DISTRIBUTED SPATIAL ANALYSIS IN WIRELESS SENSOR NETWORKS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2011

Farhana Jabeen
Computer Science

Contents

List of Tables

List of Figures

The main text of this dissertation contains 67,252 words including footnotes and endnotes. The appendices contain a further 12,687 words.

Abstract

Wireless sensor networks (WSNs) allow us to instrument the physical world in novel ways, providing detailed insight that has not been possible hitherto. Since WSNs provide an interface to the physical world, each sensor node has a location in physical space, thereby enabling us to associate spatial properties with data. Since WSNs can perform periodic sensing tasks, we can also associate temporal markers with data. In the environmental sciences, in particular, WSNs are on the way to becoming an important tool for the modelling of spatially and temporally extended physical phenomena. However, support for high-level and expressive spatial-analytic tasks that can be executed inside WSNs is still incipient. By *spatial analysis* we mean the ability to explore relationships between spatially-referenced entities (e.g., a vineyard, or a weather front) and to derive representations grounded on such relationships (e.g., the geometrical extent of that part of a vineyard that is covered by mist as the intersection of the geometries that characterize the vineyard and the weather front, respectively). The motivation for this endeavour stems primarily from applications where important decisions hinge on the detection of an event of interest (e.g., the presence, and spatio-temporal progression, of mist over a cultivated field may trigger a particular action) that can be characterized by an event-defining predicate (e.g., humidity greater than 98 and temperature less than 10). At present, in-network spatial analysis in WSN is not catered for by a comprehensive, expressive, well-founded framework. While there has been work on WSN event boundary detection and, in particular, on detecting topological change of WSN-represented spatial entities, this work has tended to be comparatively narrow in scope and aims.

The contributions made in this research are constrained to WSNs where every node is tethered to one location in physical space. The research contributions reported here include (a) the definition of a framework for representing geometries; (b) the detailed characterization of an algebra of spatial operators closely inspired, in its scope and structure, by the Schneider-Guting ROSE algebra (i.e., one that is based on a discrete underlying geometry) over the geometries representable by the framework above; (c) distributed in-network algorithms for the operations in the spatial algebra over the representable geometries, thereby enabling (i) new geometries to be derived from induced and asserted ones, and (ii) topological relationships between geometries to be identified; (d) an algorithmic strategy for the evaluation of complex algebraic expressions that is divided into logically-cohesive components; (e) the development of a task processing system that each node is equipped with, thereby with allowing users to evaluate tasks on nodes; and (f) an empirical performance study of the resulting system.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Dedication

Dedicated to my ***AMMA (MOM)***, who left this world very early, but have never left my heart.

Acknowledgements

First of all I would like to thank my supervisor, Alvaro A. A. Fernandes, for his guidance, vision, and time. His vision has opened me a new door in the field of wireless sensor networks. I would also like to thank my advisor, Professor Norman W. Paton, for his suggestions and providing a broad perspective for my work.

I can never thank my great friend Sohel Vhora for his constant suggestions, encouragement and support. Whenever I lost focus, he encouraged me with one of his unforgettable golden sentence "You are almost there!" that infused in me a new spirit to accomplish. I also wish to express my gratitude to Sarfraz Nawaz, Research Assistant, University of Oxford, for the long fruitful discussions on potential problems, precious suggestions, and technical support.

Thanks also to Christian Y. A. Brenninkmeijer and Ixent Galpin with whom I have worked on SNEE Project. I also wish to express my gratitude to Alasdair J. G. Gray, with whom I have shared office and many moments, during my write-up period.

I am extremely grateful to the School of Computer Science, The University of Manchester, for support that let me continue and survive the biggest adversity of my career, when my parent university (NUST, Pakistan) had discontinued funding and instructed me to return. I would like to thank Schlumberger Foundation for their financial assistance under their programme *Faculty for the Future*, which allows many deserving female researchers and students around the globe to complete their higher education for the benefit of their community.

I would like to express my gratitude to all the members at the School of Computer Science of The University of Manchester. Special thanks to all the members of the Information Management Group.

My special gratitude is due to my brother, Wisal Mohammad, for his continuing guidance, loving support and encouragement.

Thank you my family, for the Patience and Love Extended (I don't think I could have made up without it). Also grateful for the Encouragement that made me accomplish.

Publications

1. Farhana Jabeen, Alvaro A. A. Fernandes. Distributed Spatial Analysis in Wireless Sensor Networks. In Proceedings of the Sixteenth International Conference on Parallel and Distributed Systems (ICPADS04), IEEE, Dec. 2010.

   This publication forms the part of Chapter 5 and Chapter 7 of this dissertation.

2. Farhana Jabeen, Alvaro A. A. Fernandes. Monitoring Spatially-referenced Entities in Wireless Sensor Networks. In 7th International Conference on Ubiquitous Intelligence and Computing (MENS Symposium). UIC-ATC '10, IEEE, Oct. 2010.

   This publication forms the part of Chapter 5 and Chapter 7 of this dissertation.

3. Farhana Jabeen and Alvaro A. A. Fernandes. Impact on accuracy of deployment tradeoffs in localized sensor network event detection. In Second International Workshop on Localized Algorithms and Protocols for Wireless Sensor Networks (LOCALGOS), in conjuction with IEEE DCOSS, June 2008.

   This publication forms the part of Chapter 3.

4. Ixent Galpin, Christian Y. A. Brenninkmeijer, Farhana Jabeen, Alvaro A. A. Fernandes, Norman W. Paton. An Architecture for Query Optimization in Sensor Networks. ICDE 2008: 1439-1441

   This publication is on Architecture for Query Optimization in Sensor Networks and is not directly related to this dissertation.

5. Ixent Galpin, Christian Y. A. Brenninkmeijer, Farhana Jabeen, Alvaro A. A. Fernandes, and Norman W. Paton. Comprehensive optimization of declarative sensor network queries. In SSDBM, pages 339-360, 2009

   This publication is on Optimization of Declarative Sensor Network Queries and is not directly related to this dissertation.

Glossary

AGG	Aggregation
CBN	Common Boundary Node
CBS	Common Boundary Segment
CLUT	Common Localized Unit Triangle
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
DTE	Distributed Task Evaluation
DME	Distributed Membership Evaluation
DRG	Distributed Random Grouping
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIT	Edges Information Table
GIT	Geometric Information Table
GID	Geometric ID
GI	Geometry Induction
IDG	Intermediate Derived Geometry
LUT	Localized Unit Triangle
MAC	Medium Access Control
MEG	Multi-element Geometry
MBR	Minimum Bounding Rectangle
NIT	Neighbours Information Table
RP	Result Processing
ROSE	RObust Spatial Extension
SEG	Single Element Geometry
SNQPs	Sensor Network Query Processors
SQL	Structured Query Language.
TDMA	Time division multiple access
TC	Tree Construction
TD	Task Dissemination
TTL	Time to live
WSN	Wireless Sensor Network

Chapter 1

Introduction

A *Wireless Sensor Network* (WSN) is a collection of spatially-distributed nodes equipped with sensing, communication, processing and storage capabilities. These capabilities allow the nodes to measure properties of the physical world, to act as relays for forwarding data sensed elsewhere, to perform local processing before transmitting information, and to act as storage points for data.

WSNs allow for interaction with the environment at very high spatial and temporal densities. Since WSNs provide an interface to the physical world, each sensor node has a location in physical space, thereby enabling us to associate spatial properties with data. WSNs can perform periodic sensing tasks, therefore, we can also associate temporal markers with data. Each value measured by a sensing device in a particular node can have associated with it where, when, what, i.e., the location of the sensor node at the time of measurement, the time at which measurement was taken and what is the measured value, respectively. This can be important in many applications scenarios, e.g., in automatic detection, avoidance, and recovery from environmental disasters. As scientific understanding of physical phenomena presupposes a study of their manifestation in time and space, this makes WSNs well-suited for real-time monitoring, control, and analysis of transient physical phenomena (e.g., a moving band of rain, a shape-shifting region of low temperature).

As an example of the usefulness of this kind of information, consider the following context. Efficient water management is a major concern for farmers of many crops. Imagine that a farmer has deployed sensor nodes [Ulr08, BBB04], and is interested in part of a field where the soil moisture has dropped below a certain threshold, so that only those parts are irrigated, given the limited water supply. The nodes sense the soil moisture and, using their short-range radio, communicate with each other to send the real-time information from the fields to the farmer. WSNs therefore allow the farmer to get a real-time digital picture, in the form of sensed measurements, of the physical world. The raw data being collected enables the farmer to see what is going on in fields and to adjust their management strategies. In addition, sensor node can also help in frost monitoring [PE08] and fighting fungal disease in the field [A05, BBB04]. Such diseases tend to spread under certain temperature and humidity conditions. The collected information allows the farmers to control water, fertilizer, planting density, and pest-disease control programs. Many factors affect the growth of crops, some according to place, some by time. For each factor, there are information needs, much of it specific to the location of the crop. In particular, a farmer may be interested in knowing the spatial relationship between the physical phenomenon and the fields (e.g., whether the low moisture event region is adjacent, or inside, or outside a cultivated field). In a different context, consider now the Great Duck Island deployment [MCP$^+$02] for monitoring the nests in the petrel colony established there. The nests are situated in underground burrows, distributed in discrete patches around the island. Environmental conditions vary widely from patch to patch. This characterizes the need for a spatio-analytical perspective. For example, the biologists involved were interested in determining which environmental conditions yield an optimal microclimate for breeding, incubation, and hatching. Such microclimates are characterized by transient phenomena and by their interaction with permanent features of the physical environment.

WSNs are likely to be more prevalent as their cost-effectiveness improves [ASSC02a, HHKK04]. The spectrum of applications for WSNs spans multiple domains. In the environmental sciences, in particular, they are on the way to becoming an essential technology for monitoring the natural environment and for modelling the dynamic behaviour of transient physical phenomena over space. In most applications, a WSN acts as an entirely passive system, i.e., it helps in detecting events or in observing state of the environment, but is unable to affect it. In other applications, nodes may have actuators, with the help of which they can affect the environment. For example, in a flood monitoring application [Big09] based on water dams, the system can affect the environment automatically by opening water gates, or emitting alarms, when the water level in the dam exceeds a certain level. Examples of environmental WSN applications proposed in the literature include minimizing unintended impacts on wildlife habitats monitoring [HBC$^+$09, TPS$^+$05]; precision agriculture [PE08, AS09, BBB04, CMS03, OE08, WZW06, VTP$^+$08]; reducing the risk as well as providing expert support in a time of crisis (e.g., forest fire detection [JWZ$^+$09], flood detection [Big09], active volcano monitoring [WALW$^+$06], undersea surveillance applications [HYW$^+$06]; and pollution studies [TYIM05]). As can be seen, for many WSNs applications, the scenarios in which WSNs are best placed to act like macroscopes are those in which deployment takes place in sites with difficult access to scientists and cost constraints preclude expensive components or strategies. WSNs have, therefore, been advocated as a prime technology for building macroscopes for scientific studies. By *macroscope* is meant an intelligent, largely autonomous, instrument for scientific observation at very fine granularities and over large distributed areas.

The importance of identifying, tracking and reporting relationships between dynamic, transient spatial phenomena and application-specific geometries has been stressed in environmental monitoring applications. This dissertation argues that WSNs can be used effectively as spatial information systems, allowing management decisions about processes in the physical world to be made on the basis of sensed data. The main research challenge in this respect is how to support in-network spatial analysis in a cost-effective manner. At present, in-network spatial analysis in WSN is not catered for by a comprehensive, expressive, well-founded framework.

With a view to supporting this class of applications, this dissertation shows that such spatial analyses can indeed be efficiently implemented in WSNs using in-network processing techniques.

	Mica2	Imote 2
CPU Speed	8MHz	13-416MHz
RAM	4K	32MB
Program Memory	128K Flash (persistent)	256K SRAM (volatile)
Data Flash	512K	32MB
Radio Range	150m (outdoor)	30m

Table 1.1: Some characteristics of the sensor nodes in Figure 1.1

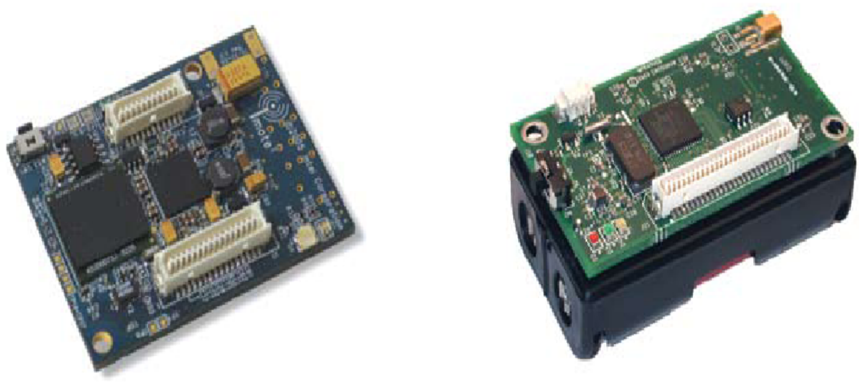


Figure 1.1: (a) Crossbow iMote2 mote [cro10a] (b) Crossbow Mica2 mote [cro10b]

The remainder of this introduction is structured, as follows. In Section 1.1, we describe the highly constrained distributed computing platform that WSNs give rise to. Section 1.2 explains the major challenges that arise, in an attempt to apply classical spatial algebra over WSNs. In Section 1.3, we summarize the context in which the research has been carried out. In Section 1.4, we describe the main motivations for conducting spatial analysis over WSNs. In Section 1.5, we describe the aims and objectives of the work that led to this dissertation. In Section 1.6, we present the resulting contributions. Finally, in Section 1.7, we outline the structure of the remainder of the dissertation.

1.1 WSNs as a Distributed Computing Environment

Sensor nodes can be considered small computers, but they are extremely resource-constrained in terms of communication, power, storage and computational resources. The size of the typical sensor node is on the order of a few centimeters. Crossbow Technologies [cro] was the first commercial supplier of WSN hardware. A number of other manufacturers and research institutions have also created their own hardware platforms. Although there are a number of different variants of sensor nodes (called *motes*), a typical mote would have a micro-controller, few kilobytes of RAM, a short range radio transceiver, sensing devices and several kilobytes of flash storage. These motes can be powered by one or two ordinary AA batteries. Figure 1.1 shows two different sensor motes which, at the time of writing, are among those motes available in the market. In Table 1.1, some characteristics for them are shown.

In WSNs, energy is valuable because it is scarce, sensor nodes only have finite energy reserves drawn from batteries whose replacement is not only expensive, but, for some applications, impossible as nodes can be deployed in human-unfriendly locations of difficult access [HBC$^+$09, TPS$^+$05]. In most, environmental monitoring applications, the WSN is, by and large, an isolated system with depletable resources. The effectiveness of WSNs is constrained, therefore, by their limited energy supplies. This causes their life time to be determined by their ability to use the available energy in an effective and frugal manner. As with other platforms, the cost of performing certain operations is more expensive when compared to others. In particular, in the case of sensor nodes, communication is more expensive than computation [GM04] and sensing operations. As described in [PK00], in a noise-free environment transmitting 1 Kb of data at a distance of 100 meters costs around 3 joules. Nevertheless, a general purpose processor with 100 MIPS/W capability consumes around the same amount of energy to execute around 3 million instructions. Therefore, reducing communication activity and increasing processing inside the network, because of the much lower cost of CPU cycles, boosts the longevity of the network.

Each of the sensor nodes typically has a short communication range and needs to work cooperatively in order for it to be effective over a large area. Assuming that they can be cost-effectively deployed in large numbers, this short-range communication constraint need not hinder their deployment, and may even be useful in cluttered environments where line-of-sight paths are short and prevent the formation of a long-range communication network. Hence, in most WSNs applications, in order to cover the desired monitoring region, multi-hop communication [ASSC02b, KR04, CSA04] is used, so that each sensor node may have to perform the additional function of forwarding/relaying the data that it receives from one to another of its neighbours thereby forming routing links, all the way back to the gateway. In addition, distributed sensing is also effective not only in cluttered areas but also in detecting events that cannot be effectively sensed from a long distance, e.g., temperature, humidity, and pressure. It follows from these observations that, as macroscopes, WSNs can cover large areas of irregular topography and with greater density of observation.

In most environmental monitoring applications, WSNs are used to sense and collect data that was impossible to collect in the past and to transmit it towards the base station for storage, where the required data analyses can be performed off-line [HM06, MGZ$^+$09, BBB04]. Some systems send specific requests to the nodes to fetch the data, others allow the nodes to send data autonomously, or else in response to the detection of some event of interest.

The technological characteristics of WSN alluded so far, viz. resource constrained sensor devices, distributed system complexity, and communication unreliability, raise extensive software development challenges. Software development costs remain high, as programming WSNs requires specialized knowledge, the scarcity of the resources puts a tight limit on code size, and debugging is cumbersome. Implementing a simple data collection application may require thousands of lines of code in an embedded programming language.

Transmitting every node sensed value [MCP$^+$02, Big09] to some destination that is external to the WSN for storage and off-line analysis may be prohibitively expensive and sometimes not possible, given the typical data collection rates and network sizes. In this approach, apart from network longevity, scalability is an issue as it will result in increased bandwidth requirements, raising the risks of packet loss due to collisions [BGS00, Sri]. In addition, the nodes that lie closer to the sink consume energy much faster, as they have to relay more packets towards the sink, than distant sensors. For supporting lower communication activity, it is therefore, preferable that, instead of raw sensor data, finer-grained information is returned by the network. In-network processing is a distributed technique for information processing that allows for the reduction of the network traffic in WSN, thus supporting network scalability, and network longevity. It has led to the development of generalized algorithms supporting different application scenarios (e.g., TinyDB [MFHH05]). Increasing in-network processing is useful in prolonging the lifetime, and hence improving the cost-effectiveness, of the deployment. In the in-network approach, energy efficiency arises from cooperation and a reduction in the need to transmit large amounts of data. One common strategy to achieve this is to perform data reduction (e.g., by computing aggregates) and filtering as early as possible in a data path. For the sake of increased network longevity, it is therefore crucial that tasks such as routing, sensing, localization, communication and others, are carried out using energy-efficient algorithms.

It follows from the considerations above that a WSN may be viewed as a distributed computing platform [BGS00], with each node being viewed as computational resource and not just a data collection and data transmission resource. Albeit limited, node resources such as processing power and memory can be used to execute application logic. WSN applications can therefore be considered to be fully-fledged distributed systems, since sensor nodes cooperate not only in the execution of application but in transmitting the results to the destination and in performing functions such as adapting to changing network topology, and ensuring the longevity of the network. Moreover, each sensor node acts as a separate data source providing data by means of its sensing capabilities or pulling it out of its flash memory.

Note, however, that sensor nodes constitute a distributed environment and hence, there is no central clock to regulate the activities of the network. Time synchronization is, therefore, a critical requirement for in-network processing, as is accurate time stamping of sensor events and, possibly of processing and communication events as well. This is a well studied problem [SKPM06, EE01], and it is beyond the scope of the dissertation to address such issues. Components that provide such functionality are assumed to exist in support of the distributed algorithms described later on.

More specifically, WSNs have been the focus of research in which the network is viewed as a database against which queries can be executed. This approach, with its reliance on declarative specifications of the data to be retrieved, can be seen as response to the software engineering challenges alluded to above and to the fact that too many existing deployments have tended to be application-specific. Gehrke et al. [YG02] and Madden et al. [MFHH05] propose that WSNs can be programmed with considerably less effort by the use of the database paradigm. This has given rise to *Sensor Network Query Processors* (SNQPs) implementing in-network declarative query processing over WSNs, examples of which include TinyDB [MFHH05], Cougar [YG02], and SNEE [GBJ^{+}09]. Declarative queries allow users to specify what data they want from a WSNs without needing to know such details as how to contact the relevant sensing devices on sensor nodes, how to deploy application logic, how to manage its execution and how to transmit results back to the user. However, current WSNs query processors are not capable of performing spatial analysis. At present, in-network spatial analysis in WSN is not catered for by a comprehensive, expressive, well-founded framework.

1.2 Core Challenges for Performing Distributed Spatial Analysis Over WSNs

There are major differences between carrying spatial analysis over WSN and classical spatial database systems. The design of distributed spatial analysis algorithms that run correctly and efficiently over WSNs poses several challenges, some of which are summarized in the remainder of this sub-section.

1.2.1 Support For Induced Geometries

In a classical centralized spatial database system (such as Tripod [GFP$^+$01], which makes use of ROSE [GS93, Sch97] algebra), spatial analysis is usually performed over *asserted* and *derived* geometries. The reader is referred to Section 2.1.1 for a description of ROSE algebra. By *asserted* geometries are meant permanent geometries that are known in advance. By *derived* geometries are meant geometries that are computed based on existing *asserted* geometries and are output by the spatial operations available for that purpose. Information about these geometries is held centrally and is rarely updated.

WSN applications are usually deployed for real world monitoring, in contexts where scientists are interested in the shape and size of an event as it occurs in the sensing scope of the nodes. The accurate characterization of the geometry of transient, physical phenomena as they take place in the sensing scope of a deployed WSN is referred to in this dissertation as *induced* geometries. Therefore, in the case of WSNs, spatial analysis needs to be carried on three types of geometries, viz., *asserted*, *induced*, and *derived* geometries. Note, furthermore, that *derived* geometries may be computed based on any combination of existing *induced*, *asserted* or *derived* geometries. The reader is referred to Section 3.2.2 for detailed description of these geometries.

1.2.2 Supporting Continuous Queries

Classical spatial DBMSs mostly operate on non-streamed data and only support one-off queries. In contrast, WSNs must support continuous queries, which are posted once and evaluated many times, result tuples being generated continuously for possibly long periods of time [SN05]. WSNs are, by definition, connected to the physical world and sensed data streams represent properties of dynamic, evolving real world events (as opposed to stored data in the case of classical DBMSs). Traditional spatial DBMSs only support queries that are posted once and produce a complete result set in one single return event.

Continuous monitoring of an induced geometry makes it possible to track the spatial evolution of the underlying spatial phenomenon. An event of interest can be characterized by an *event-defining predicate* (e.g., `humidity>98 and temperature<10`). If so, then the notion of an *event geometry* is definable in terms of the location in space of those sensor nodes that satisfy the event predicate. More specifically, changes in the measurements of physical quantities obtained by a group of sensor nodes can be used to characterize the life cycle of an event of interest by the way its geometry changes. This is so because, over time, the measurements obtained by each sensor node will vary depending on the distance at which that node lies from the physical phenomenon under observation. At different points in time, such a predicate will change in its truth value at different nodes depending on their location with respect to the event defined by the predicate. This allows the event geometry to be induced.

1.2.3 Distributed Data

In the case of the classical approach, the assumption is that a computer is used to store the geometries. WSNs are distributed platform, therefore, each node is only aware of that part of induced geometries lying in their sensing range. Because of this kind of resource constraint, complete information regarding induced, asserted and derived geometries is distributed throughout the WSN. No node can assume to have complete information about the geometries it is part of.

The ensuing challenges are highly nontrivial, particularly so in the case of induced geometries. In such scenarios, a task related to the detection of an induced geometry is re-evaluated with some periodicity and each node independently updates the local information that defines the induced geometries it is a member of. Thus, unless it cooperates with other nodes, each node is, in principle, only aware of its own membership status. The inherent scarcity of resources and the nature of underlying platform, where execution is distributed and carried out periodically over sensed data streams, give rise to non-trivial challenges.

Furthermore, the resolution or scale of the spatial data may vary, geometries may have different spatial dimensions, and spatial types (e.g., **points**, **lines**, or **regions**). These several forms of diversity give rise to challenges on how to integrate and to keep them consistent in order to provide correct answers for spatial analysis tasks.

1.2.4 Energy Efficiency

In the case of WSNs, for reasons of energy efficiency and network longevity, the algorithms should run in a distributed manner inside the network. This requirement for in-network processing arises because the cost of communication dominates energy consumption and nodes are energy-bound. The need to reduce communication often precludes sending all sensed values back from the nodes to the base station as well as any scheme that often requires the exchange of messages in a non-localized manner, i.e., beyond the one-hop neighbourhood of a node. Concrete algorithms implementing spatial operations as well as algorithms for task dissemination, routing, aggregation, etc., must be localized.

1.2.5 Aggregation of Information

Recall that information about a geometry lies in distributed fashion inside the WSN and that each node is only aware of portion of geometry. This implies that the computation of spatial operations requires aggregating information from all the nodes that belong to the operands of the operators involved. An operand may comprise a single-element geometry or a multi-element geometry with or without holes.

Furthermore, induced geometries are dynamic, which implies that, their shape and size can not be known in advance. This requires the design of efficient in-network hierarchial aggregation scheme that allows for aggregation of information, from single and multi-element geometries with or without holes to compute the final result. Such scheme must be efficient enough to reduce not only the number of messages but also the amount of information they carry.

As the nodes are resource constrained, therefore, for detailed spatial analysis instead of evaluating many simple tasks separately, the user should be able to express and evaluate complex spatial tasks (examples are given in Section 1.4) in an energy-efficient manner.

1.2.6 Network Dynamics

Another fundamental challenge in WSNs is to cope with network layer dynamics. Among the most important dynamic events are those that cause the network topology to change. There are several factors that can cause changes in network topology between evaluation episodes including: (i) node failure at various locations; (ii) change in membership status for a specific geometry resulting in increase or decrease in the number of nodes belonging to that geometry; (iii) packet collision, and (iv) loss of communication.

1.2.7 Synchronization in Complex Task Evaluation

In-network processing of spatial tasks requires nodes to perform local processing to compute operation state for each operation in the task. An operation state identifies whether the node satisfies the primitive success criteria for a spatial operator. The evaluation of some spatial operators may only require information that is available within the node itself, whereas for others the node might require information from neighbouring nodes that are also members of operand geometries to compute its operation state. For timely coordination, participating entities must start and finish the processing of each spatial operator in the complex task at the appropriate time in order to avoid delays in response and in achieving accuracy. As each node is acting as a unit in a distributed computing platform, during the evaluation of a task, at each point in time, there may be situations where some of the nodes satisfy the participation requirements of one or more operators in the task, and some satisfy for other operators, and others does not satisfy for any of the operator.

1.3 Technical Context To In-network WSN Spatial Analysis

There are several factors that motivate the investigation of spatial analysis over WSNs, the combination of which provides an opportune and timely context for research on this area:

- SNQPs, have demonstrated that in-network processing is an effective and efficient means of interacting with a WSNs in data collection tasks. Thus, spatial analysis over WSNs can build upon established distributed query processing techniques, but, here, emphasis is on the spatial aspects of the data, which are not adequately addressed in existing SNQPs [MFHH05, YG02, GBJ$^+$09].
- The emergence of event and edge detection techniques as an effective approach to detect events in a distributed manner using in-network processing techniques [CG03, JF08]. The

reader is referred to Section 3.4 for description.

- The existence of research [FZWN08, JW08, WD06] on providing a computational model for WSNs to detect topological change (i.e., hole formation/hole loss, self-split/self merge, and split/merge) in dynamic regions based on local low-level snapshots of spatio-temporal data. This work has tended to be comparatively narrow in scope in that it confines itself to detecting topological change.

- The existence of a spatial algebra based on finite resolution computational geometry represented with a discrete grid, viz., the *RObust Spatial Extension* (ROSE) algebra [Sch97]. The ROSE algebra has several desirable characteristics: it rigorously defines a comprehensive set of Boolean-valued (e.g., `meets`, `adjacent`, `vertex_inside`, etc.) and spatial-valued operations (e.g., `plus`, `intersection`, etc.), and it supports complex geometries such as regions containing holes, and multi-element ones. However, [Sch97] only presents centralized algorithm over stored data for one-off execution.

With a view to supporting environmental applications that depend on monitoring evolving phenomena, this dissertation defines a framework for representing geometries over WSNs, thereby allowing the representation of *asserted*, *induced* and *derived* geometries. The dissertation then shows how an algebra can be defined over this spatial framework whose operations enable the expression of sophisticated spatial analyses over WSNs. In particular, this dissertation shows how the algebra can be used to characterize complex and expressive topological relationships between spatial entities and spatial phenomena that, due to their dynamic, evolving nature, cannot be represented a priori. If it can be efficiently implemented in WSNs, such a characterization is very helpful in the detailed analysis of such phenomena.

The focus of the research reported here was on developing algorithms for efficient in-network evaluation of complex tasks that are expressions in the algebra mentioned above, over WSNs and over complex, dynamic phenomena. This dissertation describes distributed implementations of spatial-algebraic operations over the geometries represented by that framework, thereby enabling: new geometries to be derived from induced, derived and asserted ones, and relationships to be checked between spatially-referenced entities.

1.4 Applicability of Spatial Analyses in WSNs

The application level motivation for the research described in this dissertation stems primarily from applications where important decisions hinge on the detection of an *event of interest.* In the last decade, for example, wireless technologies have been increasingly applied in precision agriculture. Precision agriculture is a developing discipline aiming to enhance farming efficiency [KK03]. We use as a running motivating example an application of WSNs in agriculture.

WSNs have been used in precision viticulture for suggesting appropriate management practices to increase productivity and the quality of the crops. One real-world example in which WSNs are being deployed for precision viticulture is at Camalie Vineyards [Ulr08]. Monitoring the moisture in the soil, soil pH, sun exposure and temperature is crucial to improving crop yield and quality. WSNs support the monitoring of these physical quantities. The availability

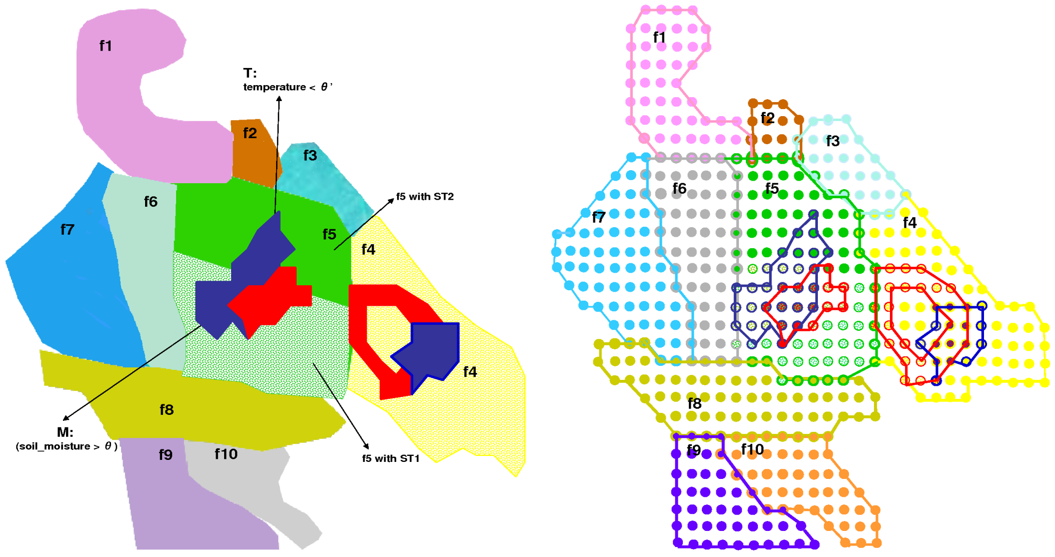


Figure 1.2: (a) Fields (f1-f10) with induced geometries in a vineyard (b) Example WSN over (a) showing approximate geometry membership

of real-time information makes it possible to promptly target irrigation when soil in one area dries up, to remove leaves and expose grapes to more sunshine, to change the schedule for using pesticides or to make arrangements to avoid the risk of frost damage to vines. For controlling the spread of disease, pesticides need to be applied if certain temperature and humidity conditions are met and these may be periodic. The application of pesticides can be delayed or stopped if high temperature and low moisture are detected, thereby avoiding the risk of damage to vine quality. It can be seen that much of the information relates to the location, as the factors affecting crop quality are inherently spatial in nature. Therefore, the spatial aspect of the data is significant in contributing to the growth and quality of vines.

As WSNs provide an interface to the physical world, our main motivation is to contribute to the goal of making it act as an intelligent device (i.e., a macroscope). Instead of retrieving raw data for farmers, we aim to make the WSN respond to the information requests by the farmers that require exploring the relationships between spatially-referenced entities, and to derive representations grounded on such relationships (e.g., to compute a new derived geometry as the intersection of the geometries that characterize the low moisture region). The power of this intelligent tool derives from the integration of spatial data with descriptive data pertaining to spatially distributed phenomena. Our aim is to contribute to the spatial analysis of such interactions using in-network processing techniques [MFHH05, YG02, GBJ$^+$09].

