations named *inverse* and *composition* operation.

In GIS direction of traveling and road data is representing as line segment between given point's source (P1) and destination (P2). By applying operations, defined in cardinal direction calculus, we can deduce the inverse direction from P2 to P1.

$$inv (dir (P1, P2)) = dir (P2, P1)$$

$$dir (P1, P2) \otimes dir (P2, P3) = dir (P1, P3)$$

Composition operation merges two contiguous paths, from (P1 to P2) and (P2 to P3), into a single path from (P1 to P3). The operation of composition is a basic step of the inference process. In case study, I consider Munster City data, assume that we have three 2D-point objects A (lake), B (small settlement) and C (small settlement). By integrating reasoner like SparQ with GIS application using API provides easy accessibility to reasoning calculi for reasoning on given data, The given data is processed by API to convert into SparQ specific syntax and extract result as qualitative description (possible cardinal directions between given entities). Java based plug-in at GIS application will support user to write a query in XML format like

```

<module name="qualify">
  <calculus name="cardir">
    <controlMode>all</controlMode>
    <entity type="51.956501 7.614026">A</entity>
    <entity type="51.97041 7.60673">B</entity>
    <entity type="51.968348 7.638659">C</entity>
  </calculus>
</module>

```

Finding the direction between (A and C), (A and B) and (B and C) based on given real coordinate value is possible through using qualify module in SparQ. Qualify convert quantitative descriptions of scenes in qualitative scenes and display possible relation between 2-D points.

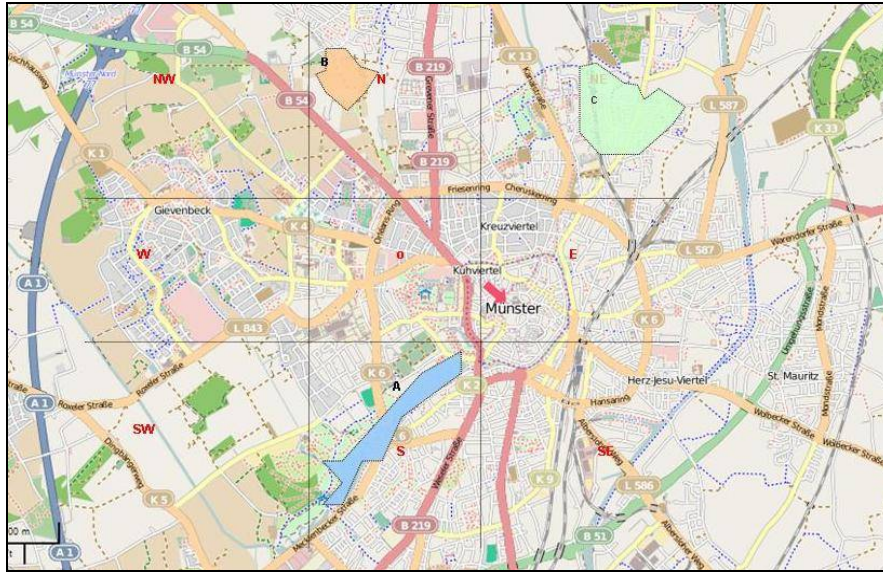


Figure 24: Orientation representation of objects P1, P2 and P3 using projection-based model Frank 1991 in Open Street Map

A middleware API will receive this XML query and parse it to generate SparQ specific query syntax, API is used to establish connection with reasoner and passes the query for reasoning. It extracts result in XML format that facilitate users to apply further processes to display in GIS application. The given result represents possible directions between objects (A, B) (B, C) and (A, C), which is represented as XML format

```

<result>
  <entity>A</entity><relation> s </relation><entity>B</entity>
  <entity>A</entity><relation> sw </relation><entity>C</entity>
  <entity>B</entity><relation> w.</relation><entity>C</entity>
</result>

```

As we know that qualitative approach to spatial reasoning does not rely on a coordinate plan and does not attempt to map all information in this framework. It can

deal with imprecise data and therefore, yields less precise result than quantitative approach (Freska, 1991).

The integration of reasoning calculi like cardinal direction calculus with GIS, although provides imprecise data but it uses verbal descriptions, such imprecise descriptions are necessary in query language like “*find all restaurants about 2 miles North of town A and East of Town B*”, such queries are easy to understand by normal GIS user.