We have used the underlying *asserted* geometries ($f1$-$f10$) in the deployment depicted in Fig. 1.2 as the basis for our examples and experiments [1]. The starting point for our motivational scenario is the observation that mildew and other bacteria tend to spread under certain temperature and humidity conditions. When this happens, chemicals need to be applied. As a result, it is important to monitor the temperature and soil moisture with appropriate temporal

[1]More information is available at `http://camalie.com/WirelessSensing/WirelessSensors.htm [12 June 2009]`. In particular, see `http://camalie.com/CamalieGIS/Naked/default.asp [12 June 2009]` for the geometries we use as our basis.

and spatial precision in order to decide when and where to apply chemicals. More formally, given thresholds θ and θ', call `M` the *induced* geometry where soil moisture is above θ and `T` the *induced* geometry where temperature is below θ'. The sensor nodes whose measurements satisfy each of these predicates characterize the two *induced* geometries. Figure 1.2 shows *asserted* geometries ($f1$-$f10$) and *induced* geometries `M` and `T`. Both geometries M and T have two elements. Both elements of geometry T are without holes, whereas one of the element of geometry M has hole and other is without hole. Our motivation is to enable WSNs that allow a farmer to pose a task whose evaluation determines whether there is a need to spray a field, say `f5`. The task is a spatial-analytic one and can be expressed in terms of the `intersects` topological relationship as follows:

```
((M Intersects f5) AND
   ((T Intersects f5) AND (M Intersects T)))
```

If this expression returns **true**, the farmer needs to take action. In order to derive a geometry, where spray is needed a farmer can pose the task as follows:

```
((M Intersection f5) Intersection
   (M Intersection T))
```

In terms of our motivating example, we note that water infiltration varies over space and time and hence influences soil moisture levels differently. The farmer can pose a task as follows to compute the relationship of the *induced* geometry to several others:

```
(((M BorderInCommon f6) OR (M AreaInside f6)) AND (M Intersects f5))
```

As mentioned, the quality of a crop is shaped by the physical conditions where it is growing (i.e., location, topography, soil, etc.) and the climatic environment. Wine makers achieve greater control over the product by defining sub-blocks within the vineyard. Batch selection can then be decided based upon suitable flavors, spatial locality, type of soil and climatic factors.

Assume that field `f5` comprises types of soils ST_1 and ST_2. Call SA the area where soil is of type ST_1 and SB the area where soil is of type ST_2. If so, a farmer can derive a new geometry comprising the area of soil type ST_1 in field $f5$ where neither soil moisture nor temperature are above the target thresholds by posing the following task.

```
((F5 Intersection SA) Minus (M Union T))
```

From the discussion in this section, it can be seen that there is a need for contributions that enable in-network distributed spatial analyses of physical phenomena using WSNs.

1.5 Aim: Distributed In-Network Spatial Analysis Over WSNs

In this section, we describe the main research aim underlying the contributions reported in this dissertation in terms of its component objectives. The main aim is to contribute to the enabling of in-network distributed spatial analyses of spatial phenomena that can be sensed using WSNs. In order to achieve that aim, the following objectives have been pursued:

1. To define a spatial framework for representing geometries over WSNs.
2. To identify distributed algorithms for characterizing induced geometries in the framework in (1).
3. To define a spatial algebra over the geometries representable by the framework in (1) and inducible by the algorithms in (2).
4. To design and implement distributed in-network algorithms for the operations in the spatial algebra in (3) over the geometries that can be represented over the framework, thereby enabling new geometries to be derived from induced and asserted ones, as well as the computation of relationships between geometries, with the optimization goal of preserving energy and obtaining short response times.
5. To design and implement a task processing system for the evaluation of complex algebraic expressions using the algorithms in (4).
6. To evaluate the effectiveness and efficiency of the concrete algorithms in (5) by means of empirical experiments over simulated deployments.

1.6 Contributions

In this section, we describe the contributions of the research. The contributions described below are constrained to WSNs where every node is tethered at one location in physical space.

1. This dissertation presents a framework for representing geometries in WSNs that can be induced, asserted and derived. These geometries can be single or multi-element and can have one or more disjoint holes inside it.
2. This dissertation presents a detailed characterization of a spatial algebra closely inspired, in its scope and structure, by the Schneider-Guting ROSE algebra over the geometries representable by the framework in (1). The dissertation also contributes extended definitions for topological operations that take into account cases where the geometries may comprise unit **regions**, which were overlooked by Schneider and Guting.
3. This dissertation presents distributed in-network algorithms for the operations in the spatial algebra over the representable geometries, thereby enabling (i) new geometries to be derived from induced and asserted ones and (ii) topological relationships between geometries to be identified. The algorithms for these spatial operations are divided into logically-cohesive components. The algorithms are specifically tailored for power-efficient in-network execution, with a focus on minimizing unnecessary communication and reducing the size of information to be communicated. The distributed algorithms are, for the most part, localized (i.e., communication is restricted to one-hop neighbourhood), and hence have desirable complexity in terms of messages complexity as well as of bit complexity, response time, and energy consumption.

This dissertation shows that the algorithms for topological operations maps the problem of distributed computation of complex algebraic expressions stemming from (3) that involve multi-element geometries to the problem of first computing a node-level task state and then aggregating that at two levels using a new bit-string-based approach. Computation of aggregates, such as count, sum, average, minimum, and maximum, is a well studied problem and can be solved in a distributed manner. The dissertation studies two aggregation approaches for handling this distributed aggregation problem and contributes modifications to these approaches that yield improvements in performance.

4. The dissertation shows that a general algorithmic strategy for the evaluation of complex algebraic expressions can be used that breaks down into logically-cohesive components. A task processing system has been developed that allows the users to evaluate task on nodes. Each node is equipped with the task evaluation components allowing it to participate in task dissemination, to contribute in the distributed evaluation of a task, and to participate in the aggregation and routing of results to the user.

5. This dissertation contributes an empirical performance study of the system stemming from 4. We evaluate our algorithms with detailed simulations using the TinyOS [LMG+04] simulators TOSSIM [LLWC03] and PowerTOSSIM [SHrC+04]. Our experimental results provide evidence that the algorithms scale well in terms of response time, message complexity, bit complexity and energy consumption under varying conditions.

1.7 Structure of the Dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 briefly outlines the background work on the ROSE algebra. It also describes the 9-intersection model based on point-set topology, which is a more restricted algebraic framework than the one adopted in the rest of this dissertation.

- Chapter 3 discusses the research dealing with spatio-temporal query processing and on event tracking and monitoring. It then defines a framework for representing *induced* geometries as well as *asserted* geometries and *derived* geometries that advances on the work in Chapter 2.

- Chapter 4 describes the spatial algebra over the geometries representable by the framework described in Chapter 3.

- Chapter 5 discusses the research dealing with spatial analysis over WSNs. It then describes an algorithmic strategy for in-network distributed spatial analysis over WSNs that addresses the challenges identified in Chapter 2. This chapter also presents distributed in-network algorithms for the operations in the spatial algebra described in Chapter 4 using the algorithmic strategy described in Chapter 5.

- Chapter 6 presents empirical performance study of the algorithms in Chapter 5.

- Chapter 7 summarizes the work presented in this dissertation and suggests some future directions that could be pursued to advance the WSNs spatial analysis research in the wake of the contributions reported here.

Chapter 2

Background

At the time of writing, there is no comprehensive approach available to perform in-network spatial analysis over WSN. Note that, therefore, this chapter does not describe potentially competing research. Spatial analysis over WSN clearly raises research issues at several levels. It is crucial, therefore, to survey the state of art on several areas of the literature, as they may contribute insights, methods and techniques for building a complete framework for evaluating distributed spatial operations in WSNs. This chapter, is therefore, a survey structured by the broad areas that comprise the technical context underpinning the research contributions reported in the remainder of this dissertation.

Section 2.1 briefly outlines the background work on centralized spatial algebra, based on finite resolution computational geometry represented with a discrete grid. By *centralized* is meant that the operations of such an algebra are applied to data stored on a single, central server. Such centralized spatial algebra cannot be implemented without modification over WSNs. The goal of doing so presents significant challenges that were discussed in detail in Section 1.2. The aim of discussing centralized, classical spatial algebra is to highlight the similarities and challenges in supporting a spatial algebra and representing spatial data in WSNs, and, more specifically, to investigate whether an existing classical spatial algebra could be redesigned, redefined and reimplemented in distributed form, thereby enabling spatial analysis over WSNs.

Section 2.2 introduces concepts and terminology related to designing a precise deployment model in WSN-based environmental monitoring applications. Section 2.3 briefly outlines the background work on aggregation approaches over WSN. Section 2.4 reviews some of the background literature on routing, with emphasis on geographical routing over WSN. The motivation behind the description of routing and aggregation approaches is to build the technical context in so far as these techniques will be required for distributed evaluation of spatial tasks. Section 2.5 summarizes the chapter.

2.1 The (Centralized) Spatial Algebra

This section briefly outlines the background work on a spatial algebra based on finite resolution computational geometry represented with a discrete grid, viz. the ROSE algebra. The main motivation behind the description of ROSE algebra is that this dissertation presents a detailed characterization of a spatial algebra in Chapter 4 closely inspired, in its scope and structure, by the Schneider-Guting ROSE algebra.

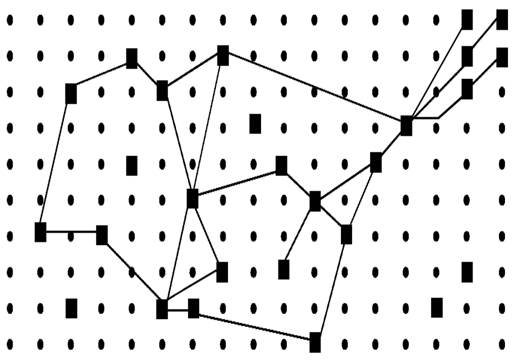
2.1.1 ROSE Algebra

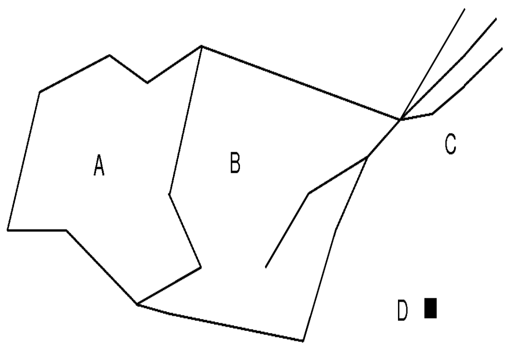
The ROSE algebra uses the notion of a *realm* for spatial modelling in order to overcome the problem of limited precision support provided by computers. With this aim, it replaces Euclidean space with a discrete geometric basis. The ROSE algebra is based on finite resolution computational geometry where spatial values consist of finite sets of points and non-intersecting line segments defined over a discrete point grid called a *realm* (see Figure 2.1). A *realm* consists of a finite set of points, called *R_points*, and a finite set of segments, called *R_segments*. It provides a discrete, numerically robust, and consistent geometric basis on which spatial data types can be defined. *R_points* are the end points of *R_segments*. It is assumed that *R_points* have coordinates in the grid and that no *R_point* lies within a *R_segment*, as shown in Figure 2.1. All intersections between segments are assumed to be pre-calculated at the time the representations enter the database, or when updates are made. Therefore, any two distinct segments neither properly intersect nor overlap. The use of a *realm* guarantees that all spatial operations are error-bounded and take as input, and produce as output, intersection-free spatial values. Thus, interactions with a realm-based database benefits from geometric consistency. Much of the material in this sub-section is summarized from Schneider et al. [GS93, Sch97].

All spatial objects processed and produced by the ROSE operations are *realm*-based, i.e., they are defined over a discrete basis. For obtaining a discrete geometric basis, it is necessary to avoid the intersection of line segments at a point not lying on the underlying grid. The topology-preserving solution to this problem proposed by Greene and Yao uses a redrawing method [GY86], and that proposed by Guibas and Marimont uses snap rounding method [GM95]. As a result, the spatial algebra and the geometric algorithms, built on top of *realm* are protected from problems of numerical discrepancy and topological incorrectness.

Application-specific geometries can be defined using the ROSE data types, which are collections of **points**, **lines**, or **regions** representing geometric entities. A **points** value can be used to denote 0-dimensional spatial entities (a value of which could denote, e.g., a well)). A **lines** value can be used to denote 1-dimensional spatial entities (a value of which could denote, e.g., a river, or a pipeline). A **regions** value can be used to denote 2-dimensional spatial entities (a value of which could denote, e.g., a building, or a cultivated field). The values of these data types can be defined in terms of points and line segments present in a *realm*. Figure 2.1 shows some spatial values defined over the *realm* in Figure 2.1(a). In Figure 2.1(b), A and B represents **region** values, C is a value of type **line** and D is a **point** value.

Over such data types, a comprehensive set of algebraic operations is defined. There are four classes of operations: binary spatial predicates expressing topological relationships, operations returning spatial objects, operations returning numbers, operations on sets of objects. Appendix A discusses the spatial data types and the two types of operation in the ROSE algebra that are of greatest relevance in this dissertation, viz., spatial predicates and spatial-valued operations. The motivation behind this is two-fold: firstly, to explain how the operations over the geometries are computed in a discrete manner; and secondly, to lay the foundations for Chapter 3.2 and Chapter 4 where we show how these definitions can be mapped to sensor space and why some of these definitions need modifications.


(a) Realm: A finite set of points and line segments over a discrete domain



(b) Application-specific geometries defined over (a)

Figure 2.1: Realm and application-specific geometries defined over *R_points* and *R_segments* adapted from [Sch97]

Recall form Chapter 1, in WSN each node has a location, which gives spatial properties to data. These nodes have limited communication and sensing capabilities, therefore, deployed in a finite large number to be effective over large area. In WSN, any pair of nodes in a WSN can only communicate with each other, if there is communication link between them. Similarly, sensor nodes can only sense an event that lies in their sensing range. WSNs deployment strategy, therefore, provides discrete, and consistent geometric basis on which spatial data types can be defined. WSN comprises finite-set of points and line segments. Each node can be considered representing point, whereas communication links between any pair of neighboring nodes can be considered a line segment. It is therefore, possible just like ROSE algebra to define the geometries based on these points and line segments. The reader is referred to Chapter 3, for the description of how a WSN can give rise to a discrete grid over which it is possible to define spatial types and values that follow the ROSE-algebraic approach and to Chapter 4, for the description of spatial algebra over the framework defined in Chapter 3.

2.1.2 Algorithmic Strategy

Schneider and Güting provided efficient centralized algorithms for the ROSE-algebraic operations [Sch97]. Algorithms for these operators make use of the plan-sweep technique. Plan-sweep technique allows for the transformation of geometric set problems into a form which are easier than original to resolve. For implementation AVL tree structure is used. Lines and regions values are stored as the ordered sequences of segments for efficiency. For the evaluation of most of the operators, parallel traversal through the points or segments of the objects are made. These algorithms are not usable for in-network processing in WSNs, because, in this case, spatial data and execution of operations is distributed and carried out periodically over sensed data streams.

2.2 Background to Sensor Network Deployment

In WSN-based environmental monitoring applications, designing a precise deployment is challenging because accurate event detection depends on a number of factors including the type and quality of the nodes, and the nature of the terrain. The type of event to be detected varies from application to application. In this dissertation, we are only concerned with events related to physical phenomena. In such applications, a sensor node can only detect a part of the event, and only if the phenomena lies in sensing range. For dynamic, transient phenomena, the size and shape of the event geometry varies over time and space. If a large number of nodes can be deployed, finely-grained event detection is a possibility, otherwise, one has to compromise on the precise shape and size of the event geometry.

Although the relatively low cost of the sensor nodes allows for the deployment of these nodes in large number, in many applications the purchasing cost is not the critical factor compared to the deployment cost and the need to collect the right data at the right spatial grain. Therefore, different applications require different geographic coverage with varying or uniform nodes density over the field under study.

The deployment strategies are broadly divided into regular grid deployments and random deployments. In regular grid deployments, the sensors are placed deterministically with some regular geometric topology along grid points. This strategy is mostly used in application scenarios where the deployer has control over node placement, i.e., where access to the deployment site is not a concern and it is safe for humans to place sensors manually. Because of the fact, that each node has a communication radius r^c and a sensing radius r^s, in most applications nodes are usually deployed at a distance r^s apart along the regular grid points ensuring coverage of the area as well as node connectivity. In the case of a regular grid deployment, the granularity of the grid is, therefore, a function of the sensing range, the radio range and whatever fine-grained event-detection resolution that can be afforded. The smallest rectilinear distance between two nodes defines the absolute distance between any two adjacent abscissae and between any two adjacent ordinates. Random deployments allow for the location of the nodes not to be planned a priori: the sensors may be air-dropped, scattered using a vehicle or any comparable means [EG02].

2.3 Aggregation Approaches

In-network aggregation is a well studied problem in WSNs. Aggregation approaches are distributed techniques for reducing the amount of data and, thus, the total number of packets that need to be transmitted from the sensor nodes to the sink node. The idea is to remove data redundancy and combine the data coming from different sources by applying simple aggregation operators (e.g. sum, average, min, max, count) to more complicated data aggregation operators (e.g., median, Wavelet Histogram [SBAS04]) outputting thereby more compact representation of the data. These aggregation functions can be applied to data produced during the same sample period or over a time span comprising of few sample intervals. In-network aggregation has been found to prolong the life time of the network and is, therefore, a crucial technique for energy-constrained WSNs [MFHH02, IEGH02].

The algorithms available for this problem can be broadly classified into tree-based and gossip-based.

2.3.1 Tree-Based Aggregation Approaches

Tree-based approaches, like those proposed by Madden et al. [MFHH02] and Zhao et al. [ZGE03], compute the exact value of the aggregation. Tree-based algorithms tend to be energy-efficient but are not resilient in the presence of topology change. These issues can sometimes be addressed by approaches that maintain the tree after it has been constructed [MFHH02, HYS04].

Tiny Aggregation (TAG) [MFHH02] is an aggregation service for WSN nodes by TinyDB. It aims to reduce the communication overhead for computing an aggregate value. The TAG approach focuses on sending a single message per sample period from the child node to its parent regardless of the depth at which the child node lies and irrespective of the aggregate function. When the parent receives a value it aggregates that value to the one it itself has sensed, and sends the value to its parent in turn. One of the features of TAG is to tolerate disconnections and loss as it is designed to sit on top of a network topology. By dividing the time into sample periods, the processor and the radio can lie in deep sleep modes for most of the time, thereby resulting in low power consumption during long idle periods. In-network aggregation has been shown to be energy-efficient. For example, in-network aggregation of the SUM function allows each intermediate node to forward a single message containing the sum of the sensor readings of all upstream nodes, instead of sending every measurement from every node that lies upstream from it.

For tree-based aggregation schemes, if the root node is not predetermined, a root node must be elected, either randomly or through a distributed leader election algorithm like the one proposed by Dulman et al. [DHS02]. Once the root node has been elected, the tree creation process is started by the root node. Once the tree is set up, it can be used for aggregation purposes.

In [HL05, LRS02], the authors compute aggregation topologies by taking into account the residual energy of each node. In addition, there are other efficient tree-based in-network aggregation approaches. *Temporal Coherency Aware In-Network Aggregation* (TiNA) [SBLC04] exploits the temporal correlation in a sequence of sensor readings for performing energy efficient in-network aggregation. It allows intermediate nodes to send data up the hierarchy only when there is a significant change in the value of collected readings over time. It allows the suppression of readings as long as the expected quality of data defined by the user or application is not affected. The *Semantic/Spatial correlation-aware tree* (SCT) approach [ZVPS08] supports application scenarios where nodes are densely deployed and in which, as a result, the readings reported may be spatially correlated. To build an aggregation structure, the network is divided into partitions and then sub-divided into sectors. A node is selected as a leader for each sector based on its residual energy and location. Nodes in a sector report to the sector leader. The leader is responsible for aggregating the data from sources. A spanning tree is built on top of these nodes to form a structure for aggregation. *Dynamic Query-Tree Energy Balancing* (DQEB) [HYS04] allows dynamic modification to be made to the tree structure once it has been constructed with a view to balancing the energy left at nodes. The *Load Balanced Tree*

Protocol (LBTP) allows the gathering of periodical data [CTC06] by constructing a balanced tree in which all non-leaf nodes have a similar number of children. The tree structure is updated when the energy at a non-leaf node falls below a certain threshold. Heuristic algorithms for real-time data aggregation [CLJ06] allow for real-time data aggregation by supporting the transmission of packets with a specified time requirement. Cheng et al. suggested three different heuristics-based algorithms for constructing minimum spanning trees that satisfy hop and degree constraints.

2.3.2 Gossip-Based Aggregation Approaches

Gossip-based algorithms (e.g., Boyd et al. [BGA$^+$05], Chen et al. [CPX05] and Kempe et al. [KDG03]) are distributed, localized algorithms that do not require any pre-computed routing structure and are thus resilient to topology changes at the cost of less energy-efficiency. These algorithms can be used to compute aggregate functions such as sum, average, maximum, and minimum within a given error range.

To the best of the author's knowledge, the first characterization of randomized gossip-based algorithms for computing aggregates in completely decentralized manner was reported in [KDG03]. Push-sum algorithm allows a node to communicate with only one of the randomly selected node in each round. Each node maintains two attributes (i.e., sum and weight). In the first round, nodes declare their *sum* attribute value to their own sensed value and *weight* to 01. For estimating average, in each round each node sends half of its *sum* and *weight* values to its randomly selected node; which adds them to its own halved values. Finally, *sum/weight* is the estimate of the average in the last round.

Boyd et al. [BGA$^+$05], like Kempe et al. [KDG03] also uses the asynchronous time model. For estimating average, in each round a node selects a neighbouring node with some probability, to which it sends its value. Upon receiving value from neighbour, both nodes set their values equal to the average of their current and received value. *Distributed Randomized Grouping* (DRG) [CPX05] takes advantage of the broadcast nature of transmissions in WSNs to compute the aggregation operator *average*. DRG is more efficient than the gossip-based algorithms [KDG03, BGA$^+$05]. In DRG, each node can be in one of three states: *idle*, *group leader* or *group member*. In each round, some nodes are elected as *group leader* nodes with some probability p_g and the other nodes take on the role of *group member* nodes. A *group leader* broadcasts a request message and waits for its neighbouring nodes in *idle* state to join the group. An *idle* node that receives a *group leader* request sends a reply including its data value to the group leader and changes its state to *group member*. A *group member* does not respond to any other *group leader* requests and waits for a reply from its *group leader*. The *group leader* computes the average by first adding values received from members, then dividing the result by the number of group members, then broadcasting the computed value in its reply message and, finally, changing its state to *idle*. Group members that receive this broadcast overwrites their own value, and changes their state to *idle*.

Chen et al. [CPX05] have shown that the upper bound for convergence of this process depends on p_g, the network topology, the variance of initial values and the accuracy requirement for the aggregate value. In addition, fastest convergence is achieved when $p_g = 1/x$, where x is the average number of two-hop neighbours around the group leader. However, a distributed algorithm to terminate the evaluation of algorithm is not given. The message complexity of each iteration depends on the number of rounds for converging to the aggregate function. Chen et al. [CPX05] have given an upper bound on the expected number of rounds needed by network nodes to converge on the average value of $O(1/\gamma \, log(\phi_0/\epsilon^2))$ where γ depends on the grouping probability p_g and network topology on the grand variance of initial values ϕ_0 and the error tolerance ϵ of the aggregate average.

Section 2.3 has described the two broad categories of aggregation protocols. The motivation behind the description is to build the technical context, as the information from nodes in the network involved in the spatial task evaluation will need to be aggregated in order to compute the fine-grained final result inside the network.

2.4 Routing Approaches

Routing protocols allow for determining the paths taken by the messages between nodes. Routing protocols [AY05b] can be broadly divided into four categories (1) data-centric, (2) hierarchical, (3) location-based, and (4) quality-of-service (QoS) aware. Data-centric protocols [HKB99, CIE00, BE02] are often query-based and may perform in-network aggregation of data, as described in Section 2.3 above. These protocols support finding routes to intermediate nodes that allow in-network aggregation of data from multiple sources on their way to sink. Directed diffusion [CIE00] is a data-centric routing protocol, in which all communication is for named data and all nodes are application-aware. Query is disseminated inside the network as an interest for named data. Which results in setting up the gradients within the network to pull events. These features supports energy saving by enabling efficient path selection and en route data aggregation.

Hierarchical routing protocols [HHT02, YF04], divide the network into small groups called clusters, where each cluster leader is responsible for aggregation and reduction of information transmitted to it by the nodes belonging to the cluster.

QoS-aware routing protocols strive to meet certain QoS requirements [HF08, AY03, CT04, YCL$^+$01, AY05a, ZGFL08, ZRLM06] such as: delivery time, energy consumption in the nodes, and network life time.

In location-based (also called geographical) protocols [JJPSW$^+$09], each node makes a forwarding decision, based on its location information, that of its neighbours, and that of the destination. One of the advantages of geographic routing is that the routing overhead is minimized. Such routing schemes operate on locally available position information to make packet forwarding decisions [ZLS07]. Other features includes scalability, statelessness and low maintenance overhead [ZRLM06].

Yu et al. [YGE01], Alexandru et al.[CNS04] and Lian et al. [LCN$^+$05] have used geographic information for dissemination of queries to appropriate regions. In [LCN$^+$05], the authors suggested using a flooding approach for forwarding the query towards the target region and its dissemination inside the region. A leader is selected among the nodes in the target region who is held responsible for performing aggregation and forwarding the result to the sink through one of the available paths. In [CNS04], the authors use a greedy forwarding approach [Sto02] for forwarding the query towards the leader located in the center of the target area. The query is then broadcast inside the target region. Flooding of the query inside the target region helps in the construction of the routing tree used by the nodes to transmit their result to the leader inside the target region. In [YGE01], the authors propose a *Geographic and Energy-Aware Routing* (GEAR) protocol, which makes use of energy and geography-aware heuristics to route a packet towards the target region.

In the case of geographical routing schemes, the approach to selecting the neighbouring node for advancing towards the destination varies. For example, greedy geographical routing schemes [KK00] use a greedy forwarding mechanism whereby each node forwards a packet to the neighbour that is closest to the destination. The approach tries to always shorten the distance to be traveled to the destination to the maximum possible extent. Therefore, the node considers only those neighbours that are closer to the destination than itself. While in *Most Forward within Radius* (MFR) [TK84] the packet is forwarded to the neighbour with the shortest projected distance to the destination. Compass routing [SL01] selects a neighbouring node on the basis of its direction towards destination such that the angle between the neighbouring node and the destination is minimized.

A greedy geographical routing scheme is very efficient and guarantees packet delivery in the case of a dense network, but it can fail to deliver packets in sparse networks or in networks having holes in their topology [JLY^{+}08]. These schemes are also susceptible to the *communication void* problem, viz., the problem caused by the fact that a node may select a neighbouring node that is closest to the destination, but this neighbouring node find itself not a destination (or part of a destination) and not having a neighbour closer to the destination to deliver the packet. Most of the variants of greedy geographical schemes are equipped with a technique to overcome the communication void problem such as routing around the perimeter of the empty region [KK00], upgrading virtual distances [JK09], rebroadcast and bypass strategies [XcL05], planar-graph-based technique [BMSU01], distance upgrading technique using cost-based idea [CFC06], void resolution-forwarding, using quadrant-level right-hand rule [KPSK09]. In particular, [DK06] combines compass routing with a mechanism to explore the area around void regions.

Some algorithms are combinations of two or more strategies. In [ZRLM06], the authors describes an approach in which the decision combines several factors such as location, energy, realistic lossy wireless channel condition and the renewal capability of the energy supply. Seada et al. [SZHK04] motivate energy efficient geographic forwarding. *Trajectory Based Forwarding* (TBF) [NN03] uses a combination of greedy forwarding and source based routing to route a packet along a predefined curve.

2.5 Summary

This chapter has provided the technical context required to understand the research contributions reported in the remainder of this dissertation. First and foremost, it described the centralized ROSE algebra based on finite resolution computational geometry represented with a discrete grid, which gives rise to a discrete, numerically robust, and consistent geometric basis to be used later on. Secondly, the implementation of the centralized ROSE algebra cannot meet the broad range of constraints and requirements that characterize WSN platforms. In light of these observations, defining a spatio-analytic framework and spatial algebra over WSN and its distributed implementation is a novel and potentially useful research contribution. Finally, background work on aggregation and routing approaches in WSN has been discussed. The next chapter presents the framework that supports algebraic abstractions as spatial types with expressive spatial operations upon them.

Chapter 3

A Framework for Spatial Analysis Over WSNs

This chapter presents a framework for performing spatial analysis over WSNs as one of the research contributions of this dissertation. The framework supports expressive algebraic abstractions as spatial types with expressive operations upon them. This framework can be construed as a conceptual platform over which distributed algorithms can be designed in order to perform spatial analysis over WSNs.

The structure of the chapter is as follows. Section 3.1 discusses related work on frameworks for spatial analysis over WSNs. Section 3.2 presents the contributed framework. Recall, from Section 1.5, that one of the research aims underlying the contributions reported in this dissertation is the identification of distributed algorithms for characterizing induced geometries. This is founded on event detection. Therefore, Section 3.4.1 surveys the related work on event and edge detection approaches. The objective is to identify useful techniques for characterizing induced geometries. Section 3.4.1 also summarizes the results of a study of two event detection algorithms, viz., FEBD [RZL06] and T-Fit [Sel06]. The study revealed shortcomings in both algorithms, so we contribute how improved versions of both algorithms [JF08]. The results presented here quantify the trade-offs between accuracy, areal coverage and sensor density that each algorithm incur. Section 3.5 summarizes the chapter.

3.1 Related Work

To date, there has been no effort to develop a comprehensive framework for spatial analysis over WSNs on the same scale as the one introduced in Section 3.2. Therefore, this section discusses those elements that might be part of a framework with reference to published proposals thereof. Thus, we touch primarily on spatio-temporal query processing and on event tracking and monitoring. Note that no proposal in the literature has, to the best of our knowledge, taken an algebraic basis, which is a major concern of this dissertation.

A framework called CLOUD is proposed in [LKH05] to collect and aggregate information over regions generated as a result of event detection. In this work, a proposal is made that after selecting the leader node and computing a tree-like structure, each node reports to the leader node. Hence, the leader node will have the information related to event and the location of the nodes. The leader node is made responsible for the computation of aggregates and of the new leader for the next evaluation period. This is to some extent related to the work described in this dissertation, as for evaluating a spatio-analytical task, the state of a node for a task is also collected and aggregated. However, whilst the focus in [LKH05] is on functions like average, min, max etc., in the work reported here the information that is aggregated is bit-level encodings of the partial results in evaluating a complex distributed task. The reader is referred to Chapter 5 for further details.

EnviroTrack [ABC$^+$04] is a programming abstraction for distributed event tracking applications. A moving object is tracked by dynamically established groups of nodes and a context label is assigned to each such object. To create context, after event detection, each node participates in the leader election. An elected leader sends periodic messages to inform its group member about its existence and is also responsible for sending a handoff message in case an event moves out of its sensing range. Each node periodically sends its sensed value towards the leader using multi-hop routing. The leader performs the aggregation over the received values in each period. The framework described in this dissertation also associates an ID to each geometry to enable addressing those nodes that belong to it and assigning them the responsibility for computations associated with that geometry. Whilst EnviroTrack is restricted to tracking moving objects, the focus of this dissertation is to track complex, transient, evolving physical phenomena and studying the spatial relationships they exhibit with other physical phenomena or with permanent geographical features.

The work reported in [Wel04] focuses on providing a high-level programming interface that abstracts away the details of routing, data collection, data dissemination, and state management. The authors propose a region abstraction for programming WSNs defined in terms of radio connectivity, geographic location, or other properties of nodes. However, there is no indication of how spatio-analytical tasks can be expressed in their approach.

The programming abstraction proposed in [MP06] introduces the concept of logical neighbourhoods whose span is not limited by the radio range but can be specified declaratively based on characteristics of nodes. For example, a logical neighbourhood for a node can be defined as nodes comprising temperature higher than a threshold and that are at a maximum of 2-hops away. The authors also propose a programming language for defining such neighbourhoods and a supporting routing strategy. In the work reported in this dissertation, the information about induced geometry and other geometries is distributed. Each node only knows about the geometries it is part of and can only communicate with neighbours lying in its radio range.

Kumar et al. [PKRVJ05] proposed a framework for distributed event detection using collaboration. It builds an event-based tree over the nodes that are in an event region for information aggregation. The framework supports one mobile node (user node), all other nodes must be static. An event detection message is transmitted by a user node with attributes including area-of-interest coordinates, a threshold value (specifying the number of nodes required to infer an event) and the tolerable delay between events. The proposed framework works with the assumption that each node is location-aware. Upon reception of a message, a node checks its participation in the area-of-interest and whether it can sense the attributes required for the detection of the event. If so, the decision is then made by a node regarding participation based on the threshold value. If successful, it sends a confirmation message to the parent, and in this way an event-based tree is constructed. Once the tree is constructed, each parent waits for a specific period for a response from its children, the information from which is aggregated with its own and transmitted towards its parent on its way to the sink. This dissertation also studies two aggregation approaches for handling distributed aggregation problem, one of which is tree-based aggregation, in which, just like the aggregation scheme presented in [PKRVJ05], each parent waits for a specific period for a response from its children, the information from which is aggregated with its own and transmitted towards its parent on its way to the sink. However, the aggregation function used in this dissertation is a new bit-string-based aggregation function. The aim of aggregation in this dissertation is to compute a fine-grained result of the in-network evaluation of the spatial task.

In [CSN05], the authors focus on a general framework for historical spatio-temporal query processing and describe techniques for queries that retrieve the relevant raw data. For storing historical data, it assumes the stream storage solutions for fixed storage space proposed in [ZGTS03], with which temporal aggregations over a data stream at multiple time granularities can be computed. The proposed framework consists of two phases. Firstly, the query is disseminated from the originator node to a query coordinator node located in the specified spatial area. Secondly, query-related information is collected by the coordinator node from nodes lying in the specified spatial area and the information is transmitted to the originator node. For the first phase, the GreedyDF algorithm has been proposed to find a routing path from the originator node to a query coordinator node that lies within the centre of the query's spatial window. Two algorithms have been proposed for the second phase, viz., WinFlood and WinDepth. In WinFlood, each sensor node that lies within the spatial window will broadcast the query to its neighbours. In WinDepth, each node that lies in the query's spatial window upon receiving the query will forward it only to those neighbours which also lie in the query's spatial window. The query answers from the relevant nodes are returned to the coordinator. The framework described in this dissertation supports in-network distributed spatial analysis. The information about geometries is distributed. The spatial task which may be complex, is transmitted towards the area-of-interest using greedy forwarding scheme and is then disseminated inside the area-of-interest. In contrast with the above work in which raw data is retrieved, in this dissertation the spatial task is evaluated inside the area-of-interest. Leaders are elected at two-levels, one at the level of geometry-element level and other at the MBR-level. Aggregation takes place at two levels: firstly at the level of SEG leaders, and then, at MBR-level leader to which SEG leader respond. The fine-grained result is then routed by the first-level leader towards the sink.

In [SKG05], the authors focus on developing a distributed spatial index SPIX over the sensor nodes for in-network processing of spatial queries. The authors like in [CSN05] exploit the fact that spatial query need to include the information regarding the target region towards which it should be forwarded. The proposed framework works with the assumption that each node is location-aware and the spatial query to be disseminated from the gateway. The construction of SPIX is similar to tree construction rooted at the gateway. After the SPIX construction phase each node is required to maintain *minimum bounding area* (MBA) information. MBA of a node represents the area which covers a node itself, its parent and all child nodes. A SPIX construction message is sent by the gateway. Upon receiving the message each node broadcasts the message. Each node selects a parent node such that it results in the least perimeter enlargement of MBA among the nodes from which it has received message. Upon selection of parent node each node inform its child nodes, so that if they find themselves more closer to their grand parent then they can change parent. Upon reception of spatial query a node forwards the query to its children for evaluation only if any of its children MBA intersects with MBR of the area-of-interest specified in spatial query. In this work the authors have not discussed method to forward the query towards MBR.

To the best of the author's knowledge, no research to date has explored the issue to develop a comprehensive spatial framework that allows for the evaluation of spatial-analytic task for performing spatial analysis over WSNs. The next section describes the spatial framework for performing distributed spatial analysis over WSNs.

3.2 A Spatial Framework for Performing Distributed Spatial Analysis Over WSNs

This section describes the spatial framework for performing distributed spatial analysis over WSN that is one of the main contributions of this dissertation.

This section consists of two sub-sections. Section 3.2.1 describes how geometries can be defined over a WSN as subgraphs of the WSN connectivity graph. Section 3.2.2 describes the kinds of geometry supported by the framework along with the data structures that associate a sensor node with the geometries it is a member of. Section 3.3 sketches a solution for the distributed evaluation of spatial tasks over this spatial framework.

3.2.1 Model and Assumptions

This section formalizes the basic terminology and assumptions upon which the description of the proposed framework is based.

Let $\boldsymbol{M} \times \boldsymbol{N}$ denote the two-dimensional Euclidean plane that discretises a rectangular geographic area G under study. Also, assume a Cartesian coordinate system to describe each point p_i in G in terms of a x-axis and a y-axis (x, y). Let $\boldsymbol{M}$ be the minimum Cartesian coordinates $(\min(p_i^x), \min(p_i^y)) \in G$ and $\boldsymbol{N}$ the maximum Cartesian coordinates $(\max(p_i^x), \max(p_i^y)) \in G$. These two points define one of diagonals of G say, the upper-left to bottom-right one. A set of nodes S is deployed inside G and the overall disposition of nodes may be regular (i.e., grid-like) or not.

It is assumed that a node is location-aware and that its position in the WSN can be determined. The position does not need to be global, it could be relative to a known point or decided by the deployer. It can be computed using *Global Positioning System* (GPS), or any other localization schemes [SHS04, BHE00, DFN06]. We assume that the sensing and communication ranges of a node $s_i \in S$ are representable by circles with radius $r_{s_i}^d$ and $r_{s_i}^c$ respectively.

A node $s_i \in S$ that is located at (x_i, y_i) in G is capable of sensing at a location (u, v) in G if the Euclidean distance $d((x_i, y_i), (u, v)) \leq r^d_{s_i}$. A node $s_i \in S$ located at (x_i, y_i) in G is capable of communicating with another node at location (u, v) in G, if the Euclidean distance $d((x_i, y_i), (u, v)) \leq r^c_{s_i}$.

The 1-hop neighbourhood $N^1(s_i)$, or $N(s_i)$, of a node $s_i \in S$ is the set of all sensors that are in the communication disc centered at s_i with radius r^c. This determines a WSN connectivity graph whose vertices are the deployed nodes and whose edges consist of pairs of nodes $s_l \in S$ and $s_m \in S$ where $l \neq m$, such that each element in the pair is capable of communicating with the other. This, in turn, allows geometries to be defined over the WSN as subgraphs of the WSN connectivity graph.

Given a geographical area G covered with sensors S where for each sensor $s_i \in S$, its coordinates *(x_i, y_i)* are known, then we only consider events and only associate application specific geometries to those points in this Euclidean space where a sensor node is deployed.

Recall that, as discussed in Section 1.4, the running example used in this dissertation is an application of WSNs in precision agriculture. Therefore, throughout this dissertation, it is assumed that the location $(x_i, y_i) \in G$ of each node $s_i \in S$ is given. The assumption is that the deployer decides where, physically, to place each node. However, the contributions made in this dissertation, do not depend on this assumption. Note also that it may or may not be the case that there is a node in every point in grid, but geometries are always defined in terms of deployed nodes.

3.2.2 Description of the Framework

A framework is now presented for describing and constructing spatial analysis systems in a systematic way, allowing for component reuse. Recall from the previous section that a WSN corresponds to a finite, discrete, two-dimensional space that models a spatial domain.

Following the ROSE-algebraic approach [Sch97], application-specific geometries can be defined based on abstract data types such as **points**, **lines**, **regions**.

Spatial values are represented in terms of deployed sensor nodes using the graph view of the corresponding WSN, in the proposed framework. Geometries (i.e., spatial values of type **points**, **lines** and **regions**) can be single-element or multi-element, i.e., spatial values are collections. A single-element geometry is a singleton, a multi-element geometry has more than one constituent geometry and can be used, e.g., to represent disjoint regions of the same event, such as regions with moisture above a given threshold in a cultivated field. Each sensor node is aware of every geometry it is a part of and stores geometry-related information locally in a table called the *Geometric Information Table* (GIT). A node can be part of one or more geometries. Therefore, a unique *Geometric ID* (GID) is assigned to each geometry a node is part of. This can be seen as roughly similar to the way the postcode of a building determines, e.g., the road, the borough, and the town it belongs to. In such a way, all nodes in the WSN that have identical GIDs comprise a specific geometry. Each sensor node stores and manages locally its geometric information in its own, local GIT. Each tuple in the GIT has attached to it a validity timestamp that defines the life time for membership in the corresponding geometry. The reader is referred to Section 5.3, for a description of the GIT.

Recall that, as discussed in Section 3.2.1, each node has a location, which gives spatial properties to data. The proposed framework, therefore, considers the data in the WSNs as a specialized distributed database, with spatial data stored in a nodes GIT. Furthermore, the framework considers three kinds of geometry; viz., asserted, induced and derived.

3.2.2.1 Asserted Geometries

The first kind of geometry is referred to as asserted. Asserted geometries are representations of physical features (e.g., a well, a pipeline, or a cultivated field). An asserted geometry can consist of a single element, or a homogenous collection of elements (e.g., to represent a set of islands or irrigation points inside a vineyard). An asserted geometry of type **regions** may contain holes such as a cultivated field with a hole representing where a lake is located and so on.

These geometries are both assumed to pre-exist the WSN deployment and to remain unchanged for the duration of that deployment. Under these assumptions, nodes can be made geometry-aware for this type of geometry by storing the geometric information in its local GIT and setting the validity period to indefinite. GIDs are also assigned once, while making node asserted geometry-aware.

3.2.2.2 Induced Geometries

The second kind of geometry supported by the framework is referred to as induced. Induced geometries are representations of transient phenomena determined by physical properties that can be sensed by the nodes. These geometries are assumed both not to pre-exist the WSN deployment and to change independently, possibly often, for the duration of that deployment.

Induced geometries are characterized by means of event detection and boundary computation. Briefly, by *event detection* is meant the outcome of a distributed process in which participating nodes evaluate the event-defining predicate and, as a consequence, declare themselves an event node if the predicate is satisfied. The boundary of the event geometry can then be computed using distributed boundary detection algorithms. A boundary detection algorithm allows a node to determine whether it is lying on the boundary or the interior of an induced geometry. The reader is referred to Section 3.4, for a detailed description of boundary detection approaches. Each node stores this boundary membership information, i.e., whether it is lying in the interior or the boundary of the geometry, in its GIT. The GID for induced geometries is passed as a parameter of the message that contains the task that characterizes it.

Induced geometry characterization may result in multi-element geometries, i.e., one consisting of disjoint component geometries. Furthermore, the interior of an induced geometry may contain non-event nodes, in which case the induced geometry will contain one or more disjoint holes. In Section 3.4.2.4, a real-world example is presented. Figure 3.1 shows the interpolated data representing measurements of distribution of chlorophyll obtained in Lake Fulmor (at the James Reserve in the San Jacinto Mountains, California, USA) as reported in [SSC^{+}07]. Figure 3.1 shows the interpolated data. The x-axis shows the interval [5m, 35m] across the transect, the y-axis show the interval [-1.0m, -4.5m] in depth. In Figure 3.1, the event predicate *chlorophyll-levels* < 20 gives rise to a multi-element event geometry with three sub-regions (shown in Figure 3.2).

In the case of induced geometries, the fulfilment of the requirement that a node is geometry-aware implies the need to reconsider periodically its event-membership status from sensed data at the relevant evaluation episodes. The distributed algorithms for the characterization of induced geometries are discussed in Section 3.4.

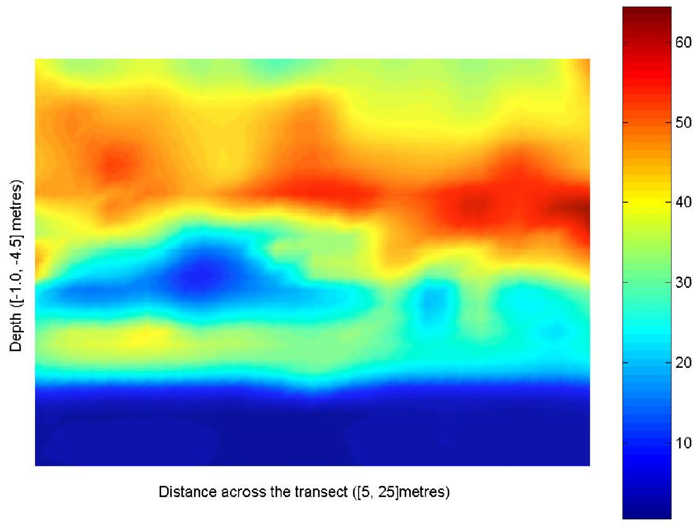


Figure 3.1: Distribution of Chlorophyll, Lake Fulmor (adapted from [SSC+07])

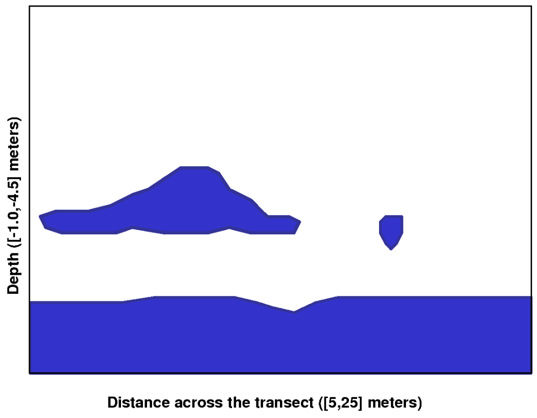


Figure 3.2: Event Geometry on applying the predicate (chlorophyll levels lie below 20) to Figure 3.1)

3.2.2.3 Derived Geometries

The third, and last, kind of geometry is referred to as derived. Derived geometries are obtained by applying spatial-valued operations to existing spatial values (e.g., applying `intersection` to two values of type **regions** would result in a new geometry of type, say, **regions**). The computation of derived geometries involves two components: membership detection and boundary computation. Briefly, by *membership* detection we mean that a node checks whether it satisfies the primitives required for the success criteria of the operation. Recall that, as described in the previous section, in the case of a geometry of type **regions**, a boundary computation algorithm is used to allow nodes to determine whether they are lying on the boundary or the interior of the derived geometry. In the case of an asserted geometry of type **regions**, nodes can be informed whether they are lying on the boundary or interior of geometry manually, or the nodes can compute such information through boundary detection algorithms. A derived geometry can consist of a single element or a homogenous collection of disjoint elements. A derived geometry of type **regions** may contain one or more holes. Given that one (or both) of the arguments for the derivation of a geometry can be induced, nodes in derived geometries also need to maintain their membership status in GIT dynamically. A node sets the validity period of a derived geometry as follows: it sets the validity period to indefinite if both operands are of type asserted; otherwise, it sets the validity period to be the minimum of the validity periods among the operands of the operation. The GID for derived geometries is also passed as a parameter of the spatial task message that defines it.

The algebra for the spatial-valued operators that leads to the characterization of the derived geometry is discussed in Section 4.2.2.

3.3 Distributed Spatial Task Processing

Recall, from Section 1.6, that the contributions made in this dissertation concern the efficient, in-network processing of spatial tasks over asserted, induced, and derived geometries. As a major constraint on sensor nodes is their limited energy supply, the focus is on energy-efficient techniques to process the spatial task inside the WSN.

In the proposed framework, an induced geometry is periodically evaluated for true-representation of the corresponding evolving phenomenon. Each node updates its geometry status regarding induced geometries at the end of each evaluation period. If an entry already exists for that geometry in the GIT of a node, its validity period is reset (typically, in the case of periodic evaluation, until the next evaluation period). Otherwise, a new entry is added to the GIT. On the expiry of the validity period, the entry is removed.

The framework also supports periodic evaluation of tasks related to the computation of derived geometries and computation of spatial relationships. The evaluation period (e.g. every 30 minutes) and duration for the task are passed as parameters in the task message that disseminates the task into the relevant nodes in the WSN.

Efficiency is achieved because information about geometry types is kept inside the WSN, thereby avoiding sending information out about these geometries at the end of every evaluation period. Spatial tasks can be evaluated over these geometries periodically and only the final answer is sent back to the user. The user can request explicitly for the information about the geometry out of network, if required.

For task message dissemination, one simple method consists of flooding the task message inside the WSN. This approach, however, can be expensive in terms of network congestion, and energy consumption [JA07, CHLS07]. For example, it is not a suitable technique for applications where multiple spatial tasks in different parts of the network must be evaluated because it results in too much network congestion. In addition, for spatial analysis, users may prefer to query a small *geographical area* inside the WSN rather than all the network [CSN05]. In this case, there is only a need to send the spatial task to one specific target region. Transmission efficiency can be achieved if the algorithm detects a path that reduces the number of relay nodes and also by reducing message payloads by the use of compression and aggregation techniques.

Recent research [WMHF03, ZRLM06] suggests that geographic routing algorithms are more suitable for WSN. There is also evidence that location-based greedy forwarding techniques [Sto02, KPSK09] are helpful in the dissemination of spatial task containing geometric information about specific area. In such a scheme, before forwarding the task message, each node needs to compute the distance from itself to its neighbouring nodes and select the neighbour that lies closest to the destination. The reader is referred to Section 2.4 for a description of greedy-forwarding routing protocol.

The framework introduced here for distributed processing of spatial tasks in a WSN targets the area of interest corresponding to each task. A gateway is assumed to exist that acts as the source for disseminating tasks expressing the desired analysis and as the sink for receiving the corresponding outcomes. Spatial tasks are input at the gateway. The gateway is responsible for storing all referable geometries and for the computation of the *Minimum Bounding Rectangle* (MBR) representing the target region. The derivation of an interpretable structure in postfix notation from the task specification is performed at the gateway and gives rise to an interpretable structure that is evaluated by the task processing system with which each node in the network is equipped with. The gateway computes the task MBR and includes its coordinates as parameters in the task message. This is done by scrutiny of the geometries specified as operands in the task. Based upon the semantics of analysis represented by task (requirements of the operators and operands part of the task) the MBR may comprise the whole WSN or part of it. Note that task involving induction of geometries need to run on every node in the network, periodically. For example, consider the example in Section 1.4. If the task is ($f5$ `area_disjoint` M) the gateway compute MBR that encloses $f5$. The task MBR denotes the rectangular region of the WSN in which the task needs to be disseminated and evaluated.

Task dissemination involves two stages: firstly, greedy forwarding is used to send the task message towards the MBR, secondly, the task message is then broadcast within the MBR. A greedy forwarding dissemination strategy is suitable because it is completely localized. However, it is not claimed as a research contribution of this dissertation since it is inspired by [KK00, JK09, XcL05, BMSU01, CFC06, KPSK09]. Broadly speaking, task dissemination uses greedy forwarding to send the task message from the root to the task MBR. The first MBR node that receives the task message behaves in a special way: it takes on the role of *first-level leader*. The *first-level leader* records the destination ID, the source node ID as the parent node ID, updates the information in the packet related to the cost of reaching the source, and sets its own ID as the source node ID and destination ID in the task message before broadcasting it. Each MBR node uses its location to compute whether it is part of the task MBR. Upon receiving the task message, a task MBR node other than the first checks whether the cost of reaching the source is less than the one it has recorded earlier (this may be so because it may have already received the same task message from more than one neighbour). If it is, it records the parent node ID and the source node ID, updates the cost of reaching source, and broadcasts the task message. If the node is not part of MBR, it discards the message. The reader is referred to Section B.1.1 for the description of attributes of the task message. At the same time that the task message is forwarded towards the MBR and then flooded inside MBR, nodes assemble into the routing tree to be used for aggregating the results from disjoint geometry elements and for delivering results back to the gateway.

A task processing system has been developed that allows the users to evaluate tasks inside a WSN. This contribution is based upon the division of an algorithmic strategy comprising a small set of logically-cohesive components. Each node is equipped with task evaluation components that allow it to participate in task dissemination, to contribute in the distributed evaluation of a task, to participate in the aggregation of intermediate results and to route results to the user. The reader is referred to Section 5.6.2 for a detailed description of the components of the task processing system. The algorithmic strategy for in-network distributed spatial analysis over WSNs is given in Chapter 5.

As described above our framework supports predicate-defined induced geometries the characterization of which requires though the detection of the boundary of the event geometry. The next section describes the related work on the schemes for boundary detection.

3.4 Background: Event and Edge Detection Approaches

This section aims to identify a suitable candidate for event boundary detection as required for geometry induction. Event detection is a distributed technique whereby each node computes whether it has detected an event. Sensor nodes can be used to detect phenomena that are extended in space. This dissertation specifically consider those events that can be modeled as having a boundary, or edge. Examples include temperature gradients, variations in levels of measurable quantities such as light intensity, chemical concentration, etc. These events are detected by applying an event predicate (such as $temperature > 90$) to the readings obtained from sensing devices present on nodes. The event characterization predicate takes the form of a threshold test. Event detection can be simple or complex. Complex event detection requires readings from more than one sensing device in the node. In addition, the characterization predicate may also be a more complex algebraic expression (such as $temperature > T_t$ *AND* $light > T_l$) requiring, therefore, that more than one threshold test is applied over several readings as well as the application of Boolean connectives to compute the final result.

Intuitively, the boundary detection problem can be formulated as follows: given a set of readings from sensors located at different points in the field and an event characterization predicate, return a description of the boundary of the event. In other words, the boundary separates the sensor nodes that satisfy the characterization predicate and those that do not. The boundary of an event geometry (i.e., the spatial shape of event) G_e is defined as the set of nodes lying on the boundary of G_e, denoted by $B(G_e)$. $B(G_e)$ describes the shape and the location of the event.

It is challenging for algorithms running in a WSN to detect an ideal boundary due to inherent limitations of WSNs such as sensor noise (e.g., faulty measurement by the sensor), unreliable wireless communication links, etc. Many proposed solutions exist for boundary detection, including those based on image processing [CG03], topological and geometrical techniques [CG03, JWZ$^+$09, NM03, Sel06] as well as statistical [CG03, JN06, DCXC05] schemes. This section now briefly surveys the most prominent proposals.

3.4.1 Algorithms for Event and Edge Detection

To the best of the author's knowledge, the first characterization of event boundary was reported in [CG03]. In that paper, the authors propose three different approaches: statistical, image-processing, and classifier-based. None of the approaches have any mechanism for detection and suppression of noise.

A statistical approach requires an event node to collect information from its neighbours, derive a set of statistics from that information and use a Boolean decision function in order to decide whether it lies in the event boundary based on an acceptance threshold. The image-processing approach treats each sensor as a pixel, thereby opening the way for the direct application of edge detection techniques used in the treatment of images, e.g., those based on computing a filtered image using convolution, Prewitt filters etc. Such filtering techniques do not consider either the topology or the geometrical locations of the sensor nodes. Note that if a WSN deployment is such that the required pixel-like regularity is not observed, a weighting scheme can be used based on the number of event and non-event neighbours. Finally, in the classifier-based approach, a node attempts to partition the information collected from its neighbourhood into regions of distinct behaviour using classification techniques.

The authors of [CG03] instantiate each approach and comparatively evaluate their instantiations using detection accuracy and boundary thickness as their metrics. For those particular metrics and instantiations, the classifier-based approach both produced better results and was shown to be less susceptible to poor performance in the presence of errors. In the first two approaches, based on statistics, and on image processing, the correct choice of a threshold value is very critical for correct identification. The statistical technique performs better than the image processing one in scenarios where all the sensor nodes are well calibrated, error free and uniformly deployed. The performance of both schemes degrades with an increase in network density, a decrease in neighbourhood size, and an increase in the percentage of faulty measurements. In addition, the authors of [CG03] noticed that for all three schemes, an increase in the neighbouring area (e.g., collecting information on two-hop neighbours) results in higher performance results albeit at the cost of more communication.

Jitender et al. [DDHG05] proposed the Interior Point (IP) algorithm to discover boundaries in uniformly and randomly distributed WSNs. The authors assumed that each node must have at least three neighbours in its radio range. The algorithmic strategy comprises two algorithms: *Interior Point* (IP) and *ChooseGoodNeighbours* (CGN). In order to detect edge nodes, IP requires nodes to broadcast their location information to their neighbours. The IP algorithm confirms whether a node is in the radio range of three neighbours. The accuracy of the algorithm depends upon the best selection of the three neighbours. For this purpose, the CGN algorithm is responsible for intelligently selecting four neighbours that are pairwise neighbours of each other. CGN selects neighbours that are close to a node and possibly surround it. Experimental evidence is derived using size of neighbourhood and the network as metrics in both random and uniform grid deployments. The accuracy of the algorithm increases if the neighbourhood increases to more than 4. It has been found that the accuracy of the algorithm decreases if the network density increases but that the inaccuracy does not increase much in the case of random deployments compared to grid deployments.

Zhang et al. propose two algorithms based on computational geometric techniques, called *Localized Voronoi Polygon* (LVP) and *neighbouring Embracing Polygons* (NEP)[JWZ+09]. In one LVP-based algorithm, the *Tentative Localized Voronoi* is computed by each node using the nearest neighbour distance and direction information. Based on such information, the neighbours are divided into quadrants. If neighbours are found in all four quadrants, a node declares itself a non-boundary node. If a node cannot find neighbours in any quadrant, then it checks for neighbours in the assistant area (constructed by calculating two sectors of 45° each, adjacent to the specific quadrant). If the neighbours in the assistant area are not the nearest neighbour in that quadrant, then the node declares itself a boundary node. The LVP-based algorithm is reported to provide continuous closed curves as boundaries. In the case of the NEP-based algorithm, a node sorts its neighbours according to their angle with itself (to create a convex hull of its neighbouring nodes). After sorting, if node finds a gap less than or equal to π among these angles, it declares itself a boundary node. The authors have reported that, compared to the LVP-based algorithm, the NEP-based algorithm provides less accuracy. The LVP-based algorithm provides higher accuracy both for uniformly deployed as well as arbitrarily deployed nodes.

Jaffer et al. [JJJES07] used an autonomous agent based approach for event boundary detection in WSNs. The main objective of the authors was to decrease the communication cost by improving the transmission efficiency. Initially, the agents are generated by event nodes based on a preset threshold value. The node in which the agent resides requests a response from non-event neighbouring nodes. The agent makes a decision about the selection of the next boundary node on receiving the response from its neighbours. Upon finding the first boundary node, the agent generates a child agent. Both the agent and the child agent then start moving around the boundary in the opposite directions until they meet. An agent stores the boundary nodes which it has visited in its boundary stack. The threshold value used for generation of an agent is not constant and is dependent on the phenomenon size. According to the experimental results, the scheme performed well for networks with high node density. Furthermore, the efficiency of the algorithm increases as an inverse function of the event region radius. No discussion is provided regarding the stack size of the agent in which it keeps boundary-related information, or on the cost associated with agent movement from one node to another.

Noise-tolerated Event and Event Boundary Detection(NED) [JN06] supports noise suppression in event boundary detection in WSNs. NED assumes that sensing errors are independent over the WSN, and is a white normal random variable (i.e., sensor error $\in$ N(0, σ^2)) and have fixed variance because of further assumption that all nodes were sourced from the same manufacturer batch. In addition it is assumed and that the event phenomenon is continuous. NED uses a statistical approach in which the probability density of a normal random variable concentrates around the mean value. So, for a random variable, N(μ,σ^2), 95% probability falls within the range ($\mu - 1.96\sigma$, $\mu + 1.96\sigma$). Since the sensing error is considered normal white noise, nodes are classified as significant, non-significant event, non-event based on the threshold value and variance σ^2 of the sensing error. Such classification allows the transmission of a message using variable length coding mechanism for communication efficiency. The NED algorithm makes use of a *moving mean* method for suppressing sensor faults. As the authors focused on continuous phenomena to represent sensed data in WSNs, they conducted experiments using a smooth gray-scale image. Performance analysis of the NED algorithm has not been provided in terms of varying network density and neighbourhood. The experimental results show that with a large density and moderate noise, NED performs well for detecting the edge of continuous phenomenon.

Nowak and Mitra [NM03] proposed a hierarchical processing strategy using a cluster-based scheme for edge detection in WSNs. The whole sensor field is first divided into four quadrants, and each quadrant is then recursively divided into 4 sub-quadrants of equal size until some maximum resolution is reached. Then, cluster formation occurs. Each of the sensor nodes in the sub-quadrant is then responsible for transmitting their original measurements to their sub-quadrant cluster head. The cluster head is then responsible for computing the average and other statistical measurements and to transmit its estimates regarding the subregion to the cluster head up in the hierarchy. A quad tree is used for representing this hierarchical structure. The cluster head on the higher level then performs some further processing in order to determine the sub-partition that provides an approximation to the boundary. The algorithm does not involve any mechanism for detection of noise. However, since the average of the measurements are used instead of the original ones, there is suppression of noise to some extent. Under this scheme, upper bounds are set on the Mean Square Error(MSE) of the estimator based on the smoothness of the curve. The authors concentrated on the trade-off between the MSE and the communication cost as a function of node density. The scheme provides better accuracy at low communication cost for low and medium density networks but at the cost of complex regularization of the hierarchical tree-based estimation method. The MSE increases with an increase in node density. One of the limitations of this scheme is that it is based on a hierarchical polling method which implies high cost in terms of cluster formation, cluster head selection, its maintenance, as well as long transmission distances.

Tangent fit (T-fit) [Sel06] is another localized edge detection technique based on geometric rules and trigonometry. In order to detect edge nodes, it requires event nodes to broadcast their location information to its neighbours. Upon reception of the messages from event nodes, a node N makes itself the origin of a circle centered at the node and partitions its neighbouring event nodes into four quadrants. The edge-detection statistic is then formulated in terms of the number of quadrants in which neighbouring event nodes are found.

If no neighbouring event nodes are found in any quadrant or else if they are found in all four quadrants, then N declares itself *not* to be an edge node. If neighbouring event nodes are found in one quadrant only, N declares itself to be an edge node. If neighbouring event nodes are found in either two or three quadrants, N computes the angle formed by itself at the origin and its two farthest neighbours in the two populated quadrants (or in the diagonal quadrants, in the case of three populated quadrants). If that angle is less than 180°, N declares itself to be an edge node. Experimental results shows that the scheme performs comparatively well compared to the PR-classifier algorithm [CG03] both in terms of accuracy and energy efficiency. The experimental evidence suggests that the performance of the scheme increases by increasing node density even in the case of arbitrarily placed nodes.

The localized fault-tolerant event boundary detection scheme [DCXC05] assumes that the set of sensor nodes with faulty measurements may contain information related to detecting events. The algorithm for faulty sensor detection is based on the moving median method, which requires every node to broadcast location information along with the sensed measurement to its neighbours. After the computation of the median, for the computation of event boundary detection, a node computes the difference d between its own sensed measurement and the median. Each node is then required to broadcast d in another message. Several statistical tests are then applied to compute the resultant value, which is then compared against the threshold to determine whether the node is an edge node. The algorithm is sensitive to the settings of the threshold, which is based on the sensor node fault probability. The communication cost of the moving median algorithm is approximately 32 times higher than the cost of a majority voting algorithm [RZL06] as it requires the sensed measurement to be broadcasted. The experimental evidence suggests that a detection accuracy of around 90% is achieved with node densities greater than 30 and that the probability of error is less than 20%.

The fault-tolerant event boundary detection scheme (FEBD) [RZL06] is based on Bayesian theory [KI03, KI04]. It is another distributed localized boundary detection scheme designed for WSNs. FEBD requires every node to send the outcome of its event detection predicate to all its one-hop neighbours, irrespective of whether that outcome was true or false. The scheme used by FEBD to contend with sensor errors is to subject a node's own call as to whether it is an event node to a majority rule [KI04]. Thus, the call of every one-hop neighbour of a node is counted. If the majority vote concurs with the node's call, then that call is allowed to stand, otherwise not. If a node calls itself an event node, then it computes a statistic to determine whether it is an edge node by comparing that with a threshold value. For this purpose, FEBD takes as input an acceptance threshold. The acceptance threshold is designer-set and thus should be decided taking into account any relevant deployment properties (such as the tolerance radius for edge thickness, network density, neighbourhood size, etc.) The performance of the scheme decreases with an increase in the number of faulty measurements due to the increase in false detections, especially in the case of low density networks. Compared to the schemes in [CG03, DCXC05], the performance of FEBD in terms of correctly detecting boundary nodes is not dependent on the density of the network and, gives reasonable results at low communication cost, even with a faulty measurement percentage up to 25%.

Zeinalipour-Yazti et al. [ZYACS07] have presented a perimeter algorithm for distributed boundary detection. This scheme requires that each node is aware of its neighbour's location. A randomly chosen node identifies the minimum y-coordinate in the field by constructing an aggregation tree [MFHH02] rooted at it. During aggregation, each node identifies among itself and its children which node has the minimum y-coordinate value. The identified node is marked as the starting perimeter node and is responsible for selecting, among its neighbouring nodes, the next perimeter node as the one that forms with it the minimum polar angle. The communication cost of the algorithm is high as it involve message flooding to construct a tree.

This section described some of the existing approaches for boundary detection of event regions. The motivation behind it is to characterize an efficient scheme that can be used for geometry induction.

There was a significant practical drawback as if environmental scientists were to try and seek guidance in the literature as to which, among proposed edge detection solutions, would perform better for a planned deployment, they would find the absence of comparative evaluations a serious impediment for an informed decision. In line with this motivation, Section 3.4.2 describes the experimental comparison of edge detection methods, on the basis of which one of the edge detection method has been selected for geometry induction. Further details are in the paper [JF08] that has been published based on this work.

3.4.2 Experimental Comparison of Algorithmic Techniques for Edge Detection

This section describes the results of an experimental comparison of two edge detections methods, viz., FEBD and T-Fit with the modifications proposed in work [JF08] and described below.

These two algorithms were chosen because they were shown in [RZL06] and [Sel06] to have good edge detection accuracy with good complexity properties. Furthermore, they produce good results over a realistic parameter space, e.g., network density, network topology, neighbourhood size, etc. Finally, they are both conceptually simple to understand and hence relatively easy to analyze, implement and evaluate, whilst still being conceptually distinct enough to suggest that their outcomes might not be strictly positively correlated. Here, the interest is in characterizing the effect on accuracy of varying the areal coverage, sensing fault probability and the number of nodes in a deployment. The two methods are comparatively evaluated with inputs that are realistic (i.e., consist of events, adapted from those reported in [SSC^{+}07, HAG^{+}07], whose geometry occurs in nature, rather than the Platonic forms (such as circles, squares and ellipses) used in the original papers.