6.2 Demonstration on Spatial Data

For reasoning demonstration using spatial data, I used OpenJUMP GIS application. It is open source JAVA based software, where we can easily extend application by developing required plug-ins and tools. I developed plug-in that contains *text-area*, *text-field* and buttons to control activates like *send-button*, *connect-button* (establish connection with reasoner) and *disconnect-button* (disconnect established connection). The developed API can be used as public API and accessed via *http*: or can integrate with any java based application. To demonstrate API activities on spatial data, I integrated API with plug-in as *.jar file format and placed in *openJUMP /bin/txt/ plugin.jar*, from where, it can easily load as extension.

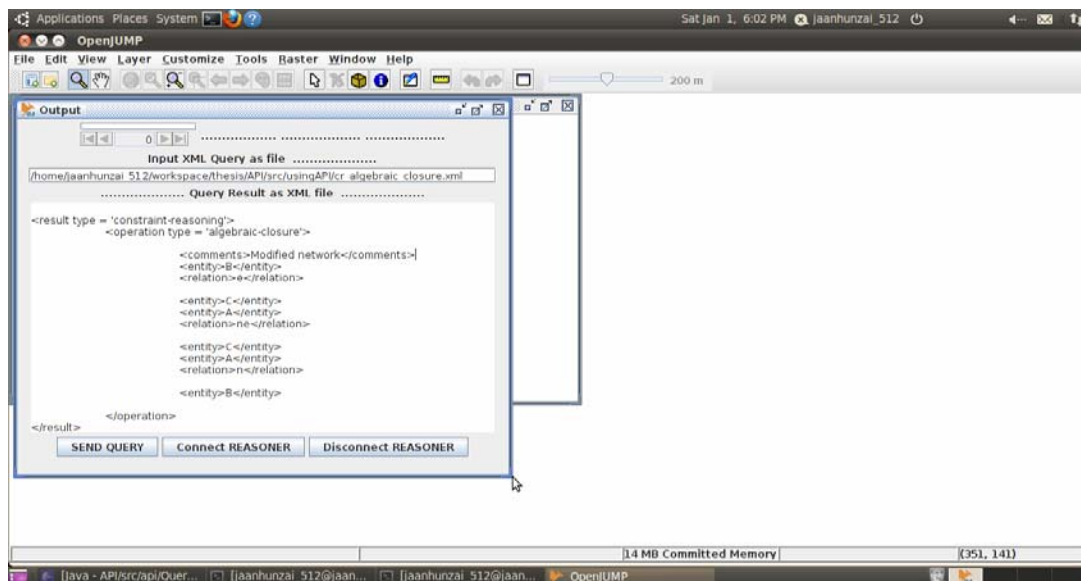


Figure 25: OpenJUMP plug-in to send and receive data

Text-field: is used to provide query path which must be written in XML format.

Text-Area: text-area is used to display query result in XML format received from reasoner.

ConnectReasoner: button is used to establish connection with reasoner over TCP/IP (localhost, 4444) and authenticate connection, it also activates input/output streams to read and write data over TCP/IP.

DisconnectReasoner: button is used to close all established connections during communication like to close both Input/output streams.

SendQuery: SendQuery button confirms given query (XML format) file path and sends query to reasoner.

For experiment, I considered Munster street data (*Muenster-street.shp* and

roadIntersections.shp) and draw three 2-D points (A, B and C) by considering road intersections within Munster City. Both shape files are geo-referenced and projected *Geographic coordinate system GCS_European_1950*

Datum D_European_1950

Linear Unit: Meter

Projection: Albers

X Y coordinate of intersections are given below:

- i. PointId (A) = (3404258.7402544618 5758202.000064869)
- ii. PointId (B) = (3405184.9683120195 5758842.075137064)
- iii. PointId (C) = (3405056.2571234703 5759567.499447379)