3.4.2.1 CFEBD: A Cautious Version of FEBD

The original scheme used by FEBD [RZL06] for contending with sensing errors can be shown to have some shortcomings. As presented in [RZL06]), the fact that a node has detected an event must be confirmed by a majority vote where the pool of voters consists of all nodes in the neighbourhood defined by the radio range RR. In experiments, it has been identified that for smaller size event regions whose shape is other than elliptical or circular, i.e. those in which narrow bands or acute jagged edges occur, the scheme in [RZL06] produces poor outcomes if the node density is high. Consider Figure 3.3. It depicts a WSN deployed as regular grid in which there is a sensor in each point of the field. The left figure depicts, by the presence or absence of shading, two regions of distinct behaviour. The right part of Figure 3.3 exemplifies what the outcome of edge detection might be. For example, assume that, in the right part of Figure 3.3, nodes are, say, 10m apart and the radio range is 15m. In addition, the node lying

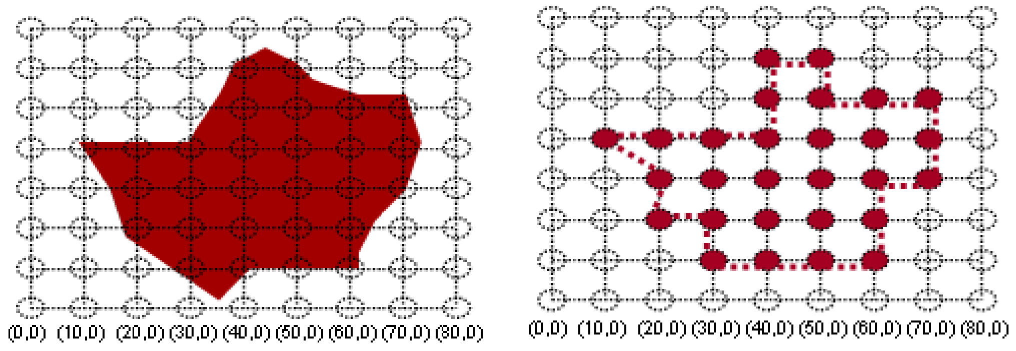


Figure 3.3: Actual Event Geometry (*left*); Induced Geometry (*right*)

at the bottom-left corner has the location (0,0). It can be seen that the node at (10,40) (2nd column from the left, 5th row from the bottom) has two neighbours calling themselves event nodes (i.e., at location (20,30) and (20,40)) and six failing to do so. In this case, FEBD (10,40) submits to the majority rule, reverses its decision and declares itself a non-event node.

Essentially, in this case, FEBD is overconfident in judging a measurement to be erroneous because it gives voting rights to too many nodes. The refinement proposed is that the pool of voters is shrunk when the density is above a certain threshold. That is, when $d < RR/2$, the pool of voters consists only of those nodes in the half-neighbourhood of the node defined by the radius $RR/2$. Because this approach makes the original FEBD more cautious when judging a measurement to be erroneous, this modified version is called *cautious* FEBD or, CFEBD, for short.

3.4.2.2 The T-Fit Algorithm

Unlike FEBD [RZL06], T-Fit [Sel06] only requires a node N to send the outcome of evaluating its event detection predicate (and location information) to its one-hop neighbours if that outcome is positive, i.e., if N calls itself an event node. The implications of this for the comparative message complexity of the two algorithms are, of course, significant, because whilst the message size in T-Fit increases with the location information with respect to FEBD, the number of messages exchanged decreases. T-Fit has no error-suppression. The implications of this for the comparative accuracy of the two algorithms are, of course, significant.

Note that, in the original algorithm, there is some ambiguity as to the definition of a quadrant in respect of the treatment afforded to nodes that lie along an axis. Such assumption has been found to give erroneous results, when the event geometry shape is irregular. It can be seen that, in Figure 3.3, the event node at location (50,50) has neighbours in three quadrants and four axes. The original T-Fit [Sel06] causes the node at location (50,50) to declare itself a non-boundary node. Therefore, such cases are explicitly treated, as follows. The neighbouring nodes lying on quadrants and axes are considered separately. If an event node has no neighbours in any quadrant and there exist neighbours in all four axes, then it declares itself a non-edge node. Otherwise, an event node is an edge node.

3.4.2.3 CFT-Fit: A Cautious, Fault-Tolerant Version of T-Fit

Empirical study has, revealed shortcomings in T-Fit which led us to modify the original algorithm in three principal ways, as follows. As presented in [Sel06], the T-Fit algorithm has no provision for contending with sensor error. Therefore, an error-suppression phase is added to the original T-Fit algorithm, and, for comparability, the error suppression scheme used in CFEBD above is used. In contrast with the original T-Fit algorithm, now every node must transmit a message to its neighbours and the message will still contain additional information about location.

It has been identified in experiments that when the event is not connected (i.e., it exhibits the geometry of an archipelago, in pictorial terms), then the assumption that a node whose four quadrants contain event nodes is, for that reason alone, not an edge node is too optimistic. Now that each node has information about both its event and its non-event neighbours, a modification is introduced that has the effect of introducing more caution into this decision, as follows. When an event node has neighbours that are event nodes in all four quadrants but in any one of those quadrants, at least half of the nodes are not event nodes, then that quadrant is considered abnormal and a more precise rule is applied for the case in which there are nodes in three quadrants, in this case, the three remaining quadrants. Because the modifications introduced made the original T-Fit both more cautious as well as fault-tolerant, this modified version is called as *cautious, fault-tolerant* T-Fit or, CFT-Fit, for short.

3.4.2.4 Example Used

For defining inputs to the algorithms under evaluation, measurements of the distribution of chlorophyll obtained in Lake Fulmor (at the James Reserve in the San Jacinto Mountains, California, USA) as reported in [SSC$^+$07] are used.

The main interest is not in using these values as such, instead, what is important for evaluation is that the *event geometries* are naturally occurring, rather than idealized. In this respect, the event predicate is satisfied if chlorophyll levels lie below 20, i.e., the low concentration values in Figure 3.1. This give rise to a disconnected event with three sub-regions as shown in Figure 3.2.

3.4.2.5 Experimental Design and Set-Up

The experiments aim to characterize the effect on accuracy of varying the areal coverage and the number of nodes in a deployment. A radio range RR of $15m$ is assumed.

The F-measure is used as the dependent variable, i.e., the weighted harmonic mean of precision and recall (expressed in terms of true positives (TP), false positives (FP) and false negatives (FN)), as follows: $F = (2 \cdot TP)/((2 \cdot TP) + FP + FN)$. If the algorithm calls a node an edge node that lies in the actual sensed boundary then it is a true positive (and so on for all the other elements in the F-measure definition). For the purposes of computing the F-measure, the thickness of the edge boundary is assumed to be $RR/4$, which is more stringent than the value ($RR/2$) that the literature tends to report. The real coverage (in m^2) and the number of nodes are the independent variables. The assumption made here is that the area is covered with a grid of square cells in which nodes are deployed at the resulting coordinates. Given $RR = 15$, the following range of values for the side of a square cell in the grid is $\{3m, 6m, 9m, 12m\}$. When there exists a sensor node at every point in a regular grid with n^2 cells, the number of nodes is $(n + 1)^2$.

Experiment 1 studies the accuracy of FEBD, T-Fit, CFEBD, and CFT-Fit for the four values of areal coverage above when there is a sensor at every point in the grid, and no sensor reading errors. Experiment 2 studies the accuracy for the four values of areal coverage above when there is a sensor at every point in the grid and the sensing density is constant at 20 nodes in the one-hop neighbourhood, but (a randomly distributed) 15% of the readings are erroneous. Experiment 3 studies the accuracy for the four values of areal coverage above when there are no sensing errors, but (a randomly distributed) 15% of the points in the grid do not have sensor nodes (this could be either because the nodes have failed or because either a design decision advises against, or a physical constraint prevents, a node being deployed at that point). As a result, the sensing density varies, but is kept at a maximum of 20 nodes in the one-hop neighbourhood.

Note that, because of the randomization, in the case of Experiments 2 and 3, for each setting of the independent variables, each algorithm is run 25 times and the average is plotted as the F-measure achieved by the algorithm.

As a consequence of the simulated setting, the algorithms do not acquire readings, rather the measurements are read from data files (extracted these from the values in Figure 3.1).

In order to compute the F-measure, reference event geometries are defined for each point defined by a pair of values for the independent variables.

3.4.2.6 Experimental Results

The results of Experiment 1 for each of FEBD, T-Fit, CFEBD and CFT-Fit are shown in the four plots in Figure 3.4 and Figure 3.5. The results of Experiments 2 and 3 are shown in the left and right plots in Figure 3.6, resp..

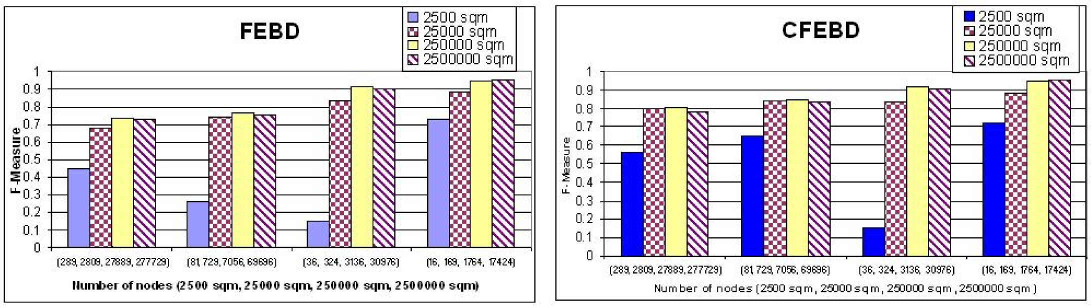


Figure 3.4: F-Measure-Based accuracy under different areal coverages for FEBD (*left*) and CFEBD (*right*)

The results of Experiment 1 confirm the intuition that, in the absence of sensor errors and with a measurement available from every point in the grid, the tendency is for accuracy to grow as the area covered grows, from left to right, within each cluster of bars. Recall that the number

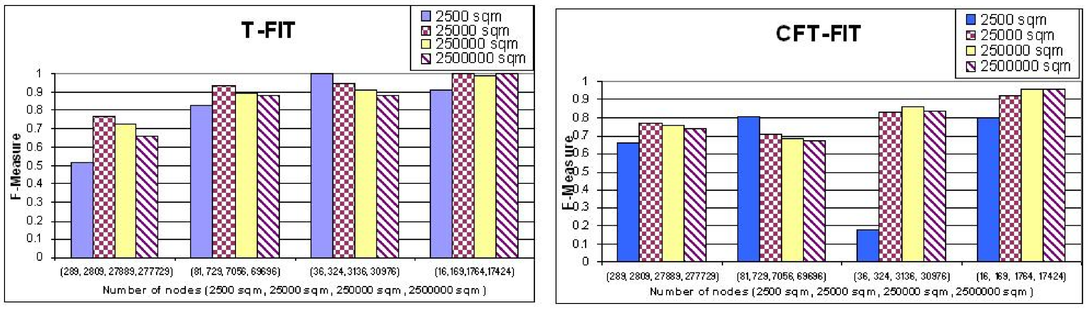


Figure 3.5: F-Measure-Based accuracy under different areal coverages for T-Fit (*left*) and CFT-Fit (*right*)

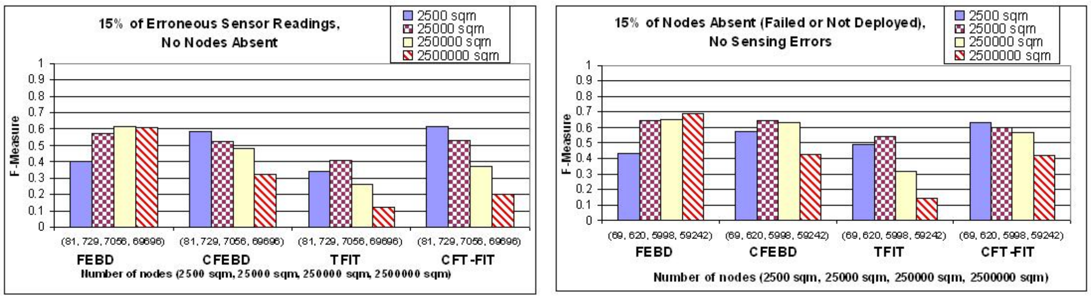


Figure 3.6: Results from Experiment 2 (*left*) and Experiment 3 (*right*)

of nodes associated with each cluster *decreases* from left to right. Thus, the sensing density decreases from left to right, cluster by cluster. The import of this is that for those algorithms that are less robust to varying sensor density, their accuracy is less consistent for the same areal coverage. As explained in Sec. 3.4.2.1, the majority voting method results in more false negatives when the shape of the event region is jagged, especially when the proportion of event nodes to non-event nodes is small. In such case, the refinement described in Sec. 3.4.2.1 can improve accuracy, as shown in Figure 3.4 (*right*) when the distance between neighbours is $3m$ or $6m$. In Figs. 3.4 and 3.5, when, for the same areal coverage, the distance between neighbours reaches $9m$, the accuracy of FEBD, CFEBD and CFT-Fit dips because the majority vote tends to give wrong results and the refinement introduced is not capable of compensating for that (because $9 \not< RR/2$). Note that, for the same areal coverage, since T-Fit does not use any error correction scheme, the accuracy does not dip when the distance between neighbours makes the same transition. When the distance between neighbours reaches $12m$, the four algorithms show no significant differences.

As shown in Figure 3.5, the introduction of a fault-suppression scheme in CFT-Fit compared with the original algorithm, is detrimental to accuracy, but not to any significant extent. This can be verified by comparing the left plot (without fault-suppression) and the right plot (with fault-suppression) in Figure 3.4. When errors are present and when measurements are not available at some of the points in the grid, CFT-Fit is never worse than T-Fit and is often better, as shown in Figure 3.6.

FEBD and CFEBD perform better, in general, than T-Fit and CFT-Fit in the presence of errors and when measurements are not available at some of the points in the grid, as shown in Figure 3.6. The results of Experiment 2 also reveal that FEBD is less sensitive than the other algorithms to an increase in areal coverage. It does perform poorly when sensing density is high, as shown by the fact that CFEBD, which specifically corrects for that, outperforms FEBD.

CFEBD and CFT-Fit outperform the algorithms as originally proposed if errors are not present and measurements are available at all points in the grid as shown in Figure 3.4. However, it could be argued that most deployments at this stage in the development of WSN technology are such that errors are present and measurements are not available at some of the points in the grid. In this case, as shown in Figure 3.6, CFEBD slightly outperforms CFT-Fit, and FEBD performs better than both. From Experiments 2 and 3, it can be seen that T-Fit and CFT-Fit are less robust to increases in areal coverage that involve large numbers of nodes.

One lesson that can be drawn from a comparison of Experiment 1 and Experiments 2 and 3 is that, in spite of advances made since the first characterization of the sensor network event detection problem, current solutions remain sensitive to sensing errors and to irregular deployments.

In the light of this experimental work, the CFEBD algorithm is used in this dissertation in order to induce geometries.

3.5 Summary

The contributions of this chapter were as follows. Firstly, a taxonomy of the existing WSN proposals was presented on the grounds of the focus of event monitoring, and the framework for the spatio-temporal query processing. Secondly, it has been noticed that a WSN deployment provides a discrete basis over which application-specific geometries can be represented, by considering the nodes as points and the communication links as line segments between them in a numerically consistent manner. This is similar to the *realm* concept that underpins the ROSE algebra. Thirdly, a generic framework for distributed in-network spatial analysis over WSN was proposed to facilitate conducting spatial analysis over WSN. It is based on the decomposition into and the separate investigation of three distinct phases that are inherently present in any WSN framework supporting in-network processing: dissemination of spatial task, distributed evaluation of the task, and routing of fine-grained results to the user. Fourthly, a survey of the related work for characterizing induced geometries was carried out. An empirical study of the two most promising event detection algorithms in the literature was carried out so that a decision could be reached regarding their use for geometry induction in the remainder of this dissertation.

Chapter 4

A Spatial Algebra for Distributed Spatial Analysis Over WSNs

This chapter describes the spatial algebra that is one of the main contributions of this dissertation. It is closely inspired, in its scope and structure, by the Schneider-Guting over the geometries representable by the framework presented in Chapter 3 for performing distributed spatial analysis over WSN. The algebra is defined over the finite, discrete, two-dimensional sensor space defined by a WSN deployment as described in Section 3.2. The spatial algebra comprises three spatial data types, which are collections of **points**, **lines**, or **regions**. The values for these spatial data types are geometries as described in Section 3.2.2.

The structure of the chapter is as follows. Section 4.1 defines the data types supported by the framework. Section 4.2 describes the definitions of the spatial operations. Finally, Section 4.3 summarizes the chapter.

4.1 Spatial Data Types

The spatial algebra comprises three spatial types, viz., **points**, **lines**, and **regions**.

A **points** value denotes a finite set of nodes with location (x, y) in the finite, discrete, two-dimensional sensor space defined by a WSN deployment as described in Section 3.2. We denote the location (x, y) of a node by writing the ID of the latter, e.g., s_1, on the assumption, discussed in Chapter 3, that every node is location-aware and knows which geometries it belongs to.

A *line segment* is a pair of distinct locations $\overline{s_1 s_2}$ in sensor space such that s_1 and s_2 are *closest* one-hop boundary neighbours. By *closest* one-hop boundary neighbours we mean that no neighbour belonging to same geometry, lies closer to s_1 or s_2, in the direction of the communication link between s_1 and s_2.

A **lines** value is a finite set of pairwise disjoint *line segment* values forming a connected sequence with no interior.

A **regions** value with no hole is defined as a finite set of pairwise disjoint polygons with an interior, boundary and exterior. More formally, a value of type **regions** without holes is a possibly singleton finite set of pairwise disjoint values of the form $\langle b, i\rangle$, where b denotes the *boundary* of a region, i denotes the possibly empty *interior* of that region, which contains all the nodes enclosed by b.

A *Unit* **regions** value is a **regions** value without interior. More formally, a *Unit* **regions** value is defined as a finite set of pairwise disjoint polygon values having an empty interior. The nodes comprising *unit* **regions** value are part of boundary. A *Minimal-unit* **regions** value is the smallest *unit* **regions** value. It is defined as a finite set of triangles, having an empty interior, i.e., comprising three segments that form a cycle, thereby implying that the three nodes are neighbours of each other.

A **regions** value with holes is defined as a finite set of pairwise disjoint triples of the form $\langle b, i, H\rangle$, where b is a *cycle* value denoting the *boundary* of the **regions** value, i denotes the *interior* of the **regions** value and contains all the nodes enclosed by b excepting those belonging to H, and H is a finite set of pairwise disjoint **regions** values, lying in the interior of b, each element of which denotes a hole in the region. Note that the nodes enclosed by an element of H *do not* belong to i, and hence not to the **regions** value either. The boundary and the exterior of a **regions** value with holes are allowed to be disconnected. Therefore, a **regions** value with holes has one outer boundary and one or more interior boundaries based upon the number of disjoint holes inside the outer boundary of the **regions** value. Each inner boundary delimits one hole. Holes represent the simple **regions** values that are not part of the **regions** value but are enclosed by it. Similarly, if a **regions** value has more than one hole, then the holes are `vertex_disjoint` with respect to each other, otherwise they form a single hole.

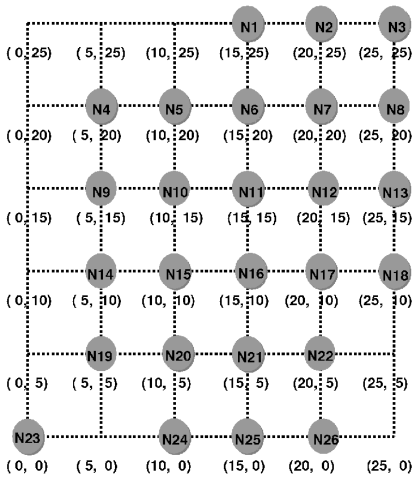
Each element r of a **regions** value partitions the space into the (possibly empty) set of points belonging to the interior of r, denoted by r_{in}, the set of points belonging to the boundary of r, denoted by r_{on}, and the set of points not belonging to either r_{in} or r_{on}, denoted by r_{out}. r_{on} consists of both the points lying on the outer boundary denoted by r_{oon}, and the interior boundaries denoted by $\bigcup_{h=1}^{k} r_{ion}^{h}$ (i.e. $r_{on} = \bigcup_{h=1}^{k} r_{ion}^{h} \cup r_{oon}$) where K denotes the number of disjoint holes. Thus, $r = r_{in} \cup r_{on}$.

If a non-empty value of type **points**, **lines**, or **regions** is a singleton, it is referred to as a single-element geometry (SEG), otherwise, it is referred to as a multi-element geometry (MEG). In the scope of this dissertation, unless it is specified specifically, a value of type **regions** may or may not have a hole, and it may or may not be a *unit* or a *minimal-unit* **regions** value.

Figure 4.1 (***a***) shows part of an example WSN and Figure 4.1 (***b***) shows some geometries defined over it as follows. `points{N18}` is a **points** value, call it `P`. `lines{N23-N19,-N19-N15,N15-N11,N11-N7,N7-N3}` is a **lines** value, call it `L`. `regions{{N11,N15,N20,N24,N25,N26,-N22,N17}:{N16,N21}:{ } }` is a **regions** value, call it `R1`, represented as a triple consisting of a boundary, an interior and in this case, an empty set of holes. Other **regions** values are `R2` = `regions{{N11,N12,N13,N8,N3,N2,N1,N6}:{N7}:{ } }` and `R3` = `regions{{N11,N6,N5,N4,N9,N14,-N15}:{N10}:{ } }`.

In Rose algebra, two representations of a region are set up, as a set of pairwise edge disjoint R-faces, and as a set of area-disjoint R-units. Operations faces and units are defined to convert between R-faces and R-units [GS95].

The formal definitions given are as follows:



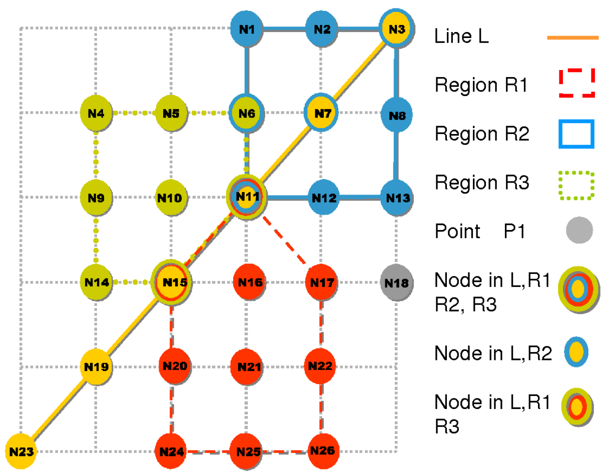


Figure 4.1: (a) An example WSN (b) Example geometries over (a)

$\forall F \subseteq F(R) : faces(units(F)) = F$

$units(F) := \;\; u \in U(R) | \exists f \in F : u$ `area_inside` f

Schneider and Guting have explained in [GS95], that the *faces* operation first forms multiset of boundary segments from a given set of area-disjoint R-units, then, all segments occurring twice are removed to compute the set of segments (that occurs once) that uniquely defines a set of edge-disjoint R-faces [GS95].

Furthermore, in [GS95] it is described as: *let T be a set of R-segments, that is, T ⊆ S. Then, cycles (T) denotes the set of all cycles (in the graph interpretation of realm R) that can be formed from segments in T.* Cycles defines the basic entities for the definitions of the **regions** value. According to the definition of R-cycle given by Schneider [GS93] the R-cycle may be empty. The reader is referred to Appendix A.1 for the description of R-cycle.

As described in Section 1.1, each sensor node based on its sensing and communication range covers a part of sensor network area. In addition, in Section 2.2 it was also described that based on application and the deployer requirements the nodes density varies. In this dissertation, the minimum-unit **regions** value is defined as cycle formed by three closest neighboring nodes as explained above. A single-element **regions** value which is not a minimum-unit **regions** value, is defined as comprising finite set of `area_disjoint` minimum-unit **regions** values.

4.2 Spatial Operations

This section presents the definition of the spatial operations over the spatial values in Section 4.1. The algebraic operations that are covered in this dissertation fall into two groups, viz., spatial-valued operations and Boolean-valued operations. The spatial data type values over which the operations are defined can denote induced, derived, or asserted geometries.

For the definition of operations, assume that p and p' denotes SEG **points** values, and P and P' denote MEG **points** values; l and l' denotes SEG **lines** values, and L and L' denote MEG **lines** values; r and r' denote SEG **regions** values representing values with or without holes or *unit* values, and R and R' denote MEG **regions** values whose constituents are SEGs. In addition, s, s', s'' and s''' denote node IDs, $\overline{ss'}$ denotes a segment between s and s', $\widetilde{ss'}$ denotes that s and s' are boundary neighbours of a **regions** value and form a boundary segment that does not intersect the interior of a **regions** value, and s $\in N_c(s')$ denotes that s' is among the *closest* neighbours of s. Recall, from Section 3.2.1, that $N(s_i)$ denotes all nodes that lie within the communication range of a node s_i. $N_c(s_i) \subseteq N(s_i)$ denotes the neighbouring nodes, where the communication link between s_i and $s_j \in N_c(s_i)$ does not have any other neighbouring node in between.

In the case of *unit* and *minimal-unit* **regions** values r, $r = r_{on}$. Let $s \in r_{on}r'_{on}$ denote that s belongs to the boundary of both r and r', and let $s \in r_{in}r'_{on}$ denote that s belongs to interior of r and the boundary of r'.

A *localized unit triangle* (LUT) $\langle s_1, s_2, s_3 \rangle$ satisfies the properties that the interior and the edges of the LUT do not contain any node that is a neighbour of s_1, s_2 or s_3 and that all the edges of LUT have length not greater than unit length (i.e., the prevailing radio range). The reader is referred to Section 5.4.2.1, for a description of the computation of LUT and CLUT. Let $MinUnit(r)$ denote the finite set of LUT in r, $MinUnit(rr')$ denote the finite set of *common localized unit triangles* (CLUTs) of r and r', and $MinUnit(r'_{on}r_{on})$ denote the finite set of CLUTs of r and r' formed by nodes that belongs to boundary of both r and r'. Let $s \in MinUnit(r_{on}r'_{on})$ denote the finite set of CLUTs of which s is a member.

4.2.1 Boolean-Valued Operations

The spatial operations in Table 4.1 are ***Boolean-valued operations***, that characterize topological relationships. To describe the operations in the algebra, second-order signatures [Sch97] are used as described in Appendix A.2.1. Operations defined in this section are the ones that are defined by Schneider [Sch97].

The definitions in [Sch97] for Boolean-valued operations have been improved upon to consider the cases in which one or both operands of type **regions** are *unit* **regions**. This section comprises six sub-sections presenting the definitions for operations over combinations of values of type **points**, **lines** and **regions**.

4.2.1.1 Operations on Points

This section defines the Boolean-valued operations that can be performed over values of type **points**. Formal definitions for spatial predicates on SEG **points** are given in Table 4.2. The definitions for these operations are created as part of the work described. Two values p and p' of type SEG **points** are considered `disjoint` and `not_equals` if a node that belongs to p does not belong to p'. Otherwise, if a node belongs to both p and p', then p `equals` p'.

The definitions of operations over **points** values of type MEG are given in Table 4.3. The definition of `disjoint` operation over the values of type MEG is given in [GS93]. Operation `equals` yields **true** if both P and P' have an equal number of elements and, for each element $p \in P$, there exists one unique element of P' with which it stands in an `equals` relationship. The `disjoint` relationship yields **true** if an element that belongs to P does not belongs to any element in P'. The `not_equals` relationship yields **true** if P is not `equals` to P'.

`equals`	: GEO $\times$ GEO	$\to \mathbb{B}$
`not_equals`	: GEO $\times$ GEO	$\to \mathbb{B}$
`intersects`	: $\text{EXT}_1 \times \text{EXT}_2$	$\to \mathbb{B}$
`disjoint`	: **points** $\times$ **points**	$\to \mathbb{B}$
`disjoint`	: **lines** $\times$ **lines**	$\to \mathbb{B}$
`vertex_disjoint`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`area_disjoint`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`edge_disjoint`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`adjacent`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`meets`	: $\text{EXT}_1 \times \text{EXT}_2$	$\to \mathbb{B}$
`area_inside`	: GEO $\times$ **regions**	$\to \mathbb{B}$
`edge_inside`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`vertex_inside`	: **regions** $\times$ **regions**	$\to \mathbb{B}$
`border_in_common`	: $\text{EXT}_1 \times \text{EXT}_2$	$\to \mathbb{B}$
`on_border_of`	: **points** $\times$ EXT	$\to \mathbb{B}$

Table 4.1: Spatial predicates supported

$$p\ \texttt{disjoint}\ p' \equiv \exists s[s \in p \Rightarrow s \notin p']$$

$$p\ \texttt{equals}\ p' \equiv \exists s[s \in p \Rightarrow s \in p']$$

$$p\ \texttt{not\_equals}\ p' \equiv \neg(p\ \texttt{equals}\ p')$$

Table 4.2: Formal definitions for spatial predicates on SEG **points**

$$P\ \texttt{disjoint}\ P' \equiv \forall p \in P\, \forall p' \in P' : p\ \texttt{disjoint}\ p'$$

$$P\ \texttt{equals}\ P' \equiv \forall p \in P\, \exists! p' \in P' : p\ \texttt{equals}\ p' \wedge \forall p' \in P'\, \exists! p \in P : p'\ \texttt{equals}\ p$$

$$P\ \texttt{not\_equals}\ P' \equiv \neg(P\ \texttt{equals}\ P')$$

Table 4.3: Formal definitions for spatial predicates on MEG **points**

4.2.1.2 Operations on Points and Lines

This section defines the Boolean-valued operations that can be performed over values of type **points** and **lines**. The definitions of operations over **points** and **lines** values of type SEG are given in Table 4.4. The definitions for these operations are created as part of the work described. The operation `on_border_of` yields **true** if a value p intersects a point that belongs to a line-segment in l.

The definitions of operations over **points** and **lines** values of type MEG are given in Table 4.5. The definition of `on_border_of` operation over the values of type MEG is given in [GS93].

$$p\ \texttt{on\_border\_of}\ l \equiv \exists ss'[\overline{ss'} \in l \Rightarrow s \in p \vee s' \in p]$$

Table 4.4: Formal definitions for spatial predicates on SEG **points** and **regions**

$$P \texttt{ on\_border\_of } L \equiv \forall p \in P\, \exists l \in L : p \texttt{ on\_border\_of } l$$

Table 4.5: Formal definitions for spatial predicates on MEG **points** and **regions**

4.2.1.3 Operations on Points and Regions

This section defines the Boolean-valued operations that can be performed over values of type **points** and **regions**. The definitions of operations over **points** and **regions** values of type SEG are given in Table 4.6. The definitions for these operations are created as part of the work described.

The definition of operation `area_inside` over the values of type SEG is given in [GS93], whereas the definition of `on_border_of` operation in Table 4.6 is created as part of the work described. The definition for the `area_inside` operation is created differently. The reason is as explained in Section 4.1 nodes enclosed by a hole inside a **regions** value, do not belong to interior of the **regions** value, and hence not to the **regions** value either. The operation `on_border_of` returns **true** if a boundary point of r belongs to the SEG **points** value p. For an `area_inside` relationship between the SEG **points** value p and the **regions** value r, the value p must also belong to r.

$$p \texttt{ on\_border\_of } r \equiv \exists s[s \in p \Rightarrow s \in r_{on}]$$

$$p \texttt{ area\_inside } r \equiv \exists s[s \in p \Rightarrow s \in r]$$

Table 4.6: Formal definitions for spatial predicates on SEG **points** and **regions**

In Table 4.7, formal definitions of operations over the MEG values of type **points** and **regions** are given. The definition of `on_border_of` operation over the values of type SEG is given in [GS93], whereas the definition of `area_inside` operation in Table 4.7 is created as part of the work described. For an `on_border_of` relationship, for each element of P, there must exist an element of R which it is `on_border_of`. For an `area_inside` relationship between the MEG **points** value P and **regions** value R, for each element of P, there must exist an element of R it is `area_inside` of.

$$P \texttt{ on\_border\_of } R \equiv \forall p \in P\, \exists r \in R : p \texttt{ on\_border\_of } r$$

$$P \texttt{ area\_inside } R \equiv \forall p \in P\, \exists r \in R : p \texttt{ area\_inside } r$$

Table 4.7: Formal definitions for spatial predicates on MEG **points** and **regions**

4.2.1.4 Operations on Lines

This section defines the Boolean-valued operations that can be performed over values of type **lines**. The definitions of operations over the **lines** values of type SEG are given in Table 4.8. The definitions of operations `meets` and `intersects` over the values of type SEG are given in

[GS93], whereas the definitions of all other operations in Table 4.8 are created as part of the work described. The definitions of operations `meets` and `intersects` are created differently, to cater for the $=$ and $\neq$ operations that are used in the definitions in [GS93], but not well explained.

In Table 4.8, for computing whether a sensor node s that belongs to both values l and l' and is not a part of common line-segment is a meeting point, s needs to compute angularly sorted cyclic list of all segments touching at s of each value. If only one segment of either l or l', or both, meet at s, then s is a *meeting point*. Node s is also considered as a meeting point, if upon finding that segments list is the circular concatenation of two sublists such that one sublist belongs to l and the other to l'. For `meets` or `intersects`, both l and l' must not have any line segment in common. In addition, for `meets`, the point where l and l' touches must be a meeting point, whereas `intersects` requires that the point where they touch must not be a meeting point. SEG **lines** values l and l' are `disjoint`, if any node that belongs to l does not belongs to l'. The `border_in_common` operation yields **true** if one or more line segments that belong to l also belong to l'.

l `disjoint` l'	$\equiv$	$\forall s[s \in l \Rightarrow s \notin l']$
l `equals` l'	$\equiv$	$\forall ss'[\overline{ss'} \wedge \overline{ss'} \in l \Rightarrow \overline{ss'} \in l'] \wedge$ $\forall s''s'''[\overline{s''s'''} \wedge \overline{s''s'''} \in l' \Rightarrow \overline{s''s'''} \in l]$
l `not_equals` l'	$\equiv$	$\neg(l$ `equals` $l')$
l `border_in_common` l'	$\equiv$	$\exists s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \in l']$
l `meets` l'	$\equiv$	$\forall s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \notin l'] \wedge$ $\exists s[s \in l \wedge s \in l' \wedge \mathit{meetingpoint}(s)]$
l `intersects` l'	$\equiv$	$\forall s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \notin l'] \wedge$ $\exists s[s \in l \wedge s \in l' \wedge \neg \mathit{meetingpoint}(s)]$

Table 4.8: Formal definitions for spatial predicates on SEG **lines**

The definitions of operations over the **lines** values of type MEG are given in Table 4.9. The definitions of operations `border_in_common`, `meets` and `intersects` over the values of type MEG are given in [GS93], whereas the definitions of all other operations in Table 4.9 are created as part of the work described.

4.2.1.5 Operations on Lines and Regions

This section defines the Boolean-valued operations that can be performed over values of type **regions** and **lines**. The definitions of operations over the **lines** and **regions** values of type SEG are given in Table 4.10 and over **lines** and **regions** values of type MEG in Table 4.11.

The definitions of `meets` and `intersects` operations over the values of type SEG are given in [GS93], whereas the definitions of all other operations in Table 4.8 are created as part of the work described. The definitions of operations `area_disjoint`, `border_in_common` and `adjacent`

$L\ \texttt{equals}\ L'$	$\equiv$	$\forall l \in L\, \exists! l' \in L' : l\ \texttt{equals}\ l'\ \wedge$ $\forall l' \in L'\, \exists! l \in L : l'\ \texttt{equals}\ l$
$L\ \texttt{not\_equals}\ L'$	$\equiv$	$\neg(L\ \texttt{equals}\ L')$
$L\ \texttt{intersects}\ L'$	$\equiv$	$(\forall l \in L\ \forall l' \in L' :\ l\ \texttt{intersects}\ l' \vee l\ \texttt{disjoint}\ l')\wedge$ $\exists l \in L\, \exists l' \in L' : l\ \texttt{intersects}\ l'$
$L\ \texttt{disjoint}\ L'$	$\equiv$	$\forall l \in L\, \forall l \in L' : l\ \texttt{disjoint}\ l'$
$L\ \texttt{meets}\ L'$	$\equiv$	$(\forall l \in L\ \forall l' \in L' :\ l\ \texttt{meets}\ l' \vee l\ \texttt{disjoint}\ l')\wedge$ $\exists l \in L\, \exists l' \in L' : l\ \texttt{meets}\ l'$
$L\ \texttt{border\_in\_common}\ L'$	$\equiv$	$\exists l \in L\, \exists l' \in L' : l\ \texttt{border\_in\_common}\ l'$

Table 4.9: Formal definitions for spatial predicates on MEG **lines**

over the values of type MEG are given in [GS93], whereas the definitions of all other operations in Table 4.8 are created as part of the work described.

$l\ \texttt{disjoint}\ r$	$\equiv$	$\forall s[(s \in r \Rightarrow s \notin l)]$
$l\ \texttt{area\_inside}\ r$	$\equiv$	$\forall s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \in r]$
$l\ \texttt{border\_in\_common}\ r$	$\equiv$	$\exists s, s'[\overline{ss'} \in l \wedge \widetilde{ss'} \in r_{on}]$
$l\ \texttt{meets}\ r$	$\equiv$	$\forall s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \neg(\overline{ss'}\ \texttt{area\_inside}\ r)]\wedge$ $\exists s[s \in l \wedge s \in r_{on} \wedge meetingpoint(s)]$
$l\ \texttt{intersects}\ r$	$\equiv$	$\exists s, s'[\overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'}\ \texttt{area\_inside}\ r]$

Table 4.10: Formal definitions for spatial predicates on SEG **lines** and **regions**

$L\ \texttt{intersects}\ R$	$\equiv$	$\exists l \in L\, \exists r \in R : l\ \texttt{intersects}\ r$
$L\ \texttt{disjoint}\ R$	$\equiv$	$\forall l \in L\, \forall r \in R : l\ \texttt{vertex\_disjoint}\ r$
$L\ \texttt{meets}\ R$	$\equiv$	$\forall l \in L\, \forall r \in R : \neg(l\ \texttt{area\_inside}\ r')\ \wedge$ $\exists l \in L\, \exists r \in R : l\ \texttt{meets}\ r$
$L\ \texttt{area\_inside}\ R$	$\equiv$	$\forall l \in L\, \exists r \in R : l\ \texttt{area\_inside}\ r$
$L\ \texttt{border\_in\_common}\ R$	$\equiv$	$\exists l \in L\, \exists r \in R : l\ \texttt{border\_in\_common}\ r$

Table 4.11: Formal definitions for spatial predicates on MEG **lines** and **regions**

4.2.1.6 Operations on Regions

This section defines the Boolean-valued operations that can be performed over values of type **regions**. The definitions over the **regions** values of type SEG are described first. The definitions for Boolean operators over the **regions** values of type MEG are provided later in this section.

The definitions of `equals`, `not_equals`, `intersects`, and `border_in_common` operations over the values of type SEG are created as part of the work described, whereas the definitions of all other operations in Table 4.12 are given in [GS93]. For `area_inside` operation, the definition given in [GS93] makes use of the operation $\subseteq$, not well explained. Therefore, the definition has been modified as it is an important operation and the definitions of `vertex_inside`, and `edge_inside` relationship depend on it. The original definition of `area_disjoint` operation has been modified, explained later in this section. The `area_disjoint` operation is an important operation because the definitions of `meets`, `adjacent`, `edge_disjoint`, and `vertex_disjoint` relationship depend on it.

Recall that, as explained in Section 4.1 nodes enclosed by a hole inside a **regions** value, do not belong to interior of the **regions** value, and hence not to the **regions** value either. A node is only part of a **regions** value if it is part of its interior or if the node lies on its boundary. So, in this dissertation, by a geometry r of type **regions** we mean the nodes that belong to the interior or boundary of r.

Formal definitions for spatial predicates on SEG values of type **regions** are given in Table 4.12. In the description of these predicates, it will help understanding if they are read in conjunction with Figures 4.2-4.9, in which it is assumed that the nodes in these figures are say, 10 meters apart and the radio range is 15 meters. Therefore, the maximum neighbourhood of a node contains 8 nodes.

Two regions r and r' are `area_disjoint` if points lying in the interior of either value do not intersect any point in the other value, and the values do not share any *minimal-unit* area. Both values may share common boundary points and boundary segments.

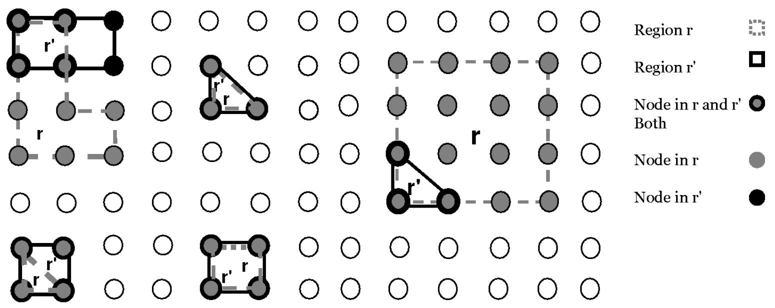


Figure 4.2: Examples of not having `area_disjoint` relationship between r and r' of type SEG **regions**.

If both values r and r' have interior points, then the definition given for the `area_disjoint` operator in [Sch97] and reproduced in Table A.1 in Appendix A.2.1 suffices. According to that definition, a point belonging to the interior of either cycle value must not intersect any point of

r `area_disjoint` r'	$\equiv$	$\forall s[(s \in r \Rightarrow s \notin r') \vee (s \in r_{on} \wedge s \in r'_{on})]$ $\wedge[MinUnit(r) \cap MinUnit(r') = \emptyset]$
r `edge_disjoint` r'	$\equiv$	r `area_disjoint` $r' \wedge \forall s, s'[s, s' \in r_{on} \wedge s, s' \in r'_{on} \Rightarrow \neg\widetilde{ss'}]$
r `vertex_disjoint` r'	$\equiv$	$\forall s[(s \in r \Rightarrow s \notin r')]$
r `intersects` r'	$\equiv$	$\exists s[((s \mid s \in r_{in} \wedge s \in r') \vee (s \mid s \in r \wedge s \in r'_{in}))] \wedge$ $[MinUnit(r) \cap MinUnit(r') \neq \emptyset]$
r `adjacent` r'	$\equiv$	r `area_disjoint` $r' \wedge\ \exists s, s'[s, s' \in r_{on} \wedge s, s' \in r'_{on} \wedge \widetilde{ss'}]$
r `meets` r'	$\equiv$	r `area_disjoint` $r' \wedge\ \exists s[s \in r_{on} \wedge s \in r'_{on}] \wedge$ $\forall s's''[s', s'' \in r_{on} \wedge s's'' \in r'_{on} \wedge \neg\widetilde{s's''}]$
r `area_inside` r'	$\equiv$	$\forall s(s \in r_{in} \Rightarrow s \in r'_{in}) \wedge \forall s'(s' \in r_{on} \Rightarrow s \in r')$ $\wedge\ ((r$ `equals` $r') \vee (\exists s''(s'' \notin r \wedge s'' \in r'))$
r `edge_inside` r'	$\equiv$	r `area_inside` $r' \wedge \forall s, s'[s, s' \in r_{on} \wedge s, s' \in r'_{on} \Rightarrow \neg\widetilde{ss'}]$
r `vertex_inside` r'	$\equiv$	r `edge_inside` $r' \wedge \forall s(s \in r_{on} \Rightarrow s \notin r'_{on})]$
r `equals` r'	$\equiv$	$\forall s[s \in r_{in} \Rightarrow s \in r'_{in}] \wedge \forall s'[s' \in r_{on} \Rightarrow s' \in r'_{on}] \wedge$ $\forall s''[(s'' \in r'_{in} \Rightarrow s'' \in r_{in}] \wedge \forall s'''[(s''' \in r'_{on} \Rightarrow s''' \in r_{on}] \wedge$ $\forall s''''s'''''[\widetilde{s''''s'''''} \in r_{on} \Rightarrow \widetilde{s''''s'''''} \in r'_{on}] \wedge$ $\forall s''''s'''''[\widetilde{s''''s'''''} \in r'_{on} \Rightarrow \widetilde{s''''s'''''} \in r_{on}]$
r `not_equals` r'	$\equiv$	$\neg(r$ `equals` $r')$
r `border_in_common` r'	$\equiv$	$\exists s, s'[s, s' \in r_{on} \wedge s, s' \in r'_{on} \wedge \widetilde{ss'}]$

Table 4.12: Formal definitions for spatial predicates on SEG **regions**

the other value. This definition cannot handle the case where one or both SEG **regions** values are *unit* or *minimal-unit* **regions**. Some example scenarios are given in the Figure 4.2 where the original definition of `area_disjoint` cannot determine whether the geometries are `area_disjoint`. Given these shortcomings, the original definition of `area_disjoint` operation has been extended by adding the condition that the two values must not have any shared *minimal-unit* **regions** area. This additional condition handles the scenarios in Figure 4.2. Two values r and r' are also considered to be `area_disjoint` if there exists a hole in r' such that r lies `area_inside` it, or if there exists a hole in r such that r' lies `area_inside` it.

Figure 4.3-4.5, show example scenarios where SEG **regions** values are `area_disjoint`. In Figure 4.3(a), both r and r' are SEG **regions** values without holes and r and r' only share boundary segments. In Figure 4.3(b), r' is a SEG **regions** value with a hole, r is a SEG **regions** value without hole, and r shares a boundary segment with r' and is `area_inside` the hole in r'. In Figure 4.3(c), r' is a SEG **regions** value with a hole and r is a *unit* **regions** value, and r shares two boundary segments with r' and is `area_inside` the hole in r'.

In Figure 4.4(a), both r and r' are SEG **regions** values without holes, r and r' are `area_disjoint` and r shares two boundary points with r'. In Figure 4.4(b), r is a *unit* **regions** value and r' is a SEG **regions** value with a hole, r shares a boundary point with r' and is `edge_inside` the hole in r'. In Figure 4.4(c), r' is a SEG **regions** value with a hole and r is a SEG **regions** value without a hole, r shares two boundary points with r' and is `edge_inside` the hole in r'.

In Figure 4.5(a), both r and r' are SEG **regions** values with an interior, r does not share any point with r'. In Figure 4.5(b), r is a *unit* **regions** value and r' is a SEG **regions** value with a hole, r and r' do not share any point and r is `vertex_inside` the hole in r'. In Figure 4.5(c), r' and r are SEG **regions** with and without a hole respectively, r and r' do not share any point and r is `vertex_inside` the hole in r'.

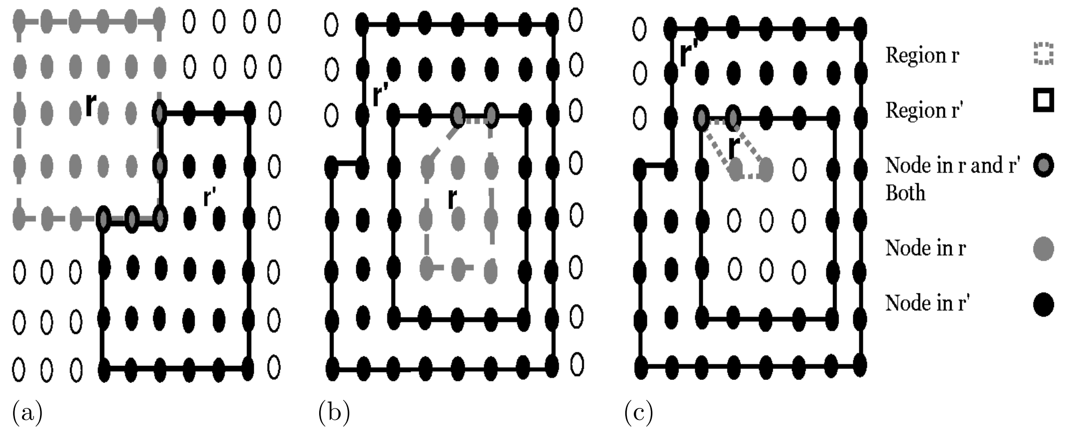


Figure 4.3: Geometries r and r' of type SEG **regions** standing in `area_disjoint`, `border_in_common` and `adjacent` relationships.

The `edge_disjoint` relationship is **true** for two geometries r and r' if they are `area_disjoint`, and do not share common boundary segments, although they may share common boundary points. In addition, two SEG **regions** values r' and r are `edge_disjoint` if there exists a hole in r such that r' is `edge_inside` in it, or if there exists a hole in r' such that r is `edge_inside` in it. Figures 4.4-4.5 show example scenarios where the two **regions** values are `edge_disjoint`.

The `vertex_disjoint` relationship simply `disjoint`, is true if values r and r' do not share any points. The values are also considered to be `vertex_disjoint` if there exists a hole in r' such that r is `vertex_inside` in it or if there exists a hole in r such that r' is `vertex_inside` in it. Figure 4.5 shows example scenarios where **regions** values are `vertex_disjoint`.

The `meets` relationship is true if the values are `area_disjoint`, and share at least one common boundary point and no common boundary segment. In addition, the SEG **regions** values r' and r stand in a `meets` relationship if there exists a hole in r such that r' is `edge_inside` it and has at least one common boundary point with r and no common boundary segment, or else if there exists a hole in r' such that r is `edge_inside` in it and has at least one common boundary point with r' and no common boundary segment. Figure 4.4 shows example scenarios where two SEG **regions** values stand in a `meets` relationship.

The `adjacent` relationship is true if the values are `area_disjoint`, and shares at least one common boundary segment. Two **regions** values r' and r are `adjacent` if there exists a hole

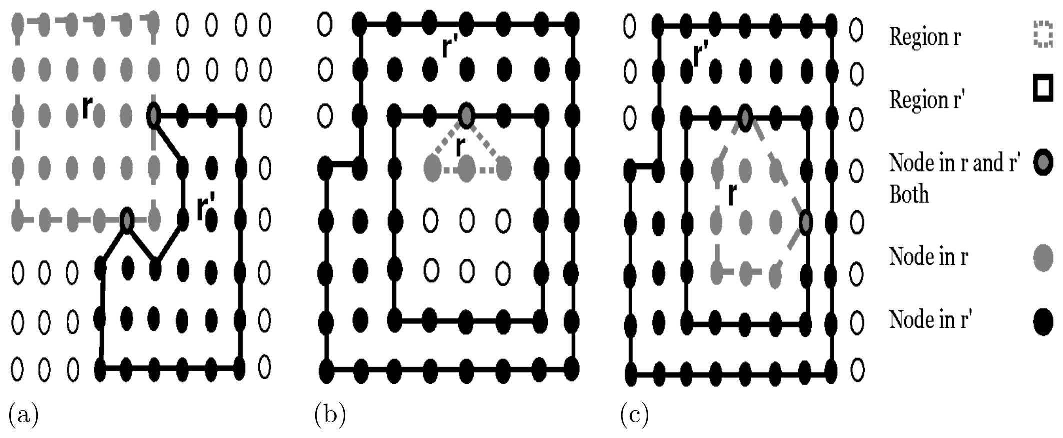


Figure 4.4: Geometries r and r' of type SEG **regions** standing in `edge_disjoint` and `meets` relationships.

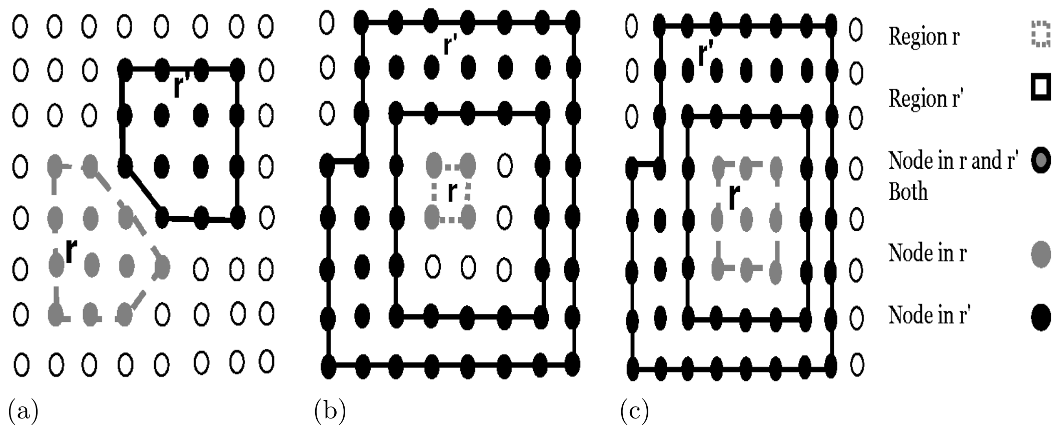


Figure 4.5: Geometries r and r' of type SEG **regions** standing in `edge_disjoint` and `vertex_disjoint` relationships.

in r such that r' is `area_inside` it and there exists at least one boundary segment in common with r, or if there exists a hole in r' such that r is `area_inside` it and there exists at least one common boundary segment with r'. Figure 4.3 shows example scenarios where SEG **regions** values are `adjacent`.

The `border_in_common` relationship is true if the two **regions** values have at least one boundary segment in common. Figure 4.3 and Figure 4.6 show example scenarios where the two SEG **regions** have a `border_in_common`.

Geometry r is `area_inside` r' if r `equals` r' or r is subset of r'. They may or may not share a common boundary point and common boundary segment. In addition, the `area_inside` relationship is also **true** if r is `area_disjoint` from all holes in r' or there exist holes in r' with holes in r `area_inside` it. Figure 4.6-4.8 show example scenarios where SEG **regions** values are `area_inside` one another.

Two values are `edge_inside` if they are `area_inside` and, in addition, do not share any boundary segment. The `edge_inside` relationship is also **true**, if r is `edge_disjoint` from all holes in r' or there exists hole in r such that there is a hole in r' that `edge_inside` it. Figure

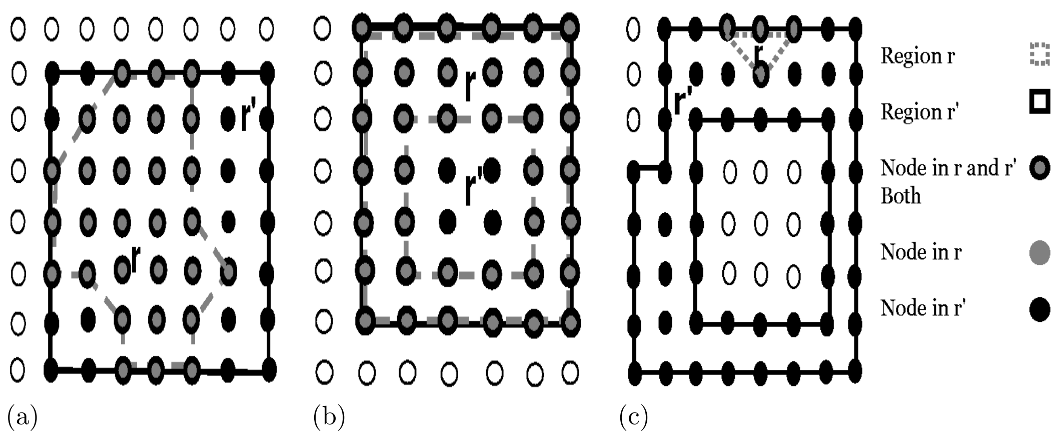


Figure 4.6: Geometries r and r' of type SEG **regions** standing in `area_inside` and `border_in_common` relationships.

4.7-4.8 show example scenarios of SEG **regions** values that are `edge_inside` one another.

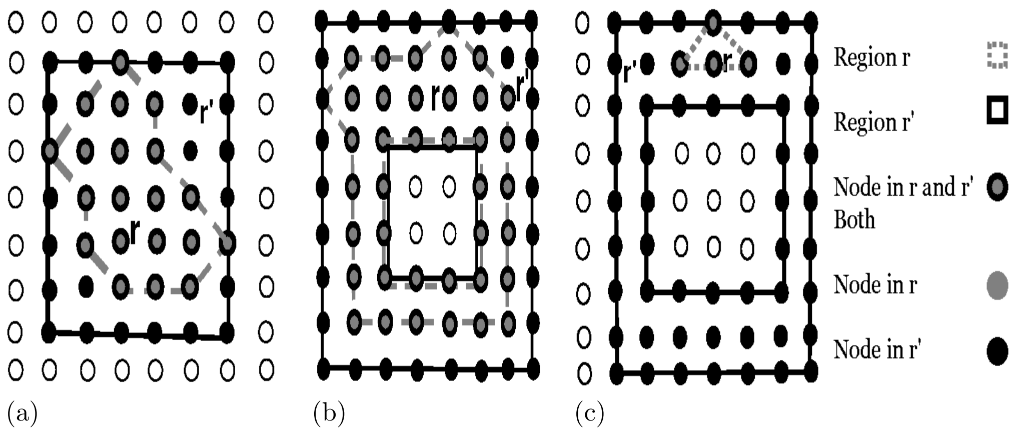


Figure 4.7: Geometries r and r' of type SEG **regions** standing in `area_inside`, `edge_inside`, and `not_equals` relationships.

The `vertex_inside` relationship is **true** if r and r' are `edge_inside` and do have any common boundary point. The `vertex_inside` relationship is also **true** if r is `vertex_disjoint` from all holes in r' or there exist holes in r such that there is a hole in r' that is `vertex_inside` it. Figure 4.8 shows example scenarios of **regions** values that are `vertex_inside` one another.

The relationship `intersects` is **true** if the operands have a *minimal-unit* **regions** value in common or else if the interior of either value shares a boundary and the interior with the other value, or the interior and boundary of either value shares a interior of the other value, or the interior and boundary of either value shares the boundary and interior with the other value. Figure 4.7-4.8 show example scenarios of SEG **regions** values that stand in an `intersects` relationship.

r `equals` r' if both values have an equal number of points and boundary segments, and each point that belongs to the interior of one value also belongs to the interior of the other value and each point that belongs to the boundary of one value also belongs to boundary of the other

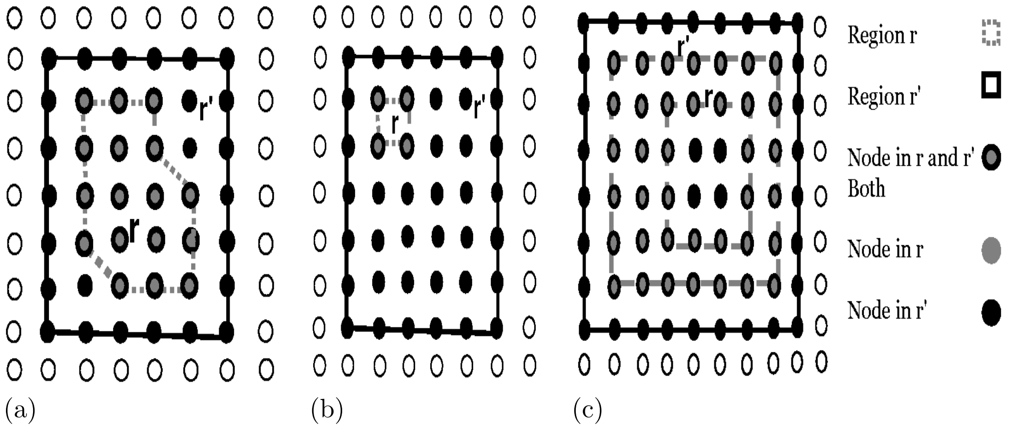


Figure 4.8: Geometries r and r' of type SEG **regions** standing in `area_inside`, `edge_inside`, `vertex_inside`, and `not_equals` relationships.

value. If r and r' are SEG **regions** value with holes, both r' and r must have an equal number of holes and for each hole in r', there must exist one hole in r which equals it. Figure 4.9 shows example scenarios of **regions** values that stand in a `equals` relationship. In Figure 4.9(a), r and r' are SEG **regions** values without holes and r and r' share every interior and boundary point. In Figure 4.9(b), both r and r' are SEG **regions** values with a hole that share every interior and boundary point and have an equal number of holes. In Figure 4.9(c), r and r' are SEG *unit* **regions** values and r and r' share every boundary point.

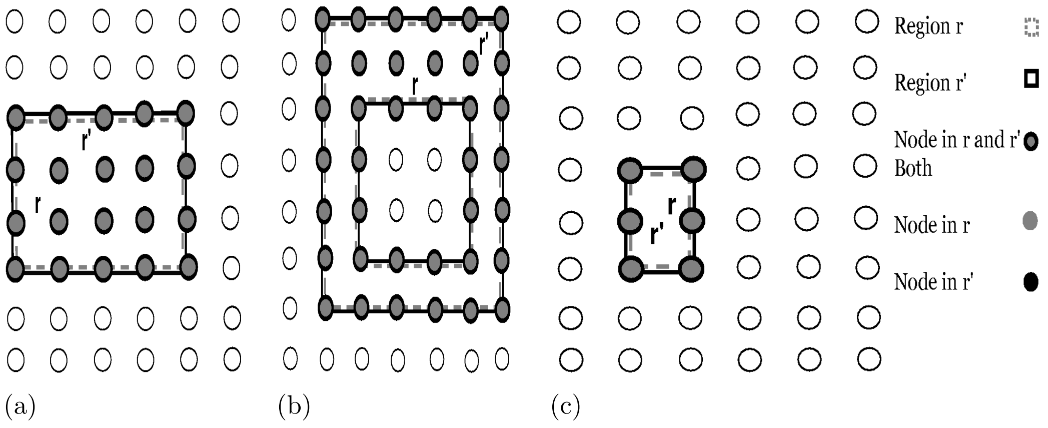


Figure 4.9: Geometries r and r' of type SEG **regions** standing in `equals` relationships.

The relationship `not_equals` is **true** if both SEG **regions** values are not `equals`, i.e., at least one point exists that is part of the interior or the boundary of one value and not of the other or atleast one boundary segment exists that is part of one value and not of other value.

Figure 4.13 presents definitions for spatial predicates over operands of type MEG **regions**. The definitions of `area_disjoint`, `border_in_common`, `intersects` and `adjacent` operations over the values of type MEG are given in [GS93], whereas the definitions of all other operations in Table 4.8 are created as part of the work described. The definition of `intersects` operation given in [GS93] has been modified, as the function *Units*() used in the definition is not well

R `equals` R'	$\equiv$	$\forall r \in R\, \exists! r' \in R' : r$`equals` $r' \wedge$ $\forall r' \in R'\, \exists! r \in R : r'$`equals` r
R `not_equals` R'	$\equiv$	$\neg(R$ `equals` $R')$
R `area_disjoint` R'	$\equiv$	$\forall r \in R \forall r' \in R' : r$ `area_disjoint` r'
R `edge_disjoint` R'	$\equiv$	$\forall r \in R \forall r' \in R' : r$ `edge_disjoint` r'
R `vertex_disjoint` R'	$\equiv$	$\forall r \in R \forall r' \in R' : r$ `vertex_disjoint` r'
R `adjacent` R'	$\equiv$	R `area_disjoint` $R' \wedge \exists r \in R\, \exists r' \in R' : r$ `adjacent` r'
R `meets` R'	$\equiv$	R `area_disjoint` $R' \wedge \exists r \in R\, \exists r' \in R' : r$ `meets` r'
R `area_inside` R'	$\equiv$	$\forall r \in R\, \exists r' \in R' : r$ `area_inside` r'
R `edge_inside` R'	$\equiv$	$\forall r \in R\, \exists r' \in R' : r$ `edge_inside` r'
R `vertex_inside` R'	$\equiv$	$\forall r \in R\, \exists r' \in R' : r$ `vertex_inside` r'
R `border_in_common` R'	$\equiv$	$\exists r \in R\, \exists r' \in R' : r$ `border_in_common` r'
R `intersects` R'	$\equiv$	$\exists r \in R\, \exists r' \in R' : r$ `intersects` r'

Table 4.13: Formal definitions for spatial predicates on MEG **regions**

explained.

The operation `equals` yields **true** if both R and R' have an equal number of elements and, for each element of $r \in R$, there exists one element of R' with which it has `equals` relationship. Operation `not_equals` yields **true** if the `equals` relationship between R and R' yields **false**.

The operation `area_disjoint` yields **true** if every possible pair of elements from both geometries R and R' are `area_disjoint`. For `edge_disjoint`, all elements of R and all elements of R' must be `edge_disjoint` with each other. Similarly, for `vertex_disjoint` every possible pair of values from both geometries R and R' must be `vertex_disjoint`.

The two values R and R' of type MEG **regions** are `adjacent`, if every pair of the elements from R and R' is `area_disjoint` and there exists an element of R `adjacent` with an element of R'.

The two values R and R' of type MEG **regions** stand in a `meets` relationship if every element of R is `area_disjoint` with every element of R', and there exists an element of R that stands in a `meets` relationship with an element of R'.

For `area_inside`, all elements of R must be `area_inside` one or more elements of R'. R and R' are `edge_inside` if all the elements of R are `edge_inside` one or more elements of R'. Operation `vertex_inside` yields **true** if all the elements of R are `vertex_inside` one or more elements of R'.

For `border_in_common`, there must exist an element of R sharing at least one common boundary segment with an element of R'. The operation `intersects` is **true** if there exists an element of R that stands in an `intersects` relationship with an element of R'.

4.2.2 Spatial-Valued Operations

The second group of operations in the algebra comprises ***spatial-valued operations***, i.e., those that return a derived geometry. It includes operations such as `plus`, `intersection`, `minus` and `contour`. These are shown in Table 4.14. For the operations defined in this section, equivalent operations exists in [Sch97].