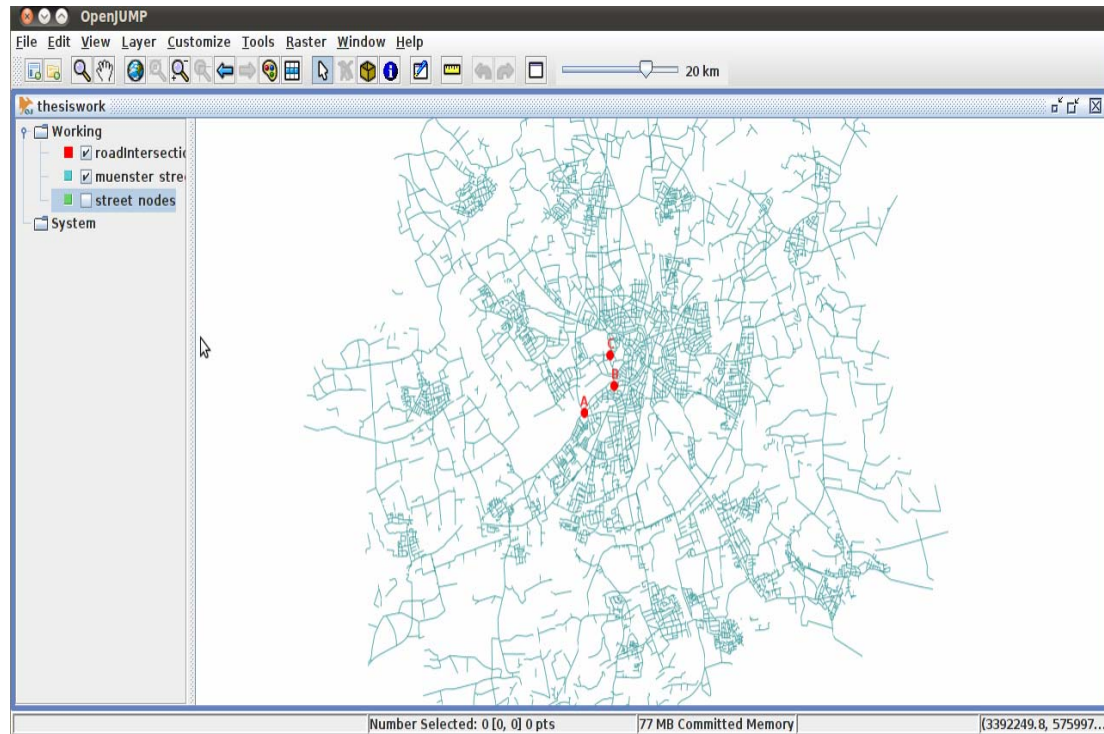


Figure 26: Munster Street data with road-intersections as points (A, B and C)

After extracting X Y coordinates, I defined query in XML format to forward to reasoner over TCP/IP. Query contains *module-name*, *calculus-name* and other required parameters like *controlMode*, *entity* with defined type and *entity-value* as identifier. As we know that qualify module in SparQ takes quantitative descriptions in the terms of coordinate values of given points and return qualitative descriptions of points in the terms of entities and relationship between entities. The API will take query in XML format and convert it into SparQ syntax for reasoning. I apply cardinal-direction (CARDIR) calculus on defined road-intersections (A, B and C) to identify possible cardinal-direction relations like [n, ne, e, se, s, sw, w, nw]. As control-mode value, I selected “all”, that returns the relations between every object and every other object will be included. In XML query, I defined given coordinate values [X, Y] as entity-type and *PointId* as entity-Id, it is given below.


```

<?xml version="1.0" encoding="UTF-8"?>
<module name="qualify">
  <calculus name="cardir">
    <controlMode>all</controlMode>
    <entity type="3404258.7402544618
      5758202.000064869">A</entity>
    <entity type="3405184.9683120195
      5758842.075137064">B</entity>
    <entity type="3405056.2571234703
      5759567.499447379">C</entity>
  </calculus>
</module>

```

API converts given XML format query into SparQ specific syntax and forwarded to reasoner over TCP/IP. SparQ processes the query and generates result in SparQ module specific syntax. The generated result is further processed with the help of API to convert into defined standard XML structure for particular module.

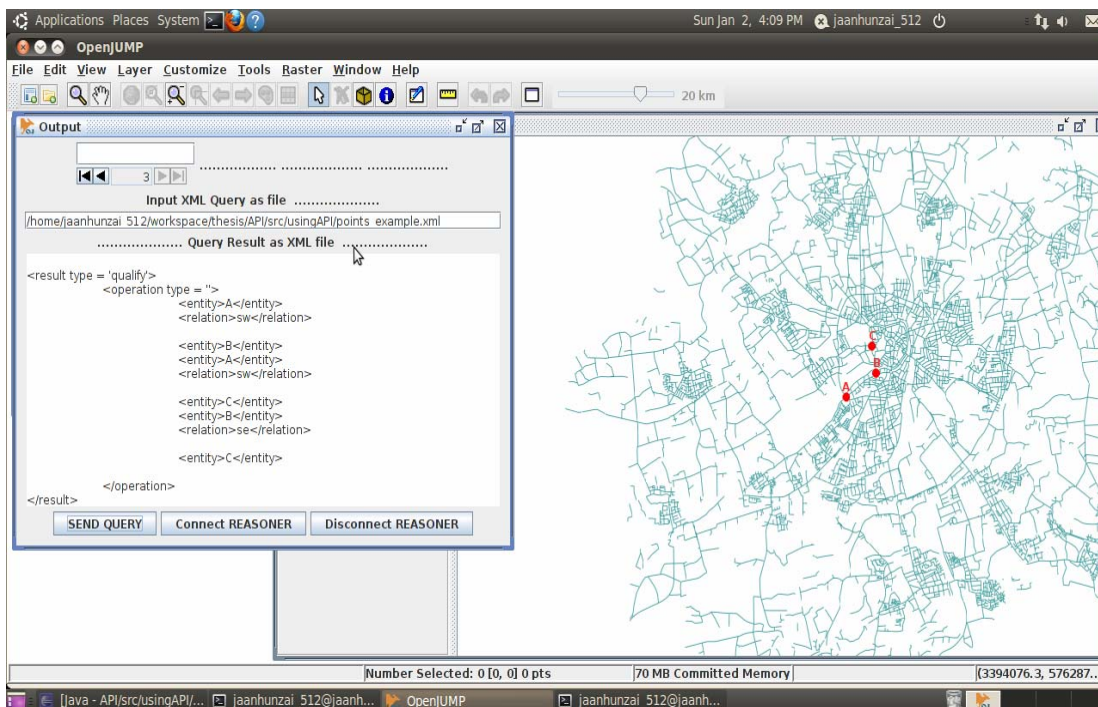


Figure 27: Reasoning result in XML format on given road-intersections as points (A, B and C).

The given XML as a result constrains tags like *result-type*, *operation-name* and *type*, *entity-name* and *relation-name*. *Result-type* is used to identify type of module used for reasoning on given data, *operation-name* and *type* defines modules specific operation name and its type. Here in this case, *qualify* module doesn't contain any specific operation therefore; it will forward empty space as *operation-name* and *operation-type*, will be module-name used for reasoning. *Entity-tag* represents used entities in query and *relations-tag* represents possible cardinal-direction relations between given entities like,

[A south-west (sw) B][A south-west (sw) C] [B south-east (se) C]

```

<result type = 'qualify'>
  <operation name = "",type= "qualify">
    <entity>A</entity>
    <relation>sw</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>sw</relation>
    <entity>C</entity>
    <entity>B</entity>
    <relation>se</relation>
    <entity>C</entity>
  </operation>
</result>

```

One of the major advantage of result in XML format is reusability, as result contains possible relationship between given entities as constraint-network, We can generate sub-queries from given result with little modification and can reuse it for further reasoning like by applying constraint-reasoning specific operation (algebraic-closure and scenario-consistency) to verify network consistency . The sub-query from resultant query is given below.

```

<?xml version="1.0" encoding="UTF-8"?>
<module name="constraint-reasoning">
  <calculus type="binary" name="cardir">
    <operation type="constraint-reasoning">algebraic-closure</operation>
    <entity>A</entity>
    <relation>sw</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>sw</relation>
    <entity>C</entity>
    <entity>B</entity>
    <relation>se</relation>
    <entity>C</entity>
  </calculus>
</module>

```

I applied algebraic-closure operation of constraint-reasoning module by modifying given result to check network consistency of generated constraint-network. As result SparQ returns constraint-network with comments like “*unmodified network*”. Given comments indicate that the constraint-network generated from coordinate values of road intersections are algebraically closed. Result of sub-query is given below.

```

<result type = 'constraint-reasoning'>
  <operation name = 'algebraic-closure', type = 'constraint-reasoning'>
    <comments>Unmodified network</comments>
    <entity>B</entity>
    <relation>se</relation>
    <entity>C</entity>
    <entity>A</entity>
    <relation>sw</relation>
    <entity>C</entity>
    <entity>A</entity>
    <relation>sw</relation>
    <entity>B</entity>
  </operation>
</result>

```

Similarly I applied dipole relation algebra (DRA-24) on line segments (A, B and C) representing streets in Munster, Germany. Queries using DRA-24 calculus contains parameters like, *module-name*, *calculus*, *controlMode* and *entity-type*. DRA-24 takes dipoles as base entities, which are oriented line segments. The object description of a dipole is in the form (*name*, *Xs Ys Xe Ye*), each dipole contains *entity-Id*, starting points (*Xs Ys*) and ending points (*Xe Ye*) of dipole.