`intersection`	: **points** $\times$ **points**	$\rightarrow$	**points**
`intersection`	: **lines** $\times$ **lines**	$\rightarrow$	**points**
`intersection`	: **regions** $\times$ **regions**	$\rightarrow$	**regions**
`intersection`	: **regions** $\times$ **lines**	$\rightarrow$	**lines**
`contour`	: **regions**	$\rightarrow$	**lines**
`plus`	: GEO $\times$ GEO	$\rightarrow$	GEO
`minus`	: GEO $\times$ GEO	$\rightarrow$	GEO
`vertices`	: EXT	$\rightarrow$	**points**
`common_border`	: $\text{EXT}_1 \times \text{EXT}_2$	$\rightarrow$	**lines**

Table 4.14: Spatial-valued operations supported

4.2.2.1 Operations on Points

Table 4.15 gives the definitions for spatial-valued operations over SEGs of type **points** and Table 4.16 gives definitions over operands of type MEG **points**. The definitions for these operations are created as part of the work described. The operator `plus` creates a new derived geometry containing the values of type **points** that belongs to either of the operands or to both. The operator `minus` returns a new derived geometry containing the values of type **points** that only belong to the geometry p. The operator `intersection` returns a new geometry of type **points** containing the values that belong to both p and p'.

p `plus` p'	$\equiv$	$\{s \mid s \in p \vee s \in p'\}$
p `intersection` p'	$\equiv$	$\{s \mid s \in p \wedge s \in p'\}$
p `minus` p'	$\equiv$	$\{s \mid s \in p \wedge s \notin p'\}$

Table 4.15: Spatial-valued operations on SEG **points**

P `plus` P'	$\equiv$	$\forall p \in P$ `plus` $\forall p' \in P'$
P `intersection` P'	$\equiv$	$\forall p \in P$ `intersection` $\forall p' \in P'$
P `minus` P'	$\equiv$	$\forall p \in P$ `minus` $\forall p' \in P'$

Table 4.16: Formal definitions for spatial-valued operations on MEG **points**

4.2.2.2 Operations on Lines

Table 4.17 gives the definitions for spatial-valued operations over SEGs operands of type **lines**. The definition of `intersection` operation over the values of type SEG is given in [GS93], whereas the definitions of all other operations in Table 4.17 are created as part of the work described.

The operator `plus` creates a new derived geometry containing the line-segments that belong to either l, l' or both. The operator `minus` returns a new derived geometry containing the line-segments that belong to value l only. The operator `intersection` returns a new derived geometry of type **points** containing the points where l `intersects` l'. The operator `vertices` returns a new derived geometry of type **points** containing the finite set of points that occurs in the line segments of l.

l `plus` l'	$\equiv$	$\{ss' \mid \overline{ss'} \wedge (\overline{ss'} \in l \vee \overline{ss'} \in l')\}$
l `minus` l'	$\equiv$	$\{ss' \mid \overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \notin l'\}$
l `common_border` l'	$\equiv$	$\{ss' \mid \overline{ss'} \wedge \overline{ss'} \in l \wedge \overline{ss'} \in l'\}$
l `intersection` l'	$\equiv$	$\{s \mid s \in l \wedge s \in l' \wedge \neg meetingpoint(s)\}$
`vertices` l	$\equiv$	$\{s \mid s \in l\}$

Table 4.17: Spatial-valued operations on SEG **lines**

4.2.2.3 Operations on Lines and Regions

Table 4.18 gives the definitions for spatial-valued operations over SEGs operands of type **lines** and **regions**. The definitions are given in [GS93].

The operation `intersection` returns a new derived geometry of type **lines** containing the line segments that are lies `area_inside` the SEG **regions** value r. The operation `common_border` returns a new derived geometry of type **lines** containing the finite set of line segments that belong to l and to the boundary of r.

l `common_border` r	$\equiv$	$\{ss' \mid \overline{ss'} \in l \wedge \widetilde{ss'} \in r_{on}\}$
l `intersection` r	$\equiv$	$\{ss' \mid \overline{ss'} \in l \wedge \overline{ss'}$ `area_inside` $r\}$

Table 4.18: Spatial-valued operations on SEG **lines** and **regions**

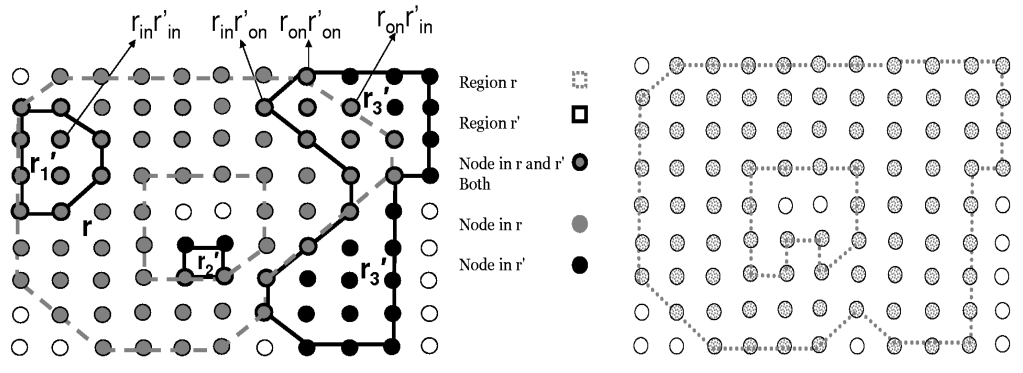
4.2.2.4 Operations on Regions

Table 4.19 gives the definitions for spatial-valued operations over SEGs of type **regions**. The definitions of `common_border`, and `contour` operations over the values of type SEG are given in [GS93], whereas the definitions of all other operations in Table 4.17 are created as part of the work described.

$$\begin{aligned}
r\ \texttt{plus}\ r' &\equiv \{s \mid s \in r \vee s \in r'\} \\[1ex]
r\ \texttt{intersection}\ r' &\equiv \{s \mid s \in r_{in} \wedge s \in r'\} \cup \{s \mid s \in r \wedge s \in r'_{in}\} \cup \\
&\quad \{s \mid s \in r_{on}r'_{on} \wedge \exists s'[s' \in r'_{in}r_{in} \vee s' \in r'_{on}r_{in} \vee s' \in r'_{in}r_{on})\} \\
&\quad \cup \{ss''s''' \mid ss''s''' \in r_{on}r'_{on} \wedge ss''s''' \in MinUnit(r_{on}r'_{on})\} \\[1ex]
r\ \texttt{minus}\ r' &\equiv \{s \mid s \in r \wedge s \notin r'\} \cup \{s \mid s \in r_{in} \wedge s \in r'_{on}\} \cup \\
&\quad \{s \mid s \in r_{on}r'_{on} \wedge \exists s'[s' \in r] \wedge s \notin MinUnit(r_{on}r'_{on})\} \\
&\quad \cup\{s \mid s \in r_{on}r'_{on} \wedge s \in MinUnit(r_{on}r'_{on}) \wedge \\
&\quad \exists s'[s' \in r \wedge \overline{ss'} \cap s \in MinUnit(r_{on}r'_{on}) = \phi]\} \\[1ex]
\texttt{contour}(\mathrm{r}) &\equiv \{s \mid s \in r_{oon} \wedge s \notin r_{ion}\} \\[1ex]
\texttt{vertices}(\mathrm{r}) &\equiv \{s \mid s \in r_{on}\} \\[1ex]
r\ \texttt{common\_border}\ r' &\equiv \{\ s, s' \mid s, s' \in r_{on} \wedge s, s' \in r'_{on} \wedge \widetilde{ss'}\}
\end{aligned}$$

Table 4.19: Spatial-valued operations on SEG **regions**

The operation `plus` creates a new derived geometry of type **regions** containing the points and segments that belong to one operand or to both. The reader is referred to Section 5.5.2.1, for a description of the computation of edge nodes and boundary segments of the derived geometry. Figure 4.10 explains the `plus` operation between two **regions** values R and R' and the resultant derived geometry, where R is of type SEG comprising r (a **regions** value with a hole) and R' is of type MEG comprising r'_1,r'_2 and r'_3.



(a) (b) Result of R `plus` R'

Figure 4.10: Derived geometry obtained as a result of a `plus` operation between **regions** values.

The operation `minus` between two **regions** values creates a new derived geometry of type **regions**. The new derived geometry contains the points that are members of r only; those belonging to the interior of r and to the boundary of r'; the common boundary points that do not form a CLUT with other common boundary points and that have among their closest neighbours at least one neighbour that is a member of r only; and the common boundary points that are part of one or more CLUTs and have among their closest neighbours at least one neighbour that belongs to r only, and the segment between itself and that neighbour does not intersect any of the CLUTs. The reader is referred to Section 5.5.2.1, for a description of the computation of edge nodes and boundary segments of the derived geometry. Figure 4.11 explains the `minus` operation on two **regions** values R and R' and the resultant derived geometry. Geometry R is of type SEG comprising r and R' is of type MEG comprising r'_1,r'_2 and r'_3.

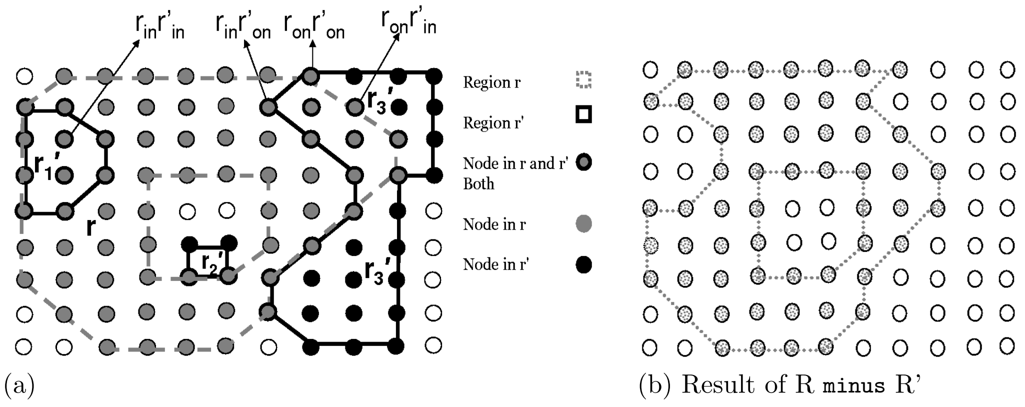


Figure 4.11: Derived geometry obtained as a result of a `minus` operation between **regions** values.

The operation `intersection` between two **regions** values creates a new derived geometry of type **regions**. It implies the need to search for a common `intersection` area that belongs to both operands. The new derived geometry contains the points belonging to the interior of r and the boundary of r'; those belonging to the boundary of r and to the interior of r'; those belonging to the interior of both r and r'; the common boundary points having among closest neighbours, at least one neighbour that either belongs to the interior of both r and r', or to the interior of r and to the boundary of r', or to the boundary of r and to the interior of r'; and common boundary points that constitute a CLUT. Figure 4.12 explains the `intersection` operation between two **regions** values R and R' and the resultant derived geometry, where geometry R is of type SEG comprising r and R' is of type MEG comprising r'_1, r'_2 and r'_3. The reader is referred to Section 5.5.2.1, for a description of the computation of edge nodes and boundary segments of the derived geometry.

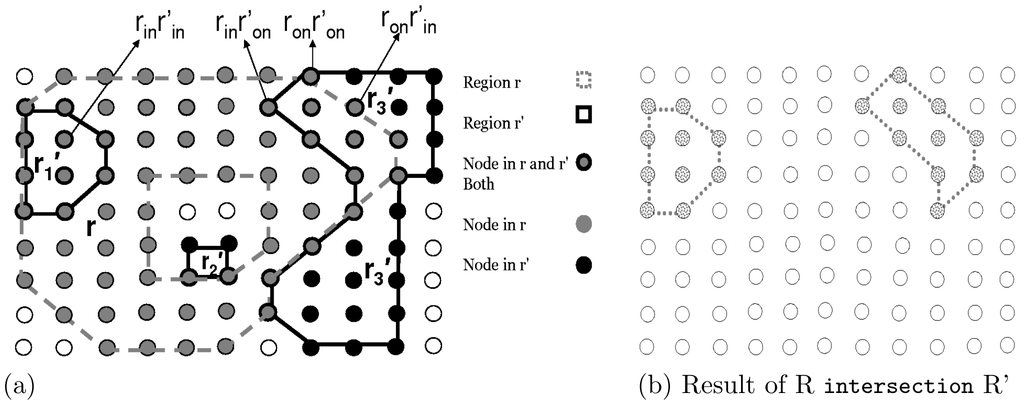


Figure 4.12: Derived geometry obtained as a result of a `intersection` operation between **regions** values.

The operation `common_border` creates a new derived geometry of type **lines** containing the common boundary segments of r and r'. Operation `contour` returns a derived geometry of type **lines** formed from the boundary segments of the outer boundary of the **regions** value r, i.e. inner boundaries are omitted. Recall, from Section 4.1, that a **regions** value with a hole has a disconnected boundary. The `vertices` operator returns the boundary points of a **regions** value r and produces a new derived geometry of type **points**.

The algebra presented this chapter allows for inferences about the set of spatial relationships that hold between the geometries and the derivation of geometries based on the existing geometries. The spatial inferences are formalized within the spatial framework described in Chapter 3 and the remainder of the dissertation is about the challenge of implementing these operations over WSNs. The inherent scarcity of resources and nature of underlying platform where execution is distributed and carried out periodically over sensed data streams give rise to non-trivial challenges. The challenge is very different in all respects from the ROSE algebra implementation developed [GS93, Sch97].

4.3 Summary

This chapter has shown how an algebra can be defined over the spatial framework presented in Chapter 3 whose operations enable the expression of sophisticated spatial analysis over WSNs. The chapter has provided rigorous, formal definitions of the spatial data types **points**, **lines** and **regions** together with spatial-valued and Boolean-valued operations over them. These definitions of spatial data types clarify the structure of spatial data type values from an abstract point of view. The spatial values can be of type SEG or MEG and the **regions** values may contain holes. This chapter has also presented the definitions of operations over valid combination of values of type **points**, **lines** and **regions**. These definitions serve as a specification for our distributed implementation discussed in Chapters 5 and 5.6.

Chapter 5

Algorithmic Strategy for In-Network Distributed Spatial Analysis Over WSNs

In line with the vision of the spatial analysis framework and distributed spatial algebra presented in the previous chapters, this chapter presents the in-network strategy for the evaluation of spatial analysis tasks as another contribution of this dissertation.

The algorithmic strategy for in-network distributed spatial analysis over WSN is specifically tailored for energy-efficient in-network execution, with a focus on issues related to minimizing unnecessary communication and the size of information to be communicated. The algorithmic strategy for the evaluation of complex algebraic expressions is divided into logically-cohesive components thereby facilitating component reuse and sharing. Each node is equipped with the task processing system thereby allowing it to participate in task dissemination, to contribute in the distributed evaluation of tasks, and to participate in the aggregation of intermediate results and in the routing of results to the user. In this chapter, the advantages of our approach over alternatives are argued in detail. The reader is referred to Section 5.6.2 for the description of task processing system.

This chapter describes how the problem of distributed computation of complex topological tasks, involving multi-element geometries as operands, can be mapped to the problem of first computing a node-level task state and then aggregating the node-level task state at two-levels using a new *bit-string-based* approach. The distributed computation of aggregates, such as count, sum, average, minimum, and maximum, is a well studied problem in WSNs. This dissertation introduces `BitwiseAND` as a new aggregate operator for aggregation of node-level task states. In addition, the algorithmic strategy for the evaluation of spatial-valued tasks is described in detail.

The structure of the chapter is as follows. Section 5.1 discusses the related work. Section 5.2 introduces the different types of spatial tasks to be evaluated over the framework described in Chapter 3. Section 5.3 describes the basic data structure which allows a node to associate itself to a particular geometry. The algorithmic approach for the evaluation of Boolean-valued tasks is described in Section 5.4, and that for spatial-valued tasks is described in Section 5.5. Section 5.6 presents distributed algorithms that constitute the first comprehensive, implemented, empirically evaluated proposal for expressive in-network spatial analysis in WSNs. Over this distributed computing platform, an application is executed as collection of processes. Section 5.7 summarizes the contributions of this chapter.

5.1 Related Work

To the best of our knowledge, the representational framework described in this dissertation and its associated spatial algebra and distributed algorithms constitute the first comprehensive, proposal for expressive in-network spatial analysis in WSNs. There exists work on detecting and reporting of topological changes. The topological changes reported include hole formation, hole disappearance, event region splitting, and event region merging. However, none of the existing work focuses on the computation of topological relationships among the three types of geometries or the derivation of new geometries from existing geometries using spatial-valued operations.

The work described in [JWN09] provides a computational model for WSNs to detect topological changes in dynamic regions based on local low-level snapshots of spatio-temporal data. The assumptions made by authors includes: the nodes are densely deployed and induce a voronoi diagram in the sensing area. Jiang et al. focus on using connectivity information, instead of location information, for the detection of topological changes to single element event geometry with no hole. Each node records the event state of its previous and current evaluation periods. The descriptions of the change is computed by the comparison of the event state at the consecutive two evaluation periods.

In [JWN09], at each evaluation period, the boundary of the event region is computed after the event detection. Once the boundary nodes are identified, the next step is the formation of groups based on the boundary nodes that have the same state in consecutive evaluation periods. Each group is assigned a unique ID, which is the node ID of the group leader. Each boundary node waits for a short random period of time for the *group construction* message from group leader. Upon the expiry of wait period, it declares itself a group leader and transmits requests to neighbouring nodes that have the same state to join the group. The *group construction* message from the group leader helps to construct the routing tree, which is then used for the aggregation of information. Each boundary node in a group communicates with its direct neighbours to find out the group ID of its adjacent groups in order to compute *neighbouring greater label set*, comprising the group IDs of the groups that are greater than the node's own group ID. After aggregation, the leader node is responsible for routing the group information to the sink, which includes the group state, the group ID, the *neighbouring greater label set* and group hop distance representing minimum hop distance in the group between nodes. After the reception of information from the group leaders, the sink node constructs the snapshot of the event region and the location of topological change.

Other work on using WSNs to detect topological change includes [FZWN08, JW08]. Farah and Zhong [FZWN08], propose an event-driven approach for capturing topological changes. The basic data structure used for the detection of topological changes is a *neighbourhood ring*. Each node maintains such a data structure for storing the event state of the nearest neighbours. In addition, it is assumed that each node is location-aware and records the event state of its previous and current evaluation periods. In this dissertation, it is also assumed that the nodes are location-aware and that each node keeps a record of its membership to induced, asserted and derived geometries in its own GIT.

Furthermore, for resource management, [FZWN08] divides a WSN into rectangular regions called clusters. The leader for each cluster maintains information about adjacent cluster heads and is also responsible for the activation of nodes in its associated cluster whenever its event state changes relative to the previous evaluation period.

For detecting the type of topological change, a non-boundary node checks for the consistency of information in its *neighbourhood ring* whenever its event state changes relative to the previous evaluation period. If, all the values in the neighbourhood ring are uniform, then an event node computes that the topological change is *hole loss*. Otherwise, a non-event node declares the topological change to be *hole formation*. If the values in *neighbourhood ring* are not uniform, then it makes use of the adjacent neighbouring components, their ID information, and its own event state to compute whether the topological change is *merging*. For *splitting*, it needs to compute the existence of cycle. Depending upon the requirement of detection of particular topological changes, a cluster head needs to communicate with one or all adjacent cluster heads. Detailed description about the aggregation of information and the reporting of topological change to the user is not provided in [FZWN08]. With a view to supporting this class of applications, this dissertation shows that such spatial analysis can indeed be efficiently implemented in WSNs.

Worboys et al. [WD06] propose a scheme for the detection of topological changes. The assumption is that the nodes and the communication links in a WSN divide the underlying plane into set of triangles (i.e., triangulations are superimposed on top of the underlying communication topology of a WSN). The authors use the triangulation method for the estimation of the spatial extents of event geometries. All triangles that are exterior to an event geometry constitute the boundary triangles, and all the triangles that are lying in the interior constitute the interior. The event state of a node in successive evaluation periods results in dynamic triangulation because of the changes, i.e., addition or deletion, made to the triangles in the embedded triangulation framework. The authors have proposed triangulation transition rules which are used for the detection of topological changes including splitting, merging, movement, stasis and hole formation and disappearance to single element event geometries. Whilst [WD06] is restricted to the detection of topological changes, the focus of this dissertation is on tracking, complex, transient, fast-evolving physical phenomena and enabling the analysis of the topological relationships they exhibit with other physical phenomena or with permanent geographical features.

Jiang and Worboys [JW08] propose in-network algorithms for the detection and reporting of topological changes. The work allows for reporting the *Minimum Bounding Rectangle* (MBR) of the area where change has taken place and the type of topological change. It is assumed that an event region and a hole inside it is a two dimensional object, i.e., not of type **points** or **lines**, and that each node is location-aware. This is related to the work described in this dissertation, as here it is also assumed that a node is location-aware and that holes inside **regions** value are two-dimensional objects instead of type **lines** or **points**.

For detecting of topological change, each node assigns itself a colour based on its event state information in successive two evaluation periods. For example, a node with event state (0/1) denotes a **false** event state in the previous evaluation period and a **true** one in the subsequent evaluation period. A node assigns itself the *black* colour on having the state (1/1), and *white* on having state of (0/0). Otherwise, it assigns itself the *gray* colour. The *black* nodes are called as *black* C-component nodes, having one-hop neighbour of *gray* colour or having a *black* colour neighbour part of *black* C-component nodes. *White* C-component nodes are defined in similar manner. Communication is required for the assignment of the *red* and *blue* colours. A *gray* colour node which is a k-hop neighbour of *black* or *white* is assigned a *blue* colour. The *red* colour node is selected among the *black* and *white* colour nodes that satisfies either one of the following: having one-hop neighbour of *gray* colour, or having one-hop neighbour part of *black* or *white* colour C-component.

After colour assignment, a leader election is held between *blue* colour nodes, resulting in the selection of a *representative* node and the construction of a routing tree. A leader is also elected among *red* colour nodes to act as cluster leader. After that, cluster formation occurs among *red* nodes, to which the leader node ID is assigned as label. Cluster formation results in the formation of a routing tree rooted at the cluster leader. Each cluster head is connected to one of the neighbouring *blue* node. After that, the *red* nodes engage in aggregation of information. The aggregation phase results in the computation of the MBR of the cluster and the collection of information about adjacent neighbouring clusters. Each cluster head is then responsible for transmitting this information towards the *representative* leader for analysis. After the reception of information from cluster heads, the *representative* node is then responsible for detecting topological change and aggregating MBR information, which is then transmitted towards the root node. This aggregation and reporting scheme is related to the work described in this dissertation in which, aggregation of membership states for computing topological relationships takes place at two levels. Therefore, leaders are selected at two-levels: first at the SEG level and then at the top-level. However, whilst the focus in [JW08] is on computing the minimum bounding rectangle of the area where the topological change is detected, whereas in the work reported here the information that is aggregated consists of bit-level encodings of the partial results in evaluating a complex distributed task to compute the final result.

Bi et al. [BGTD06] propose an algorithmic strategy for the detection of topological holes created because of environmental factors, of random deployment, or the fact that the nodes have run out of energy. The algorithm works under the assumption that the nodes are uniformly deployed. The algorithm exploits two facts. Firstly, that the hole boundary nodes have a smaller number of neighbours than non-boundary nodes, and secondly, that most of the *one-hop* neighbours of boundary nodes are also neighbours of the *two-hop* neighbours of boundary nodes. Each node maintains a list of *one-hop* and *two-hop* neighbours, and list of neighbours received from its neighbours lying at most *two-hop* away. After the computation of this information, each node computes the count of neighbours for each of its *two-hop* neighbour. Based on this information, each node computes the boundary state for its *two-hop* neighbourhood, and sends its computed boundary state to those neighbours. Upon the reception of such information, a node computes whether it is a boundary node based on the number of **true** and **false** boundary states and the weight assigned to each neighbour boundary vote (based on its degree, i.e., the count of its two-hop neighbours).

Bi et al. [BTG$^+$06] provide an improvement to the algorithm proposed in [BGTD06] in order to achieve energy efficiency. The idea is that instead of a node computing the boundary state of each *two-hop* neighbour, it transmit its computed state towards those neighbours. A node needs to compute the average of the degree of its *two-hop* neighbours. If its own count of *two-hop* neighbours is less than the average degree of *two-hop* neighbours, it needs to compute whether it itself is a boundary node and transmit its boundary state to its *one-hop* neighbours with a request to confirm its boundary state value. Upon the reception of a response from its *one-hop* neighbours, a node decides to retain its boundary state value or to flip it. In contrast to [BTG$^+$06, BGTD06], the work presented in this dissertation does not detect topological holes, instead it detects the topological relationships between geometries, with or without holes.

Zhu and Sarkar [ZSGM08] provide a detection algorithm to track the contours of an evolving region, while guaranteeing that the contour represents information about the topological features. To detect a contour, each node needs event state information from its neighbours at each evaluation period. On the basis of its own and its neighbours event state information, a node assigns itself a colour. It assigns itself *black* if its own and all of its neighbours event states are **true**. If all of its neighbours and its own event state are **false**, it assigns itself *white*. Otherwise, it assigns itself *gray*. The contour is computed among the *gray* nodes that are located close to the black nodes based on some threshold value. In this dissertation, the focus is not on the in-network aggregation of information or on designing a scheme that provides information about topological features, instead the focus is on the detection of topological relationships between asserted, induced and derived geometries.

Jiang and Worboys [JW09] present a tree model for modelling the topological structure of an event geometry at each evaluation period. The idea is that as the areal object undergoes topological change, the associated tree also changes. The work presented proposes the idea that tree structures allows to represent topological relationships between regions and some of the topological relations such as *surrounded by* can be represented by using single tree. An event geometry is defined as a collection of region components, possibly with holes. This is to some extent related to the work described in this dissertation, as an induced geometry is a finite set of pairwise disjoint, polygons with or without holes.

Deb et al. [DBN03] propose a distributed parameterized algorithm for sensor topology retrieval at multiple resolutions. The resolution of the topology is defined in terms of number of edges and number of nodes. The parameters that control the resolution include the *virtual range* (which defines the maximum distance r between nodes that form an edge), the *resolution factor* (which defines the percentage of edges originating from a node, which a node needs to return to its parent as part of computing reply to the topology discovery reply), and the *query type* (which defines the type of query so that the specific filters and aggregation functions can be mapped to it). The algorithm uses a colouring scheme. Initially all nodes are *white* except for the node that initiates the topology extraction request, which is *black*. The initiating node becomes the root of the black node tree. Each *black* node sets a *black* node as a parent which is at most two-hops away and from which it lastly received a topology extraction request message.

Upon reception of the broadcast message, each node broadcasts a request to its neighbours lying in a distance r. A node that receives a request from a *black* node changes its colour to *red*. A node that receives a request from a *blue* or a *red* node, or that is in the communication range of a *black* node, sets its colour to *blue*. Each *blue* node sets a timer (expiry time of which is proportional to its deviation from $2r$ distance from the *black* colour neighbour), upon the expiry of which it changes its colour to *black*. The time of the timer is set such that *blue* nodes that are closer to $2r$ distance change their colour earlier. Upon changing colour from *blue* to *black*, a node sets a acknowledgement timer to reply to the discovery request. Until the expiry of the acknowledgement timer, it receives information from child nodes and aggregates all topology information from its children and adds f fraction of edges from its own region. Upon the expiry of the acknowledgement timer, a *black* node transmits its response to its parent. Whilst [DBN03] is restricted to the retrieval of topology at multiple resolutions, the focus of this dissertation is on performing in-network distributed spatial analysis over WSNs.

In summary, in contrast, to the related work presented in this section, our work provides a comprehensive approach to the detection of topological relationships and derivation of new geometries from existing geometries using distributed algorithms for in-network processing techniques. The related work provides comprehensive techniques for the following: detecting topological changes, detecting and reporting topological changes, detecting topological holes, extracting topological features, and topology information retrieval. We will now describe how tasks are conceptualized as algebraic expressions and then move on to describe the data structures and algorithmic strategies used in their evaluation.

5.2 Tasks as Complex Algebraic Expressions

Simple tasks consists of a single spatial operator. Complex tasks consists of more than one operator. In the case of complex Boolean-valued tasks, the logical operators **and**, **or** and **not** (i.e., the classical Boolean connectives) are used to construct complex algebraic expressions.

5.3 Concrete Data Structures

In our approach we assume that each node holds a data structure known as a *Geometric Information Table* (GIT). This section describes the attributes of GIT. In a node S, an entry in its GIT is a quadruple $\langle I, T, B, \theta \rangle$, where I denotes the geometry ID of the geometry to which N belongs, T denotes the spatial data type of I, B is **true** iff N is in the boundary of the geometry identified by I, and θ denotes the validity period, i.e., the time-to-live (TTL) after which the GIT entry becomes invalid. In the case of induced geometries, when an event of interest is detected at a given time in a node, its GIT is updated. If the entry already exists for that induced geometry, its TTL is reset. Otherwise, a new entry is added to the GIT. Upon the expiry of the TTL, the entry is removed. In the case of derived geometries, an entry is added to GIT with a TTL when the geometry is derived. The TTL is set to the minimum TTL of the original operands. The reason is that the new geometry is derived on the basis of existing geometries, where one or both operands used for its derivation can be of type induced. The TTL for asserted geometries is set to ∞ and is not updated. Thus, the GIT allows a node to keep a record of which geometries it is part of. This is used, e.g., to decide whether the node should evaluate some task at a specific evaluation period.

5.4 Evaluation Components for Boolean-Valued Tasks

The procedure for the evaluation of Boolean-valued tasks consists of three sub-tasks, *task dissemination* to the nodes, *distributed task evaluation*, of complex spatial-algebraic expressions, and *result processing*, which sends the results to the gateway.

5.4.1 Task Dissemination

In this phase, the task message is disseminated to the relevant nodes in the WSN. The task dissemination phase accomplishes the additional purpose of constructing the routing tree for result processing rooted on (as well as electing) the *first-level* leader to whom the results from leader nodes at the SEG levels are sent for aggregation, as described later. A description of the task dissemination phase was provided in Section 3.3. The reader is referred to Section B.1.1 for more details.

5.4.2 Distributed Task Evaluation

Every task MBR node evaluates the interpretable structure conveyed by the task message, i.e., every node runs an interpreter for the event detection and algebraic evaluation steps required in our approach to spatial analysis. After receiving the task, a node may need to wait until the estimated (on the basis its location in the task MBR) time needed for the task message to be received by the furthest node in the MBR. The reader is referred to Section B.1.1 for more details. Once the task message is received, the MBR nodes have to execute the task specified in the task message in a distributed manner. Apart from other attributes, the task message contains the internal, interpretable form of the task specification represented in postfix notation.

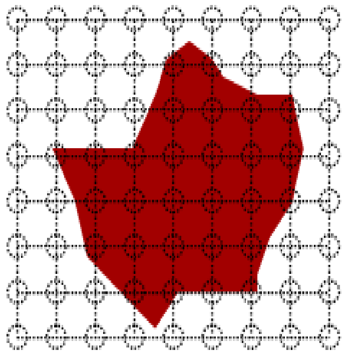
This section describes how a task containing Boolean-valued operators can be evaluated in a distributed manner. Note that this section only describes the case where the operands are of the type **regions**. Furthermore, in all the figures presented in this section, which provide illustrative example scenarios, it is assumed (unless specified otherwise) that the nodes are deployed in a uniform grid, and that a node has a maximum neighbourhood of 8 nodes. For the description of the algorithmic strategy for the evaluation of Boolean-valued tasks, it is assumed that r and r' denote SEG **regions** values, and R and R' denote MEG **regions** values. Furthermore, r (or R) represent the left-hand-side (LHS) operand and r' (or R') the right-hand-side operand (RHS) of a binary Boolean-valued spatial operation represented in infix notation. Equivalently, in postfix notation r (or R) represents the first operand and r' (or R') the second operand of a binary Boolean-valued spatial operation. In this dissertation, spatial values are also referred to as operands or geometries.

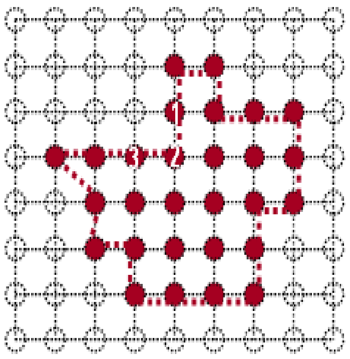
As described in Chapter 3 and Chapter 4, our sensor space framework represents a WSN as a graph where sensor nodes denote vertices and the communication link between nodes represent edges. An induced geometry represents a continuously-evolving phenomenon and the pre-requisites for the characterization of induced geometries are event detection and boundary computation. The boundary detection phase allows nodes that are part of an induced geometry to determine whether they belong to the interior or the boundary of the induced geometry. Note that, due to reasons including node sensing, communication range, network density and size and shape of event geometry, it is not possible to characterize accurately the induced geometry, as shown in Figure 5.1, in which (a) depicts the actual event geometry, exemplifies what the characterization of induced geometry might be, and represents the spatial extent of the induced geometry to be considered in the scope of this dissertation. The boundary separates the sensor nodes that satisfy the event predicate and those that do not. Figure 5.1(c) depicts the approximate spatial extent of the event geometry that the current network density supports to be detected. The spatial extent of the induced geometry includes the edges that is formed by the two closest boundary nodes, therefore, it includes the spatial area that is enclosed by considering those edges. Figure 5.1(d) depicts the edges that are considered as part of induced geometry spatial extent (e.g., Nodes 1 and 3 are closest boundary nodes and there exist edge between them).