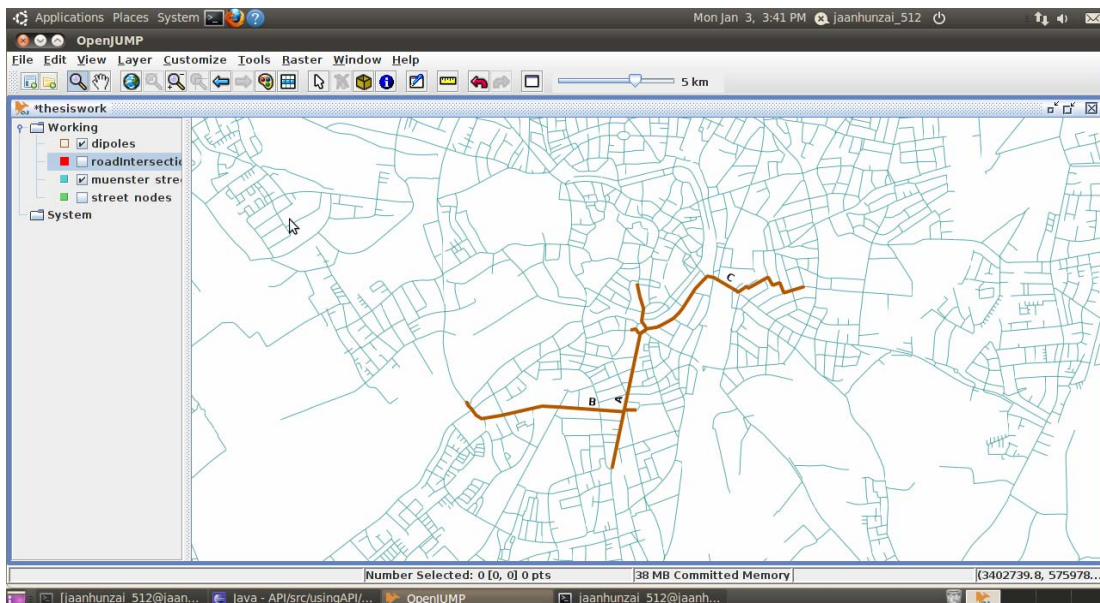


Figure 28: Line segment (A, B and C) representing Munster road, used for reasoning

Query using XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="qualify">
  <calculus name="dra-24">
    <controlMode>all</controlMode>
    <entity type="3405410.3201312614 5757710.502361109
3405618.164074955 5759248.54754444">A</entity>
    <entity type="3405616.7541148234 5758197.3177945595
3404197.3360080197 5758261.044056213">B</entity>
    <entity type="3405690.9158829013 5758867.441955996
3406996.0213157325 5759225.303069583">C</entity>
  </calculus>
</module>
```

Generated result from reasoner (SparQ) contains set of possible relations between defined lines segment like:

[A (rllr) B] [A (rllr) C] [B (rrrr) C]

A line segment “A” has relation (rllr) with “B” means *starting point of “B” is on right-side and ending-point is on left-side of “A” and “starting-point of “A” is on left-side and ending-point is on right-side of “B”*.

Line segment “A” has relation (rllr) with “C” means, *starting point of “C” is on right-side and ending point is on left-side of “A” and starting point of “A” is on left-side and ending point is on right side of line “C”*.

Similarly line segment “B” has a relation (rrrr) with line segment “C” means *both starting and ending points of “C” are on right-side of “B”, and starting and ending points of “B” are also right-side of line segment “C”*. Reasoner inferred possible relation between “A” and “C” which is **[A (rllr) C]** based on given relations between “A” and “B”, “B” and “C”.

```
<result type = 'qualify'>
  <operation type = ">
    <entity>A</entity>
    <relation>rllr</relation>
    <entity>B</entity>
    <entity>A</entity>
    <relation>lrll</relation>
    <entity>C</entity>
    <entity>B</entity>
    <relation>rrrr</relation>
    <entity>C</entity>
  </operation>
</result>
```

The above mentioned result contains constraint-network provided against the given quantitative description of dipoles. The result is further used as sub-query by applying constraint-reasoning module operation (scenario-consistency) to validate that, the given constraint-network has any scenario that is algebraically closed. It

returns possible scenarios. Modified sub-query used constraint-reasoning module to check algebraically closed scenario in the given network.

```
<?xml version="1.0" encoding="UTF-8"?>
  <module name="constraint-reasoning">
    <calculus type="binary" name="dra-24">
      <operation type="constraint-reasoning">scenario-
        consistency</operation>
      <return>all</return>
      <entity>A</entity>
      <relation>rllr</relation>
      <entity>B</entity>
      <entity>A</entity>
      <relation>lrll</relation>
      <entity>C</entity>
      <entity>B</entity>
      <relation>rrrr</relation>
      <entity>C</entity>
    </calculus>
  </module>
```

The result generated from above mention sub-query contains constraint-network with comments like “*1 scenario found, no further scenarios exist*”. It indicates that the given network is algebraically closed and it contains single relation between the given dipoles.