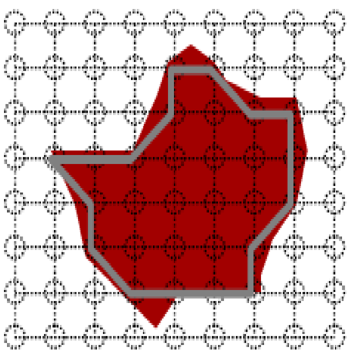
Let s_i belongs to boundary and $\mathrm{N}(s_i)$ represent the set of *closest* neighbours of s_i and $\mathrm{N}_{on}(s_i)$ represent the set of boundary neighbours of s_i where $\mathrm{N}_{on}(s_i) \subseteq \mathrm{N}(s_i)$. Node s_i confirms that the segment it creates with $s_j \in \mathrm{N}_{on}(s_i)$ is a boundary segment only if it finds that boundary segment does not intersects the interior of the **regions** value.

In this dissertation, it is assumed that information about asserted geometries is manually configured by the deployer since these geometries represent static real-world physical entities not continuously-evolving entities like induced geometries. In most real world scenarios, asserted geometries of type **regions** are `area_disjoint`. We assume, therefore, that our framework all asserted geometries of type **regions** are `area_disjoint`. For example, one real world scenario is shown in Figure 5.2. Figure 5.2 (a) shows part of a university campus and Figure 5.2 (b) shows how part of the map on the left can be represented over a sensor space. In (b), nodes s_{19} and s_{24} are closest boundary neighbours and a boundary segment exists between them. The existence of such a boundary segment makes the **regions** values $R48$ and $R49$ not `area_disjoint`.

To avoid such scenarios, each node is equipped with an *edge information table* (EIT) in which it maintains information about edges that do not exist between itself and its closest boundary neighbour. An entry for a node S in the EIT is a pair $\langle GID, N_{ID}\rangle$ where *GID* denotes the geometry ID and N_{ID} denotes the neighbour ID. Such a table is also required to represent information about derived geometries as described in Section 5.5. Therefore, whenever information about a geometry is removed from the GIT of a node, its associated entries in the EIT are also removed. The snapshot of GIT and EIT of nodes s_{19} and s_{24} is given in Figure 5.3.


(a)Actual Event Geometry


(b) Induced Geometry


(c) Spatial extent of detected event geometry

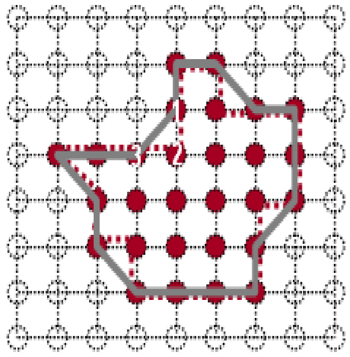
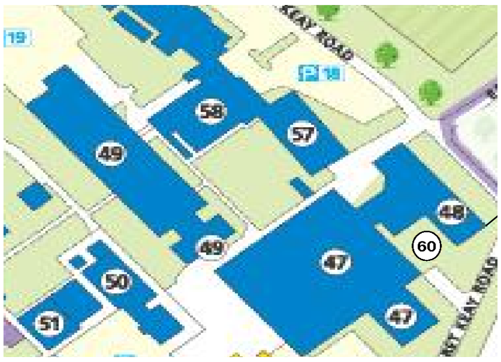

(d) Spatial extent of induced geometry

Figure 5.1: Characterization of induced geometry and its spatial extent.



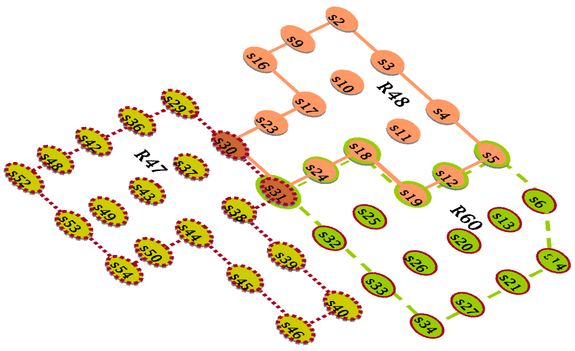
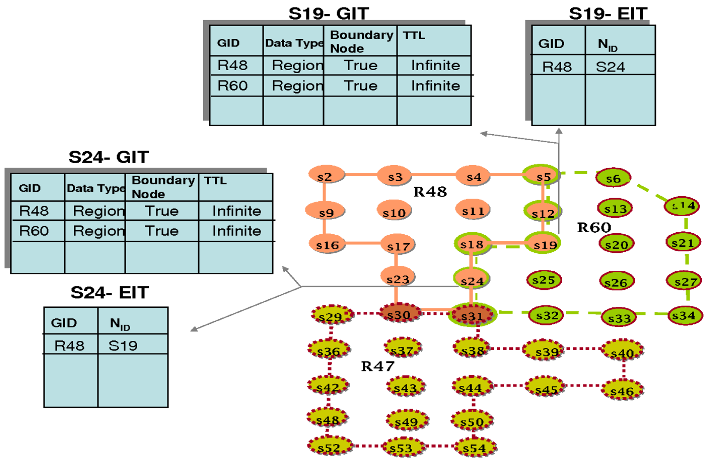


Figure 5.2: (a) Snapshot of a university campus (b) Geometries over the sensor space

In this dissertation, two boundary nodes s_i and s_j belonging to a **regions** value r form a boundary segment iff they satisfy the following conditions: (1) s_i and s_j are closest boundary neighbours, (2) the segment created by $\widetilde{s_i s_j}$ does not pass through the interior of r, and (3) s_i and s_j do not have an entry for each other with respect to r in their respective EITs.



S19- GIT

GID	Data Type	Boundary Node	TTL
R48	Region	True	Infinite
R60	Region	True	Infinite

S19- EIT

GID	N_{ID}
R48	S24

S24- GIT

GID	Data Type	Boundary Node	TTL
R48	Region	True	Infinite
R60	Region	True	Infinite

S24- EIT

GID	N_{ID}
R48	S19

Figure 5.3: Snapshot of GIT and EIT of nodes s_{19} and s_{24}

5.4.2.1 Evaluation of Simple Boolean-Valued Tasks on Single Element Geometries

As discussed in Section 3.2.2, in a WSN, information about geometries is distributed rather than centrally held. Each node member of a geometry is only aware of part of that geometry, the scope of which is restricted by its radio and sensing range.

Recall from Section 5.2, that a simple Boolean-valued task consists of a single Boolean-valued operator. The use of a universal quantifier $\forall$, in the formal definitions of the topological predicates for operands of type SEG **regions** in Table 4.12, implies that, for the distributed evaluation of topological predicates, one needs to consider all the nodes that belong to an operand. For example, from the formal definition of the `vertex_disjoint` operator in Table 4.12, for the relationship to hold, all nodes that are members of operand r must not be members of operand r'.

One strategy for evaluating a topological operation in WSNs might be to collect information at a central destination, external to the network, from all nodes that belong to one or both operands of the corresponding operation. The following pieces of information would be needed: the node ID, the indication as to whether that node belongs to either or both of the operands, the indication as to whether it belongs to the interior or the boundary, of the corresponding geometry. The central node, upon the reception of this information, can evaluate the task. This strategy is not energy-efficient and may lead to network congestion (as the geometry size increases) because nodes need to transmit all the information towards the sink. To achieve energy efficiency, it is better to perform data reduction and filtering as early as possible in a data path, and to reduce the number of relay nodes. This characterizes the first challenge in task processing in WSNs, viz., to preserve energy stocks through in-network processing.

A more efficient in-network approach is for each node to produce, through localized processing, a local outcome for each operation in the task, specifying whether it satisfies the prerequisites for the success of the operator being evaluated. These local outcomes from nodes inside network can then be combined using an aggregation scheme and only the fine-grained result is routed to the gateway. For example, for the computation of a local outcome for a `vertex_disjoint` operation, a node needs to know which operands it is part of, an indication as to whether it lies at the boundary or the interior of any of the operands, and the data type of the operands. It can be seen that, for the computation of `vertex_disjoint` relationship in our framework and under the assumptions we have made, the required information is available in the node's GIT and therefore it is feasible, to compute the local outcome (referred to, hence forth, as the membership state of the node) using local computation alone.

This strategy for membership state evaluation resembles a *voting scheme*, where the vote cast is the node's membership state. Therefore, in the case of a `vertex_disjoint` operator, an MBR node casts one of three types of votes: **true**, **false** or **OP_not_applicable**. A **false** vote denotes that the node does not fulfill the prerequisites for `vertex_disjoint`, a **true** vote denotes that it fulfils those prerequisites, and an **OP_not_applicable** vote denotes either that the node does not belong to either of the operands or that the node does not satisfy the spatial data type requirement for the evaluation of the operator, e.g., , for the `adjacent` relationship, it is required that both operands are of type **regions**. In the case of `vertex_disjoint`, a node that is a part of the task MBR and a member of exactly one of operands declares its membership state to be **true**, otherwise, it declares it to be **false**.

For the distributed evaluation of topological predicates, one needs to consider the membership state from the nodes that belong to one or both operands. For considering the membership states from all the nodes there is a need to have an aggregation scheme. Computation of aggregates, such as count, sum, average, minimum, and maximum, is a well studied problem and can be solved in a distributed manner. In the case of a simple Boolean-valued task on a SEG, after membership state computation, the problem of detecting a topological relationship reduces to the problem of computing a *count* aggregate, which can be formally described as:

Given a network of n nodes, where each node i holds a value v_i, determine the count of these values in a distributed manner.

This is a well studied problem in WSNs. The algorithms available for this problem can be broadly divided into two categories: tree-based and gossip-based. The reader is referred to Section 2.3 for the description of related work on aggregation approaches. Section 5.4.2.6 provides more discussion. Let us suppose for the time being that tree-based aggregation is used for computing the aggregation function. Each child node needs to transmit its state towards parent nodes. Each parent node on its way to the leader nodes wait to receive state information from the child nodes. After receiving information from child nodes, each parent node computes the partial aggregated result by including their own state and transmitting the result towards their parent on their way to the leader node. Therefore, in the case of the `vertex_disjoint` operator, if after applying aggregation function (i.e., count) over both its own computed and the received partial aggregated results from child nodes, the leader node obtains a count of zero for **false** membership states and a count of greater than zero for **true** states, it declares that r and r' are `vertex_disjoint`.

Consider the example scenario in Figure 5.4, where geometries r and r' are `vertex_disjoint`. Figure 5.4 shows that an efficient and cost-effective solution for selecting a leader node for the aggregation tree is to select a node that is part of at least one of the operands. The strategy used

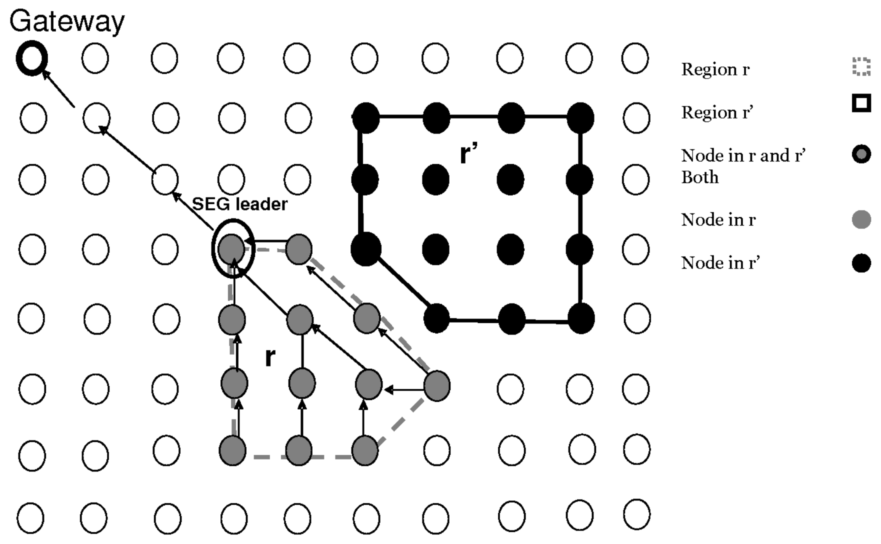


Figure 5.4: `vertex_disjoint` relationship between two geometries of type **regions**.

in the dissertation for the selection of a leader node is discussed in Section 5.4.2.6. Furthermore, it can be seen from Figure 5.4 that for computing the outcome of the `vertex_disjoint` operation, the membership states from just one of the operands suffice. It will be seen in the next examples that the same holds for every topological operator except `equals` and `not_equals`, in which case the membership states from both r and r' must be aggregated.

For some operators (e.g., `vertex_disjoint`), a node only needs to perform a look-up in its own GIT to produce a membership state. Other operators (e.g., `adjacent`) require, in addition to the local GIT look-up, the gathering of GIT information from their one-hop neighbours in order to produce a local outcome. Therefore, from this point onwards, this section first discusses Boolean-valued operations that only require a local GIT lookup and then moves on to consider operations requiring both a local GIT lookup and the collection of information from neighbours.

Membership State Computation Based On Local GIT Information. In addition to `vertex_disjoint`, computing a membership state for `area_inside`, `vertex_inside`, `equals`, and `not_equals` only requires a local GIT look-up. An MBR node declares its operation state to be **OP_not_applicable** if the node does not belong to either of the operands of an operator or if the node cannot satisfy the spatial data type requirement for the evaluation of the operator. Possible membership states for these operators are given in Table. 5.1.

`area_inside`	:	**true**, **false**, **unknown** or **OP_not_applicable**
`vertex_inside`	:	**true**, **false**, **unknown** or **OP_not_applicable**
`equals`	:	**false**, **true**, or **OP_not_applicable**
`not_equals`	:	**false**, **true**, or **OP_not_applicable**

Table 5.1: Possible membership states of operators that only require local GIT look-up to compute their membership state

In the case of `area_inside` and `vertex_inside`, an MBR node declares its state to be **unknown** if it belongs to r' only, reflecting that it is part of the container operand only. An MBR node declare its state to be **false** if it does not fulfill the prerequisite for the corresponding operator (as described in Table 4.12), and **true** state if it fulfils those prerequisites. In case of `equals` and `not_equals`, an MBR node is responsible for checking entries for neighbours in its EIT related to operands r and r'. It sets its computed state to **false** upon finding that it has an entry in EIT for one of the operands but not for both.

Moreover, in the case of `vertex_inside`, `area_inside` and `vertex_disjoint`, aggregation of states from one of the operands is enough to compute the final outcome. Detailed discussion as to which operand is selected is given in Section 5.4.2.5. Considering the same tree-based aggregation approach discussed above, the leader on the computation of final aggregation, over its own and the received partial aggregated results from child nodes, declares the result to be **true** if it computes a count of greater than zero for **true** and count of zero for **false** state.

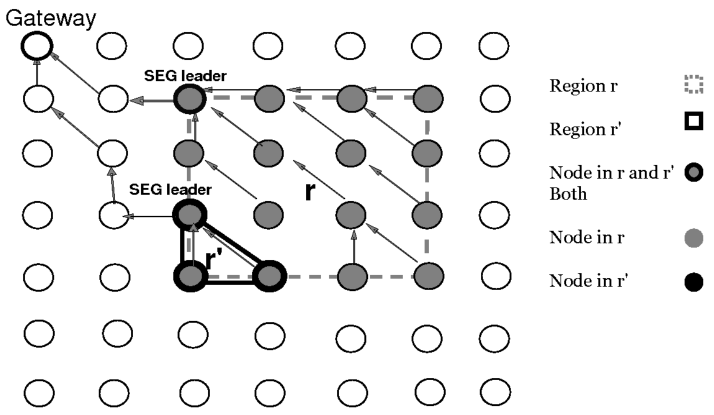


Figure 5.5: `equals` relationship cannot be determined in some scenarios if aggregation is performed on the basis of a single operand.

An example scenario is shown in Figure 5.5. If the states are collected from operand r', then the wrong outcome would result. The reason is r' has no interior and all nodes and edges of r' belongs to r, but some nodes and edges of r does not belong to r'. To avoid such scenarios, therefore, in the case of both `equals` and `not_equals`, the membership states from both operands must be considered in order to compute the final outcome for `equals` and `not_equals` operations accurately.

Membership State Computation Based On Local and Neighbour GIT Information. Some operators (e.g., `adjacent`, `area_disjoint`, `edge_disjoint`, `edge_inside`, `meets`, `intersects`, `border_in_common`) require, in addition to a local GIT look-up, that common boundary nodes (CBN) i.e., those that belong to the boundary of both operands, obtain GIT information from their one-hop neighbours to compute their membership state. All non-CBNs can compute their state using the information available in their own GIT. For example, in the case of `edge_disjoint`, `edge_inside`, `border_in_common`, a CBN needs to test for the existence of a common boundary segment (CBS); in the case of `intersects`, `area_disjoint`, it needs to test for the existence of a CLUT, and in the case of `meets`, `adjacent`, it needs to test for the existence of a CBS and a CLUT. A CBS exists between two CBNs that are *closest* one-hop boundary neighbours and form a valid *boundary segment* of both r and r'. Recall from Section 4.2, that a LUT $\langle s_1, s_2, s_3 \rangle$ satisfies the properties that its interior and its edges do not contain any node that is a neighbour of s_1, s_2 or s_3 and that all its edges have length less than or equal to unit length (i.e., the prevailing radio range).

BorderInCommon. Suppose that s_i and s_j are two closest CBNs of r and r' and do not have entry in their EIT for each other related to r or r' or both. To compute whether they form a valid *boundary segment*, information is collected from their neighbours in order to check whether the segment created by them does not overlap with the interior of either r or r'. Let the space around every CBN be divided into 12 sectors defined by an angle of 30° each. As each node knows only about the geometries it is part of, a CBN, therefore, requests its neighbours that are part of r or r' (or both) for their location, and their geometry membership information. The neighbours, then send the required information. With this information, the CBN computes the distance to each neighbour, the sector where it lies, and its closest neighbours in each sector for whom it has no entry in its EIT.

This information allows the CBN to construct the LUT graph for each operand of the subregion around it by taking itself as the center [PR00]. Firstly, for the r operand (i.e., considering the LUT graph of the subregion constructed by nodes belongs to r), the CBN computes whether the boundary segment forms with its neighbour CBN is part of one of the LUTs and no other LUT is adjacent to it. If it satisfies the condition for r, it will check it for r'. If it finds the segment is shared by two adjacent LUTs, for any of the operands, the CBN will repeat this process for other neighbour CBNs. Upon finding that segment is not shared by two adjacent LUTs for any of the operands, it declares its state to be **true**. After repeating the procedure for each neighbouring CBN, if a CBN does not form a valid CBS, it declares its state to be **unknown**. The **unknown** state denotes that the node belongs to either r or r' or both but not to a CBS. The description of the computation of LUT is given later in this section under the description of `area_disjoint`.

Consider the example scenario in Figure 5.6. Geometries r and r' have two boundary nodes (36 and 47), that are CBNs. Both nodes request information from their neighbours. Upon receiving request from node 36, the neighbouring nodes send a reply. Similarly, for a request from node 47, the neighbouring nodes send a reply. Upon reception of these replies, nodes 36 and 47 compute minimum distance neighbours in all sectors. Lets explain the working of algorithm on CBN 36 (same is true for CBN 47). CBN 36, first computes whether the segment forms with its neighbour CBN 47 is part of one of the LUTs in operand r and no other LUT is adjacent to it. On computation, it finds out that the segment created by itself with neighbouring CBN 47 overlaps the interior of r (as the segment $\overline{3647}$ is shared by two adjacent LUTs <36, 37, 47> and <36, 46, 47>). Therefore, both 36 and 47 declare their state to be **unknown**. The arrows in Figure 5.6 denote routing of information, which is aggregated on its way towards the parents at various levels of the tree.

In Figure 5.7, geometries r and r' have two CBNs (viz. 36 and 47). The segment created

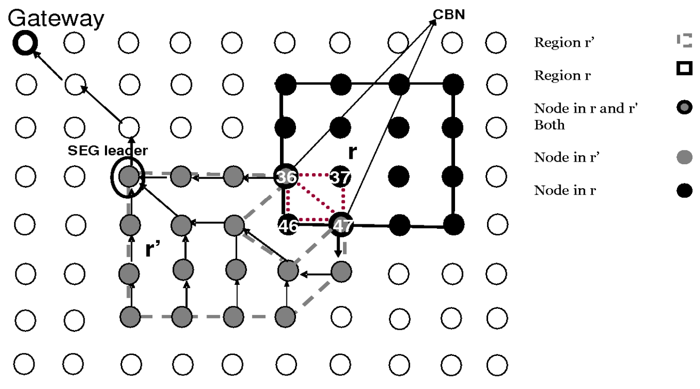


Figure 5.6: Geometries r and r' not having `border_in_common` and `adjacent` relationship

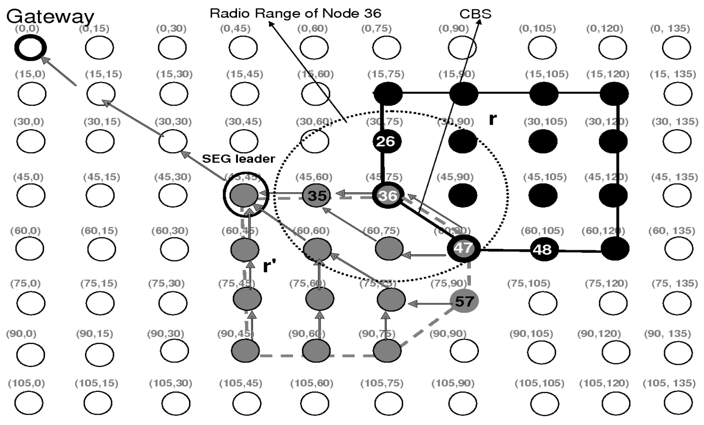


Figure 5.7: Geometries r and r' having `border_in_common` and `adjacent` relationship

by CBN 36 and neighbouring CBN 47 is a valid boundary segment ($\widetilde{3647}$) not overlapping the interior of either operand. Therefore, both 36 and 47 declare their membership state to be **segment_node**. In the case of `border_in_common`, upon reception of partial aggregated results from child nodes, the leader performs the final aggregation, over its own and the received partial results, and declares final outcome of the `border_in_common` operation to be **true** if after final aggregation it get a count of greater than zero for the membership state **segment_node**.

AreaDisjoint. The formal definition of `area_disjoint` in Table 4.12 states that two **regions** values are `area_disjoint` if they do not have common shared areas.

For two **regions** values that have an interior, the definition of `area_disjoint` relationship given in [Sch97] (viz., that no **point** lying in the interior of one of the geometries intersects the interior or the boundary of the other geometry) suffices. In this case, evaluation only requires a local GIT look-up. However, if one or both **regions** values are *unit* **regions**, or if the two geometries have a shared *unit* **regions**, the definition in [Sch97] is not sufficient as explained in Section 4.2.1.6. This has led to the extended definition in Table 4.12.

In such cases, the CBNs perform localized communication with neighbouring nodes to determine whether they are part of the shared *unit* **regions** value. All other nodes compute their state based on the information available in their GIT. For `area_disjoint`, the possible membership states are **true**, **false**, and **OP_not_applicable**. All nodes that do not belong to a common boundary can compute their state using their own GIT. If a non-CBN belongs to exactly one of the operands, then its state is **true**, otherwise it is **false**.

Each CBN transmits an information request to its one-hop neighbours. All one-hop neighbours that belong to either operand respond with their location and an indication as to whether they belong to the interior or the boundary of either or both of the operands. Upon receipt of this information, a CBN declares its state as **false** if one or more of its one-hop neighbours belong either to the interior of both operands or to the boundary of one operand and the interior of the other. If a CBN finds no neighbouring non-CBN belonging to both geometries, then it declares its state to be **true** if it has less than two neighbouring CBNs, otherwise (i.e., if it has two or more neighbouring CBNs) it needs to find whether it is part of a common *Unit* **region** area. In this case, a CBN computes whether it forms a CLUT with its neighbouring CBNs.

A CBN can compute whether it is part of a CLUT using a barycentric technique [CM69]. The method is explained briefly below. The order of the three CBNs in the computation of a CLUT is important. For each CLUT in case a CBN participates in more than one, each CBN places itself and its neighbouring CBNs in an order based on their node ID. If a CBN participates in one or more CLUTs, it sets its state to **false**.

Now suppose that s_1, s_2, and s_3 denotes CBNs and the CBNs order themselves based on their IDs as s_1, s_2, and s_3. Let $N(s_i)$, where $i = 1, 2, 3$, denote the set of neighbours of each s_i. Each s_i then computes the *barycentric coordinates* for each of their neighbours in $N(s_i)$ as shown below, with $s_j \in N(s_1)$, and with $v0$, $v1$ and $v2$ representing vectors:

v0$_{xLoc}$ = S_{2xLoc} - S_{1xLoc}
v0$_{yLoc}$ = S_{2yLoc} - S_{1yLoc}
v1$_{xLoc}$ = S_{3xLoc} - S_{1xLoc}
v1$_{yLoc}$ = S_{3yLoc} - S_{1yLoc}

DotProdv0v0 = ((v0$_{xLoc}$ * v0$_{xLoc}$) + (v0$_{yLoc}$ * v0$_{yLoc}$))
DotProdv0v1 = ((v0$_{xLoc}$ * v1$_{xLoc}$) + (v0$_{yLoc}$ * v1$_{yLoc}$))
DotProdv1v1 = ((v1$_{xLoc}$ * v1$_{xLoc}$) + (v1$_{yLoc}$ * v1$_{yLoc}$))

invCrossDotProd = (1.0 / ((DotProdv0v0 * DotProdv1v1) - (DotProdv0v1 * DotProdv0v1)))

v2$_{xLoc}$ = s_{jxLoc} - S_{1xLoc}
v2$_{yLoc}$ = s_{jyLoc} - S_{1yLoc}

DotProdv0v2 = ((v0$_{xLoc}$ * v2$_{xLoc}$) + (v0$_{yLoc}$ * v2$_{yLoc}$))
DotProdv1v2 = ((v1$_{xLoc}$ * v2$_{xLoc}$) + (v1$_{yLoc}$ * v2$_{yLoc}$))

// Compute Barycentric coordinates b_m and b_n
b_m = ((DotProdv1v1 * DotProdv0v2) - (DotProdv0v1 * DotProdv1v2)) * invCrossDotProd
b_n = ((DotProdv0v0 * DotProdv1v2) - (DotProdv0v1 * DotProdv0v2)) * invCrossDotProd

Figure 5.8 describes the values of *Barycentric coordinates* inside a CLUT and in the direction of each edge in the CLUT. If a neighbour node $s_j \in N(s_1)$ lies inside a CLUT or on its edges, then the barycentric coordinates for that neighbour are as $(b_m > 0)$ and $(b_n > 0)$ and $(b_m + b_n <= 1)$.

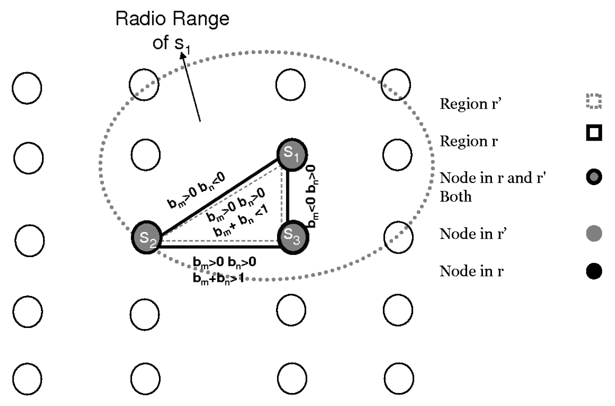


Figure 5.8: *Common Localized Unit Triangle* (CLUT) Computation

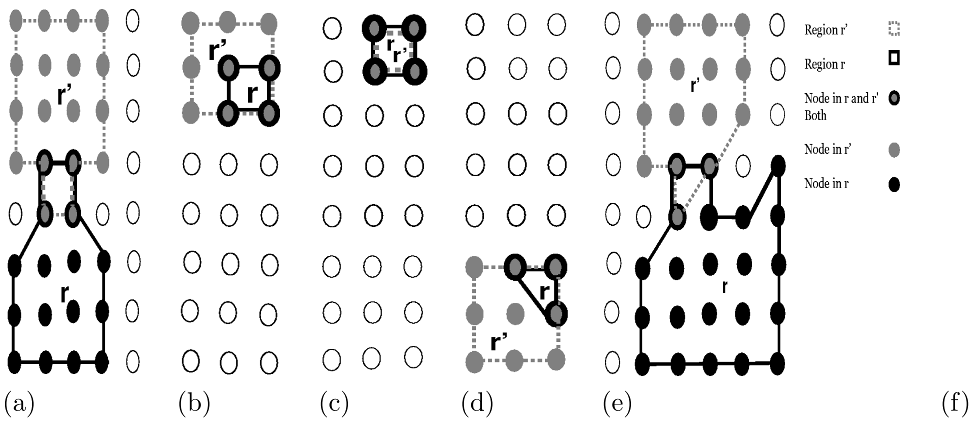


Figure 5.9: Examples of existence of `intersects` relationship between values r and r' of type SEG **regions**

Suppose there exist only two CBNs as neighbours of node s_1. Node s_1 declares its state to be **true** if $(b_m > 0)$, $(b_n > 0)$, $(b_m + b_n <= 1)$ and no segment of the CLUT is in the EIT. The state **true** denotes that s_1, s_2 and s_3 forms a CLUT. A CBN declares its state for `area_disjoint` to be **false**. If it is in one or more valid CLUTs. This scheme helps to detect the common minimum unit **regions** area in scenarios such as those shown in Figure 5.9.

If a CBN participates in one or more CLUTs, it computes whether it is part of any invalid CLUT, i.e., whether one or more segments are in the EIT. If so, it communicates with the neighbour CBNs to share information about invalid CLUTs. The neighbouring CBNs then update their CLUT information. After that, if a CBN is still on at least one valid CLUT, it declares its state to be **true**.

Other Operators. The operation states for other operators are given in Table. 5.2. The formal definitions of the topological predicates in Table 4.12, stipulate that, in the case of SEGs, some topological operators (e.g., `area_disjoint`) are only **true** if all the member nodes (part of operands) satisfy the prerequisites. Others (e.g., `adjacent`) are only **true** if all the member nodes satisfy the prerequisites of a secondary operator (i.e., `area_disjoint` in case of `adjacent`), and there must exist some nodes that satisfies the requirements of the primary operator (i.e., `adjacent` in this example).

`adjacent`	:	**false**, **disjoint**, **segment_node** or **OP_not_applicable**
`edge_disjoint`	:	**false**, **true**, or **OP_not_applicable**
`meets`	:	**false**, **disjoint**, **commonBoundaryNode**, or **OP_not_applicable**
`edge_inside`	:	**false**, **true**, **unknown** or **OP_not_applicable**
`intersects`	:	**true**, **unknown**, or **OP_not_applicable**
`border_in_common`	:	**true**, **unknown**, or **OP_not_applicable**
`area_disjoint`	:	**false**, **true**, or **OP_not_applicable**

Table 5.2: Possible membership states of operators that require, in addition to a local GIT look-up, GIT information from their one-hop neighbours to compute their membership state

In the case of `adjacent`, the state **disjoint** denotes that it satisfies the prerequisites for `area_disjoint` and is not part of a CBS. The state **segment_node** denotes that it fulfils the prerequisites for `area_disjoint` and that it is part of a CBS. The state **false** denotes that a node does not satisfy the prerequisites for `area_disjoint` as explained above. After checking whether the operands are `area_disjoint`, a CBN computes whether it is a **segment_node** using the information it has obtained while considering the existence of CLUT in case of `area_disjoint`. The test for CBS was explained in the description of `border_in_common`.

For the `edge_disjoint` operator, the state **false** denotes that a node is part of a CBS or else that it belongs to the interior of one of the operands and to the interior or the boundary of the other. For `edge_inside`, the state **false** denotes that a node is either part of a CBS or does not satisfy the prerequisites for `area_inside`. For `intersects`, a node that belongs to only one of the operands declares its state to be **unknown**. All other nodes in the boundary of one of the operand and the interior of other, or in the interior of both operands, declare their state to be **true**. The CBNs test whether they are part of shared area between operands. A CBN declares its state to be **true** if it belongs to the CLUT or if it has neighbouring nodes that belong to the interior of both r and r', or else to boundary of one operand and to the interior of other. Otherwise, a CBN declares its state to be **unknown**.

In the case of `meets`, the state **disjoint** denotes that the node is not a CBN and satisfies the basic prerequisites for `edge_disjoint`. The **commonBoundaryNode** state denotes that it fulfils the prerequisites for `edge_disjoint`, and that it is a CBN and not part of a CBS. The **false** state denotes that a node either does not fulfill the basic prerequisites for `edge_disjoint` or else is a part of a CBS.

5.4.2.2 Evaluation of Simple Boolean-Valued Tasks on Multi-Element Geometries

A multi-element geometry of type **regions** may consist of elements with or without holes, and *unit* cycles in any combination. Induced geometries are dynamic, i.e., their shape, size, location may change at each evaluation period. In addition, it is not known in advance, whether the induced geometry is a SEG or a MEG. Therefore, in WSNs, a node cannot be made geometry-element aware, i.e., whether it belongs to single-element geometry or whether it belongs to one element of a geometry consisting of more than one disjoint elements.

The aggregate calculation problem can, therefore, be formally redefined as:

Given the geometry of m elements, with each element comprising n_k nodes $k = 1...m$ but not necessarily the same for all elements, and with each node $i \in n_k$ holding a value v_i, determine the aggregate count of these values in a distributed manner.