```
<result type = 'constraint-reasoning'>
  <operation name = 'scenario-consistency', type = 'constraint-reasoning'>
    <comments>1 scenario found, no further scenarios exist</comments>
    <entity>B</entity>
    <relation>rrrr</relation>
    <entity>C</entity>
    <entity>A</entity>
    <relation>lrll</relation>
    <entity>C</entity>
    <entity>A</entity>
    <relation>rllr</relation>
    <entity>B</entity>
  </operation>
</result>
```

7. Conclusion, Shortcomings and Future Work

7.1 Conclusion

Qualitative reasoning deals with commonsense knowledge without using numerical computation. Spatial reasoning is present in our everyday's interaction with the geographical world, particularly in orientation and distance in the space, but GIS application that is used to carry out spatial tasks, does not support commonsense reasoning. Qualitative spatial reasoning enables computer to make predictions about spatial constraints (relations) between existing objects in specific domain. During last two decades a multitude of spatial constraint based calculi have been proposed and discussed in literature to represent different aspects of the space and temporal information qualitatively. The implantations of these spatial calculi are comparatively small due to different factors like selection of appropriate calculi and complication in calculus specific composition table development to integrate with in the spatial application. Reasoning engines like SparQ contains spatial calculi used to represent and reason about different aspects of the space based on defined calculi specific relations and operations; it provides broader range of service including integration capabilities with other applications.

In this thesis, there are two major components on which, I concentrate, first one is analysis of reasoning engine (SparQ) and second is API development that facilitate GIS users to reason on spatial data. SparQ is analyzed to identify commonalities between reasoning technique it supports and type of query syntax in each module used for reasoning. Based on analysis, I generalized all possible input query syntax like queries with constraint-networks, and queries with nested operations and their parameters (relations). Similarly SparQ output results are analyzed based on given modules and operation specific queries. These given results are generalized in main five categories like simple text, text and constraint-network, error etc.

Based on SparQ input and output syntax analysis, I developed a platform independent API that allows developers to integrate the reasoning engine with spatial applications like GIS. API provides a set of functionalities like establishing a connection with the reasoner through a GIS application, and sending and receiving queries over TCP/IP. It contains two major activities that handle sending and receiving queries and their results over TCP/IP. Sending involves taking queries in a predefined XML structure from GIS application, converts it into SparQ specific syntax and forwards the formatted query to SparQ over TCP/IP for reasoning. Receiving activity extract results from reasoner, converts these results into defined XML structure and forward to the GIS application over same connection.

The main advantage of API is that it supports machine and human understandable language (XML), which can be enhanced and improved in terms of automating queries and reusability of received results. In general automating query means directly generating queries by selecting spatial features from a visual representation of their data and a task that will use the selected features as input. An automated query in this context works more or less like an automated workflow in which the next reasoning task and its data input are defined by the user selected task and the

output of one of its preceding tasks. The developed API can easily integrate with any Java based application or can be accessed over a network or can be converted into a web-based API, to easily access and reason on spatial data from multiple platforms. As we know that, GIS application deals with spatial data including point, line and polygon features. To bring qualitative spatial reasoning close to GIS application needs mechanism that supports these spatial objects qualitatively. In reasoning engine SparQ qualify module takes quantitative descriptions of the scene to generate qualitative descriptions as a possible set of relations between given entities like points and line segments. Most of GIS data contains geometry type polygon to represent to objects in the space like regions. To bring reasoner (SparQ) close to GIS application needs improvements in qualify module, where we can able to apply qualitative reasoning on geometry type polygons entities. Such facilities will enable GIS users to use RCC and Cardinal Direction Calculi for reasoning on polygon type entities.

7.2 Shortcomings

The shortcomings that, I found during this research work are mostly related with reasoning engine (SparQ). SparQ is organized into four modules and each module has its own input syntax. The initial commands like *module-name*, *calculus-name* are common in all modules but the remaining commands and syntaxes are dependent upon module specific operations.

Constraint-reasoning operations takes constraint-networks as argument, the structure of constraint-networks vary with respect to module specific operation like for algebraic-closure and scenario-consistency takes same constraint-network structure as input. In contrast other operation of the same module such as refine and extend, for example, takes constraint-networks as argument with extra parentheses. Compute-relation module's defined operations take nested arguments as compositions of operations and relations. In some cases it takes single operation and relations as arguments enclosed with in quotes. This was very challenging task to define and structure in Java code. If each modules and their specific operation take same type of parameters as argument it will be easy to understand by users and developer.

There are different types of results and that these results are structure very differently. Most of the results are forwarded between the starting tag *sparq>* and an ending flag which can be a string of length zero, an end line (\n) character, a carriage return (\r) or some combination of these. Based on these tags we can easily extract required result. In contrast some modules like compute-relation and algebraic-reasoning, results are forwarded with starting tag *sparq>* without forwarding end flag, that provides difficulties to extract required result and to stop input stream. It will be better if SparQ provide result with in defined tags like *sparq>* and with ending flag with any special character or number like [-1 or -2] to indication ending of result. SparQ support limited set of command like quite, interactive (-i), port (-p).that are used to start SparQ in interactive mode and to provides allocate port for communication over TCP/IP There must be some command like clean sparq-buffer, stop-interactive mode with out leaving shell. In some case SparQ provides result in composition of text and constraint-networks, the syntax and sequence of such a result

is different with respect to module specific operations, there must be some common standard structure for such a type of results. Major shortcoming in SparQ is basis-entity support. SparQ doesn't support polygon type basis-entity; it deals with 1d-point, interval, 2d-points, dipoles etc. To use SparQ with spatial application as reasoning tool, it is very important that reasoner must support polygon type geometry especially for applying qualify module, as spatial data contain points, lines as well as polygons.

Another major shortcoming of the research work is related with integration of multiple reasoning engines with GIS application under the same framework. At the moment developed framework support only single reasoner (SparQ) as it provides services to integrate with our own application, in contrast reasoning engine (GQR) doesn't support integration services.

7.3 Future Work

The framework (API) developed in this thesis is limited in several respects like selection of qualitative reasoning engines, automating spatial queries and representing results visually on the client side.

In this thesis, I consider reasoning engines SparQ and GQR, after analysis both reasoner engines, I came to know that GQR supports only binary queries and it doesn't provides functionalities to integrate with our own applications. In contrast SparQ support both binary and ternary calculi and can easily integrate into own application with the help of TCP/IP connection. Therefore I considered only SparQ engine to integrate with spatial application and developed API based on particular reasoner. It is possible to upgrade developed API to interact with other reasoning engines and define their modules specific syntax in API. Integration of multiple reasoning engines with the help of API can provides opportunity to reason on spatial data by selecting specific reasoner.

Algebraic-reasoning module in SparQ is used to provide consistency checking mechanism for given constraint-reasoning. It deals with reasoning about real-valued domain using algebraic geometry techniques. It contains under development operations like consistency checking, compute calculus operations, operation analysis and qualification. During my further studies I would like to upgrade developed API, to interact with above defined algebraic-reasoning module specific operations through XML queries. Compute-relation module in SparQ allows to compute with the operations defined in the calculus specification. Module takes operations (binary, ternary) and basic relations a parameter depend upon arity of used operation. The developed API deals with basic type of queries using compute-relation module, in my future studies I like to implement all type of structure that compute-relation module supports.

Due to shortage of time I implemented binary calculi specific query syntaxes and only constraint-reasoning specific query syntaxes for ternary calculi in this API. I would like to upgrade API to integrate all module specific queries syntaxes for ternary calculi as well. At the moment, the developed framework supports single reasoner (SparQ). I would like to continue my research to find out possibilities to

integrate multiple reasoning engines under the same framework, where user will be able to use multiple engines for reasoning on spatial data. As I mentioned above SparQ doesn't support geometry type polygon, in my future studies, I would like to work on, how to integrate geometry type polygon in SparQ to provide facilities for reasoning on polygon type entities like other geometry type entities.

8. Bibliographic Reference

1) Articles of periodicals

1. C. Feska, 1990. Representation und Verarbeitung raumlichen Wissens. Springer-Verlag, Barlin.
2. Christian Freksa, Qualitative spatial reasoning institute for Informatik Technische Universitat Munchen.
3. Anthony G., Cohn, Brandon B., John G., Nicholas M.G., 1997, Qualitative Spatial Representation and reasoning with region connection calculus.
4. A. G Cohn, 1997, qualitative spatial representation and reasoning techniques.
5. C. Freksa, Springer, Berlin, 1992a. Using orientation information for qualitative spatial reasoning. In A. U. Frank, I. Campari, and U. Formentini, editors, Theories and methods of spatiotemporal reasoning in geographic space, pages 162–178.
6. O. Stepankova, V. Marik, R. Trapple (eds), 1992, An introduction to qualitative reasoning, Advanced Topic in Artificial intelligence.
7. F. Dylla, L. Frommberger, J. O. Wallgrun and D. Wolter, 2006. SparQ: A toolbox for qualitative spatial representation and reasoning. In Qualitative Constraint Calculi: Application and Integration, Workshop at KI 2006, 79–90.
8. A. David, Randell, Z. Cui, A.G. Cohn, 1992, A Spatial logic based on regions and connection.
9. Christian Freksa, R. Rohrig, 2000, Dimensions of Qualitative spatial reasoning.
10. M. Westphal, S. Woffl, Z. Gentner, GQR: A first solver for Binary Qualitative Constraint Networks.
11. A. Frank, 1992, Qualitative spatial reasoning about Distances and direction in Geographic Space.
12. Z. Jing, D.M. Mark, Z.Z. Rong, The new Reference frame about the spatial orientation expression for Way-Finding.
13. Jan Oliver Wallgrun, Lutz Frommberger, Frank Dylla, Diedrich Wolter, Frank, Christian, Qualitative Spatial Representation and Reasoning in the SparQ-Toolbox.
14. Nebel B., 1997. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ORD-Horn class.

2) Articles / book chapters

1. R. Jochen, N. Bernhard. Qualitative spatial reasoning using constraint calculi, Australian National University and Albert Ludwigs Universitat Freiburg.
2. Jan Oliver Wallgrun, Lutz Frommberger , Frank Dylla, Diedrich Wolter, January 13, 2009, SparQ User Manual V0.7.
3. A. G. Cohn, 1997. Qualitative spatial representation and reasoning techniques.
4. Egenhofer, David Mark, 1995. Naïve Geography, National center for geographic information and analysis report 95-8.

5. U. Furbach, G. Dirlich, C. Freksa, 1985. Towards a theory of knowledge representation Systems, in W. Bibel & B. Pethoff (eds.) *artificial intelligence Methodology, System, Applications*.
6. J. F. Allen, November, 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM*, pages 832–843.
7. A. G. Cohn, and S. M. Hazarika, 2001. Qualitative spatial representation and reasoning: an overview. *Fundamental Informatics* 46:1–29.
8. Tobler, R. Waldo and S. Wineberg, 1971. A Cappadocian Speculation, *Nature* 231 (May 7) 39-42.
9. Kuipers, Benjamin, 1983. The cognitive Map: could it have been another way in spatial orientation. Edited by H.L. Pick and L.P. Acredolo. 345-359.
10. Christoph Schlieder, 1991, representing visible locations for qualitative Navigation.
11. P.A. Lonley, M.E Goodchild, D.J. Mauire, D.W. Rhind, 2005. *Geographic information System and Science*.
12. Reinefeld and P. B. Ladkin, 1997. Fast algebraic methods for interval constraint problems. *Annals of Mathematics and Artificial Intelligence*. Volume 19, 383–411.
13. H. Maruyama, K. Tamura, N. Uramoto., 2002. *XML and Java Developing web Application*, second edition.
14. Dalluege, 2006. OpenJUMP tutorial, department of Geomatik, Hafencity University Hamburg.
15. B. Bennett, D. R. Magee, A. G. Cohn, and D. C. Hogg, 2004. Using spatio-temporal continuity constraints to enhance visual tracking of moving objects. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004*, 922–926. IOS Press.
16. Duntsch, H. Wang and S. McCloskey, 1999. Relation algebras in qualitative spatial reasoning. *Fundamental Informaticae* 39(3):229–249.
17. R. Moratz, F. Dylla and L. Frommberger, 2005. A relative orientation algebra with adjustable granularity. In *Proceedings of the Workshop on Agents in Real-Time and Dynamic Environments (IJCAI 05)*.

3) electronic resources

1. Doug Tidwell, 2002, introduction to XML.
2. <http://www.ibm.com/developerworks/xml/tutorials/xmlintro/section5.html>.
3. http://en.wikipedia.org/wiki/Use_case
4. http://en.wikipedia.org/wiki/Class_diagram
5. <http://download.oracle.com/javase/tutorial/java/index.html>
6. <http://www.w3schools.com/w3c/default.asp>
7. <http://www.quirksmode.org/dom/intro.html>
8. <http://en.wikipedia.org/wiki/Parsing>
9. <http://download.oracle.com/javase/1.4.2/docs/api/index.html>

Student Declaration

I declare that the submitted work has been completed by me the undersigned and that I have not used any other than permitted reference courses or materials nor engaged in any plagiarism. All references and other sources used by me have been appropriately acknowledged in the work. I further declare that the work has not been submitted for the purpose of academic examination, either in its original or similar form, anywhere else.

Munster, 28th February 2011

..... (Matrikelnummer 368660)

(Signature)

.....
(Name)

2011

*Framework development for providing
accessibility to Qualitative spatial*

Sahib Jan





Masters Program in **Geospatial Technologies**