The formal definitions of the topological predicates in Table 4.13, stipulate that, in the case of MEGs, some topological operators (e.g., `area_disjoint`) are only **true** if all the member elements satisfy the prerequisites. Others (e.g., `adjacent`) are only **true** if all the member elements satisfy the prerequisites of a secondary operator (i.e., `area_disjoint` in case of `adjacent`), and if there is one pair of elements that satisfies the requirements of the primary operator (i.e., `adjacent` in this example).

As explained in Section 5.4.2.1, for topological relationships between SEGs (with the exception of `equals` and `not_equals`) selecting a single leader in one of the operands for performing the aggregation phase is a correct and energy-efficient solution. In the case of MEGs, however, one also needs to decide which element is to supply the leader node. Moreover, the nodes of all the elements of that operand must know who the leader is for them to participate in aggregation.

One strategy for that is to elect a leader node at the level of the entire sensor field. All network nodes participate in the election of the leader and in this way all are aware of it. This strategy is not efficient because, firstly, it results in more energy consumption not only in the leader election and tree construction but also in aggregation as the nodes in all elements transmit their state towards the central leader and, hence, involve more relay nodes; and, secondly, it restricts the possibility of evaluation of different spatial tasks efficiently at different parts of the WSN.

Another strategy is to elect a leader node that belongs to an element of an operand (e.g., the node having the smallest ID or the largest energy stock). This strategy is also not energy-efficient for leader election and aggregation, because it may involve too many relay nodes in so far as the operand elements (e.g., as is likely for induced geometries) may spread widely over the WSN. In addition, because the shape, number of disjoint elements and extent of induced geometries may change at each evaluation period, and, therefore, this strategy may require the election of a leader at every evaluation period, which would generate network traffic and, ultimately, restrict the possibility of evaluating different spatial tasks at different parts of the WSN.

A more efficient solution is to have leaders at the SEG level and at the top-level. Firstly, at the level of individual geometry elements, for each element, node-level outcomes are partially aggregated on their way to a leader node into an SEG-level outcome. Secondly, at the level of multiple geometry elements, the SEG-level outcomes are aggregated at a *first-level leader* into the final aggregated result. Recall, from Section 5.4.1, that a *first-level leader* node will have already been elected during task dissemination.

Figure 5.10 shows an example scenario for the evaluation of the adjacent operation between r' and R, where R is a MEG and consists of SEGs $r1$ and $r2$. The labels adjacent to each node represent the node membership state for the `adjacent` operation, and the labels adjacent to the arrows (on the way from SEG leaders to the first-level leader and from first-level leader to the gateway) represent the state as aggregated by SEG leaders and by the first-level leader. Figure 5.10 shows that aggregation is performed over R. Detailed discussion as to which operand is selected is delayed till Section 5.4.2.5. The SEG leader of the r_1 geometry element is located at the lower-left corner and computes the geometry-element state to be **disjoint** after aggregation because all the nodes that are part of geometry-element have the **disjoint** membership state. The SEG leader of r_2 is located at the top-right corner and computes the geometry-element state to be **true** after aggregation because some nodes have declared their membership state to be **segment_node**, and others have declared it to be **disjoint**. After receiving the partial aggregated results from the SEG leaders, the first-level leader aggregates them and declares the result to be **true**, because it has received both **true** and **disjoint** membership states.

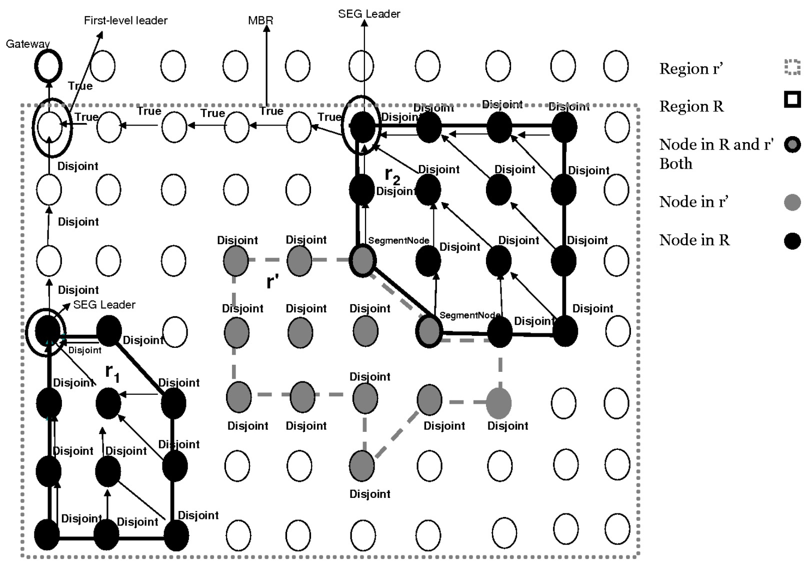


Figure 5.10: Example scenario: Adjacent relationship between two geometries of type **regions**. Operand r' is of type SEG and R of type MEG

5.4.2.3 Evaluation of Complex Boolean-Valued Tasks on Multi-Element Geometries

Recall from Section 5.2 that complex topological tasks are complex algebraic expressions, consisting of more than one topological operator connected by **and**, **or**, and **not**.

Complex tasks are difficult to evaluate inside the network. Reasons include: (1) there being more than one Boolean-valued operator, (2) operands may be SEG or MEG, with or without hole, and possibly *Unit* **regions**, (3) each operation in a task may have shared operands (e.g., consider complex task in Figure 5.11 where operands of both operations are the same), (4) operations in a task may have distinct (in terms of data type, geometry ID, and independence in terms of sensor space where they are located) operands, and (5) the Boolean-connectives must be handled as well.

It would not be efficient to use the *count* as an aggregation operator for a complex task. As discussed in the previous sections, the membership states associated with each operator can be computed by performing localized processing. These local outcomes are then aggregated. For example, for `adjacent`, the possible states that an MBR node can declare are **disjoint**, **false**,

segment_node, and **OP_not_applicable**. For `vertex_disjoint`, the possible states are **true**, **false**, and **OP_not_applicable**. For the evaluation of a complex task, one would, therefore, need to separately aggregate each membership state for each of the operators that occur in the task. For example, in the case of the complex task in Figure 5.11, each node on its way to leader node needs to compute separately the count of seven membership states (as described above for `vertex_disjoint`, a node can be in one of the three possible membership states and for `adjacent` in one of the four possible states) to be forwarded to its parent. The number of states will increase with an increase in the number of operations in the complex task, which leads in turn to an increase in the message length. Each child node may end up needing to transmit more than one message containing membership states to its parent for aggregation because of restrictions on the length of individual messages.

Furthermore, for a complex task, the aggregation strategy described in Section 5.4.2.2 cannot be used without modification. Firstly, each parent node in the path to the leader node must relate the membership states which it has received from child nodes with specific operators in the task. Secondly, some strategy is also needed for applying the Boolean connectives.

```
(NOT(r VertexDisjoint r’) AND
(r Adjacent r’))
```

Figure 5.11: Complex Task 1

```
(NOT(r VertexDisjoint r’) AND
(d Adjacent e))
```

Figure 5.12: Complex Task 2

5.4.2.4 Compression and Aggregation Schemes

For efficiently evaluating complex tasks, this dissertation contributes a novel approach comprising a compression scheme that represents the node's membership state for each of the operators in the task, and an aggregation scheme over this compressed representation that yields the final outcome.

The compression scheme merges the operation states, for each operator in the task into node-level task state. In this dissertation, node membership state is also referred to as node operation state. Table 5.3 shows how the possible operation states are coded into an octal digit (i.e., a 3-bit representation). Each constant represents one *operation state*. Let a task have n operations, and let $[S_{OP_1}, S_{OP_2}, \ldots, S_{OP_n}]$, where each S_{OP_i} denotes an operation state in Table 5.3, be the sequence of computed operation states. A node-level task state is generated by initializing it to 0 and then iteratively left-shifting its current value by three bits and `BitwiseOR`-ing each S_{OP_i}. This scheme allows the use of the `BitwiseAND` as an aggregation operator instead of *count*.

Consider a simple example of how this aggregation strategy works in Figure 5.13. The numbers inside the node represent the operation state, and the numbers adjacent to the node

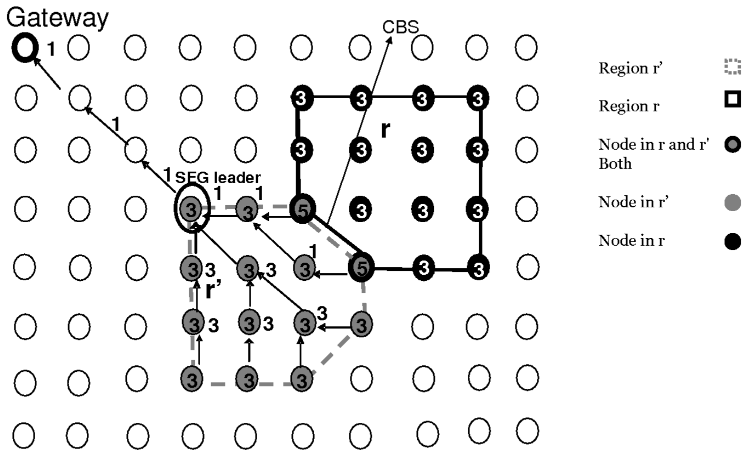


Figure 5.13: Running example to describe computation of `adjacent` relationship between two geometries of type **regions**

represent the result after performing the aggregation over its own and partial aggregated results received from the child nodes. For two SEGs to be `adjacent`, two or more nodes in those SEGs must have declared a **segment_node** membership state (i.e., 5_8) whilst the others must have called a **disjoint** membership state (i.e., 3_8). `BitwiseAND`-ing a bag each of whose elements is either 5_8 or 3_8, yields 1_8, i.e., **true**.

The compression scheme has three main benefits. Firstly, communication energy is saved by compressing the information which a node has to transmit towards the SEG-leader, and from the latter towards the *first-level* leader. Secondly, the compression scheme decreases the number of computations to be made by each node. Thirdly, this compression scheme allows aggregation to be based on a single operand (i.e., a node needs to participate at most once in the aggregation process at the geometry element level). The MBR nodes that participate in the aggregation are those with at least one operation state other than **OP_not_applicable**. A node participates at most once in an aggregation process at the geometry-element level. A node elects its SEG leader based on only one of the operands of only one of the operators in a task for which the operation state is not **OP_not_applicable**. Detailed discussion is given in Section 5.4.2.5.

The compression scheme, therefore, allows the same aggregation scheme described in Section 5.4.2.2 for the computation of simple tasks on multi-element geometries. The aggregation problem can, therefore, be redefined as follows:

Given a geometry comprising m elements, with each element comprising n_k nodes where $k = 1...m$ but not necessarily the same for all elements, and with each node $i \in n_k$ holding a task value v_i that is encoded as per Table 5.3, determine the `BitwiseAND` *of these values in a distributed manner.*

The hierarchy of leaders to be elected as described in Section 5.4.2.3 is sufficient for the computation of a complex task. There is one *first-level* leader and a number of SEG leaders varies depending on the *task complexity*. *Task complexity* is based on the number of operations

OP state	Encoding
true	1_8
false	0_8
disjoint	3_8
unknown	3_8
segment_node	5_8
commonBoundaryNode	5_8
OP_not_applicable	7_8

Table 5.3: Representing operation states

in the task, whether the operation only needs a local GIT lookup or, additionally, the collection of GIT information from the node's neighbours, the operands are SEG or MEG and, finally, how independent, in terms of the sensor space, the operand elements of an operation are with respect to other operations in the complex task. Let us suppose that operands r, r', d and e are SEG of type **regions** in the tasks given in Figure 5.11 and Figure 5.12. The complexity of the task in Figure 5.11 is smaller compared to the task in Figure 5.12, especially, in the case where the nodes that are part of the r and r' are not part of d and e, or vice versa. As, in such a case the task involve operands that are disjoint from each other and may be located far apart in the network from each other.

To compute the partial result at the geometry-element level, the nodes task states are aggregated on the way to the SEG leader node by using `BitwiseAND` as the aggregation operator. After performing the aggregation, the SEG leaders transmit the result to the *first-level* leader. Once the distributed evaluation sub-task is concluded, results are routed towards the *first-level* leader by the SEG leaders along the tree that was established during task dissemination. The *first-level* leader performs the final result processing. The reader is referred to Section 5.4.3 for more on the result processing phase.

Let G, h, D and e be four induced geometries of type **regions** where G and D are MEGs, and h and e are SEGs, G consists of elements $g1$ and $g2$, and D consists of elements $d1$ and $d2$. Now let the task in Figure 5.14 be transmitted for evaluation. All nodes that are part of the MBR evaluate the task after receiving it in the task message. These nodes first perform distributed membership evaluation to compute the operation state for each of the three predicates in the task, and then compute the task state.

```
(NOT(G VertexDisjoint h) AND
(D Adjacent e) AND (G AreaInside D))
```

Figure 5.14: Complex Task 3

The evaluation of `vertex_disjoint` is first carried out by all MBR nodes. The MBR nodes that are not part of an element of G or of h declare their operation state to be **OP_not_applicable**. An MBR node that belongs to an element of G or to h, but not to both, declare its state as **true**. Otherwise, an MBR node that belongs to an element of G (i.e., $g1$ or $g2$) and to h declares its state to be **false**. Recall, from the formal definitions of the topological predicates in Table 4.13, that `vertex_disjoint` on MEGs requires all elements of both geometries to be `vertex_disjoint`.

After the evaluation of the first operation, the nodes evaluate the second (i.e., `adjacent`) operation. Recall, from Section 4.2.1.6, that `adjacent` on MEGs requires at least one element of each geometry to be `adjacent` and all other elements of both geometries to be `area_disjoint`.

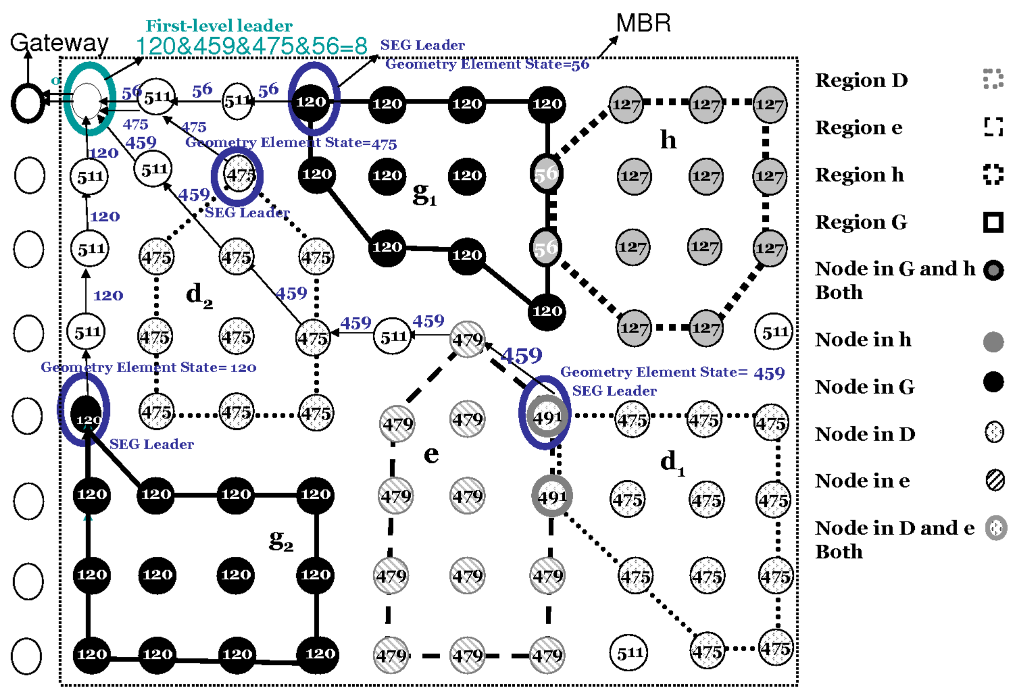


Figure 5.15: Example scenario: To evaluate the complex task in Figure 5.14

After the evaluation of `adjacent`, the nodes evaluate the `area_inside` operation. Recall, from the formal definitions of the topological predicates in Table 4.13 (Section 4.2.1.6), that `area_inside` on MEGs requires all elements of G to be `area_inside` one or more elements of D.

Table 5.4 depicts all the possible combination of operation states for the spatial predicates specified in task, their octal-encoded representation, and the corresponding task state computed after applying the compression scheme. We use as examples the geometries in Figure 5.15. We now describe how, in the top most row of Table 5.4, the task state 120 is computed from the three operation states i.e., **true**, **OP_not_applicable** and **false**. Recall that a node-level task state is generated by initializing it to 0 and then iteratively left-shifting its current value by three bits and `BitwiseOR`-ing the individual states. A node acquire a task state of 1 after assigning the first operation state to the task state, for second operation state it left-shifts the task state (i.e., 1) three bits and after `BitwiseOR`-ing it with 7 the task state becomes 15, and lastly, for the third operation state it left-shifts the task state (i.e., 15) three bits and after `BitwiseOR`-ing it with 0, the task state becomes 120.

After the distributed membership evaluation, the next step is aggregation. The only nodes that participate in aggregation are those with at least one operation state other than **OP_not_applicable**. Recall that a node only needs to participate at most once in an aggregation process at the geometry-element level. A node selects the operand of which it is a member and its state for that operation is not **OP_not_applicable** whilst scanning the task from left to right. The

OP1State : OP2State : OP3State	OP Constant Rep.	Task State
true: OP_not_applicable: false	1 : 7 : 0	120
false: OP_not_applicable: false	0 : 7 : 0	56
OP_not_applicable: false: false	7 : 0 : 0	448
OP_not_applicable: disjoint: false	7 : 3 : 0	472
OP_not_applicable: segment_node: false	7 : 5 : 0	488
OP_not_applicable: OP_not_applicable: false	7 : 7 : 0	504
true: OP_not_applicable: true	1 : 7 : 1	121
false: OP_not_applicable: true	0 : 7 : 1	57
OP_not_applicable: false: true	7 : 0 : 1	449
OP_not_applicable: disjoint: true	7 : 3 : 1	473
OP_not_applicable: segment_node: true	7 : 5 : 1	489
OP_not_applicable: OP_not_applicable: true	7 : 7 : 1	505
true: OP_not_applicable: unknown	1 : 7 : 3	121
false: OP_not_applicable: unknown	0 : 7 : 3	59
OP_not_applicable: false: unknown	7 : 0 : 3	451
OP_not_applicable: disjoint: unknown	7 : 3 : 3	475
OP_not_applicable: segment_node: unknown	7 : 5 : 3	491
OP_not_applicable: OP_not_applicable: unknown	7 : 7 : 3	507
true: OP_not_applicable: OP_not_applicable	1 : 7 : 7	127
false: OP_not_applicable: OP_not_applicable	0 : 7 : 7	63
OP_not_applicable: false: OP_not_applicable	7 : 0 : 7	455
OP_not_applicable: disjoint: OP_not_applicable	7 : 3 : 7	479
OP_not_applicable: segment_node: OP_not_applicable	7 : 5 : 7	495
OP_not_applicable: OP_not_applicable: OP_not_applicable	7 : 7 : 7	511

Table 5.4: Expected operator and task state for the complex task in Figure 5.14, by a node in the scenario in Figure 5.15

process of selecting an operand is now described in detail.

5.4.2.5 Selection of the Geometric Operand for Aggregation

This section addresses the question as to which operand of an operation should be selected for performing aggregation.

The conditions are as follows. If the node has computed no operation state other than **OP_not_applicable**, then it need not participate in the aggregation. If the node has computed exactly one operation state other than **OP_not_applicable**, it participates in the aggregation on the basis of one of the operand geometries of the corresponding operation, as explained below. If the node has computed more than one operation state other than **OP_not_applicable** and belongs to more than one operand geometries involved, it participates in the aggregation but still on the basis of only one of those operand geometries, as now explained.

For participating in the aggregation, a node scans the operations in a complex task from left to right and selects the first one whose computed operation state is different from **OP_not_applicable**. Once the operation is selected, the next step is to select the operand. If the operation is `equals`, or `not_equals`, then a node selects the operand it is a member of, and if it is a member of both, it selects the LHS operand. If the operation is `area_inside`, `edge_inside`, or `vertex_inside`, then it selects the LHS operand. It can be seen in Table 4.13 that these predicates evaluate to **true** even if some elements in the RHS operand contain more than one element from the LHS operand or if some element of the RHS operand does not contain any element of the LHS operand. For the other operators, i.e., `vertex_disjoint`, `edge_disjoint`, `area_disjoint`, `meets`, `adjacent`, and `border_in_common`, any of the operands can be selected. So in this dissertation, we selected the LHS operand. The topological operators are, therefore, classified into two groups: the first one, comprising `equals` and `not_equals`, has higher precedence; the second one, comprising all other operators, has lower precedence. Therefore, before the task is disseminated, the gateway rewrites the submitted task in order to ensure the higher precedence operators appear leftmost in the task.

Therefore, in the case of the example scenario in Figure 5.15, the nodes that belongs to an element of G participate in the aggregation process of the corresponding element. Otherwise, the nodes that do not belong to any geometry-element of G and instead belong to an element of D participate in the aggregation process of the corresponding element of D.

The aggregation scheme that allows for the selection of SEG leader and aggregation of information at the single-geometry element level is discussed in Section 5.4.2.6.

5.4.2.6 Strategies for Leader Election and Aggregation

To perform the aggregation at the geometry-element level, we make use of two algorithmic strategies and compare the results. The first strategy is a randomized grouping gossip-based technique and the second is a tree-based technique.

The first algorithmic strategy for aggregation that we consider is inspired by the *distributed random grouping* (DRG) approach [CPX05] for computing aggregations described in Section 2.3. In [CPX05] this approach is used to compute an aggregate function like the average of node values within a given error range, where the values to be aggregated are physical measurements.

We have used DRG with modifications to adapt it to our setting. It is effective and very efficient in the presence of topological changes since it does not have to pay any cost for constructing, and frequently maintaining, an aggregation tree. Although it involves additional communication cost, it comes with greater reliability in situations where consideration of each node value is important towards the computation of the result as is the case with Boolean-valued operations. It is effective and efficient even in the presence of MEGs, of holes in a geometry, of irregularly-shaped geometries, and of overlapping geometries. It is also resilient to node and message failures, time synchronization delays, and node additions.

Tree-based algorithms tend to be energy-efficient but are not resilient in the presence of topology change. These issues can be addressed by using tree maintenance approaches like those by Madden et al. [MFHH02]. However, in the case of phenomena that are continuously evolving, one would need to reconstruct the aggregation tree at every evaluation period.

Gossip-based Technique. We now provide more details of our gossip-based scheme inspired by the *distributed random grouping* (DRG) approach that underpins the selection of geometry-element leaders and the aggregation of local outcomes.

The process involves a number of rounds, in each of which a node can be in one of seven states, viz., **idle**, **leadershipPending**, **groupLeader**, **membershipPending**, **groupMember**, **tentativeConvergence**, **converged**. The process works as follows. In each round, some nodes are elected as **groupLeader** with some probability and the other nodes take on the role of **groupMember**. Initially, each node is in the **idle** state for a given time. Once this state expires, the node generates a random value that denotes its bid to be **groupLeader**. If the value is less than a probability threshold, the node moves to the **leadershipPending** state and broadcasts the bid to move to the **groupLeader** state, otherwise it moves back into the **idle** state and waits for the associated time to expire once more. If the bid for leadership is successfully transmitted, the node moves to the **groupLeader** state, otherwise, it moves back to the **idle** state.

Upon receiving a bid from a candidate group leader, those nodes that are in the **idle** state move to the **membershipPending** state and transmit their partially aggregated results, as well as SEG leader information, to the candidate group leader, after which they move to the **groupMember** state. If transmission failed, they move back to the **idle** state. In the first round, group members declare themselves SEG leaders. After transmitting this report, such a node move to the **groupMember** state. In the **groupMember** state, a node waits for a specified duration for the reply from its group leader. Upon receiving the reports from group members, a node in the **groupLeader** state aggregates using `BitwiseAND` the reported results and identifies the minimum node ID amongst its own and proposed SEG leader IDs received in the group member reports. Upon expiry of the wait period for members to report, the group leader node broadcasts a reply message to its group members containing the computed aggregated result and the updated SEG leader ID. Upon receiving such a reply message from its group leader, a node in the **groupMember** state performs a `BitwiseAND` on its own outcome with the outcome sent by the group leader and updates its SEG leader ID to the minimum node ID amongst its own and the one received in leader reply.

As a stopping criterion for this iterative process, we use a technique that allows each node to determine if convergence has occurred. For this purpose, each node monitors any change in its aggregated result and its SEG leader ID. If these do not change after a specified number of consecutive rounds, the node moves to the **tentativeConvergence** state.

Upon changing the state to **tentativeConvergence**, it waits in this state for a specified duration. If it does not receive any request from the group leader during that time, the node moves to the **converged** state when the waiting period expires. Otherwise, it moves to the **membershipPending** state. Therefore, at the end of each round, a node in the **groupLeader** or the **groupMember** states, upon receiving the required information and performing the required computation, increments the count of tentative convergence rounds if no changes have taken place since the last round, otherwise it sets the count back to zero. At the end of each round, a node checks whether it can move to the **tentativeConvergence** state. If it cannot, it moves to **idle** and engages in another round.

To address the risk of packet loss due to collision, wait period has been used. When the wait period expires, the entity changes state as described above even if it does not receive a packet for which it was waiting. The purpose is to differentiate the scenarios in which the `GROUP_LEAD` does not receive any membership message because of packet loss, or because neighbouring entities have converged, or because neighbouring entities are already in either the `GROUP_LEAD` or the `GROUP_MEMBER` states. In such a case, as there will be no change in `GROUP_LEAD` results, at the end of its wait for membership reports, incrementing the tentative convergence round count would impact on the successful convergence of the geometry and not incrementing could prevent an entity from converging (i.e., keep on trying). In order to handle such a situation, if the group leader entity does not receive any group membership message, it increments the tentative convergence round count by less than one (i.e., 0.4), instead by one as explained above .

Tree-based Technique. The second algorithmic strategy for aggregation that we consider is tree-based. A tree-based aggregation algorithm works by creating a tree rooted at a root node. This tree acts as a routing tree for the incremental aggregation of data. Any of the geometry-element nodes can be used as a root node. This root node can be chosen either randomly or through a distributed leader election algorithm like the one proposed by Dulman et al. [DHS02]. In this dissertation, the root node, i.e., the SEG leader, selected is the closest node to the gateway. The tree construction phase is subdivided into two phases: leader election and tree construction. In the leader election phase, all nodes that belongs to the geometry broadcast their ID to their neighbours. Upon receiving the information each node compute whether there is a neighbouring node having minimum ID then itself. If it finds any, it sets its status as non-leader node and sets wait timer for information from leader node, upon expiry of wait time, if it does not receive information about leader node it sends the *tree construction message* (TCM). Otherwise, if it finds that it has minimum ID, it starts the tree creation process by broadcasting a TCM. Before broadcasting the TCM, it sets the value of the *leaderID* and *sourceID* attributes to its own ID, its *level* attribute to zero, the *distance* attribute to its distance to gateway, and the *geometryID* attribute to the operand ID, on the basis of which it participates in the aggregation process.

Any node belonging to the same geometry-element that has not heard the TCM earlier, records the information in the message upon receiving it. It keeps the ID of the sender node as its parent node. After incrementing by one the *level* attribute in the message and setting the *sourceID* attribute to its own ID, the node broadcasts the message. If a node receives a TCM from some other node with a different *leaderID* and if the distance of this leader to the gateway is smaller (or if the distance is same but the *leaderID* is smaller) than the recorded *leaderID*, it updates its information, increments the *level* attribute in the message by one and broadcasts the message. Otherwise, it ignores the message. If a leader node receives a TCM with better leader information (i.e., with a smaller distance to the gateway or with a smaller *leaderID*), it records the information and changes its state to non-leader node. This process continues until all the nodes in the geometry-element have been assigned a parent node, leader information and a level value.

Upon the reception of a TCM, each node waits for a specified duration. The node refreshes the wait period upon the reception of a TCM from some other neighbour or the same neighbour. Upon the expiry of the wait period, each node registers with its parent node. This allows each node to learn how many children nodes it has. Nodes with no child nodes are leaf nodes in the resulting tree.

After the tree has been set up, the process of aggregation of membership states is started at the leaves. Leaf nodes insert their task state in a single packet and forward it to their parent. Each parent, depending on its *level* and the number of children it has, waits for a specified period to receive the packets from its children. Each parent then computes a partial outcome by applying the `BitwiseAND` operation to its own and to the values received from its children.

It then forwards the result to its parent. This computation of partial aggregations of task state continues at each level of the tree and eventually the aggregated task state, representing the task state at geometry-element level, is computed at the SEG leader node.

Figure 5.15 shows the node-level task state computed by each node and the element-level task state (i.e. the partial aggregated result at the SEG leader) computed after applying the aggregation scheme. In Figure 5.15, there are four SEG leaders which transmit the partial aggregated task states towards the *first-level* leader.

5.4.3 Result Processing

As discussed earlier in Section 5.4.2.6, after the aggregation operation, the SEG leaders transmit the aggregated task state towards the *first-level* leader. The *first-level* leader then performs the `BitwiseAND` operation on the task states received from SEG leaders and computes the aggregated task state. The first-level leader then computes the operation states from the aggregated task state by decompressing the task states. These operation states represent the final aggregated result for each spatial predicate occurring in the task. The next step is to apply the Boolean connectives that occur in the task. For this purpose, each spatial predicate in the task is replaced by an operation state and the corresponding Boolean connectives rules are applied to compute the final outcome. If the outcome is **true** then **true** is transmitted transmitted by the first-level leader towards the sink, otherwise, **false** is transmitted. Table 5.4.3 defines how **and** and **or** correspond to bitwise operations depending on the operations states they apply to. In Table 5.4.3, ¬(**false**) denotes any operation state in Table 5.3 other than **false**, and ¬(**true**) denotes any operations state other than **true**. In the case of **not**, it yields **true** if the operation state is any other than **OP_not_applicable** and **true**.

$OP1_{Res}$	$OP2_{Res}$	OP for **and**	OP for **or**
false	**false**	`BitwiseAND`	`BitwiseOR`
¬(**false**)	**false**	`BitwiseAND`	`BitwiseOR`
¬(**false**)	¬(**false**)	`BitwiseOR`	`BitwiseAND`

Table 5.5: Bitwise operations for **AND** and **OR** connectives

In Figure 5.15, after applying the `BitwiseAND` aggregation operator to all partial outcomes received from the SEG leaders, the *first-level* leader, computes the MEG task state to be 8. The *first-level* leader node then decompresses and computes each operation state from the task state (i.e., 8 represents 000|001|000 in binary) as (**false**:**true**:**false**). It then applies the connectives after substituting these operations states (i.e., **not** (**false**) **and** (**true**) **and** (**false**)) in place of the spatial predicates in the task given in Figure 5.14. For applying the connectives, the *first-level* leader make use of the classical rules for each connective which yields the value **false**. The *first-level* leader node then transmits the computed result towards the sink.

In the example scenario in Figure 5.15, geometry G has two elements $g1$ and $g2$ and $g2$ satisfies the `vertex_disjoint` relationship whereas $g1$ does not. Therefore, the final aggregated state for `vertex_disjoint` is **false**. Geometry element $d1$ satisfies the `adjacent` relationship with e, but not with $d2$, $d2$ satisfies the `area_disjoint` relationship with e only. Therefore, the final aggregated state for `adjacent` is **true**. In the case of `area_inside`, $g1$ and $g2$ do not lie within one or more geometry elements of D, therefore the final aggregated result for `area_inside` is

false.

5.5 Evaluation Components for Spatial-Valued Tasks

This section describes how a task containing spatial-valued operators is evaluated in a distributed manner. Note that this section only discusses spatial-valued tasks whose operands are **regions**. Furthermore, in all the figures in this section which provide illustrative example scenarios, it is assumed (unless specified otherwise) that the nodes are deployed in uniform grid, that a node has a maximum neighbourhood of 8 nodes. For the description of the algorithmic strategy for the evaluation of spatial-valued tasks, it is assumed that r and r' denote SEG **regions** values, and R and R' denote MEG **regions** values whose constituents are SEGs, denoted by r and r', resp.. Furthermore, it is assumed that r (or R) represents the LHS operand and r' (or R') the RHS operand of binary spatial-valued operations represented in infix notation.

As with Boolean-valued tasks, the task evaluation process can be broken down into three phases viz., *task dissemination*, *distributed task evaluation*, and *result processing.*

5.5.1 Task Dissemination

Task dissemination of a spatial-valued task is identical to that of a Boolean-valued task as described in Section 5.4.1. Therefore, we focus on *distributed task evaluation* and *result processing* for the remainder of this section.

5.5.2 Distributed Task Evaluation

Spatial-valued tasks comprise operators that return a spatial value. The logical structure of the process that evaluates a spatial-valued task is a sequence involving: *membership evaluation*, and *geometry derivation. Membership evaluation* is the process whereby a node computes whether it satisfies the conditions for membership in the derived geometry defined by the spatial value returning operator. *Geometry derivation* is the process whereby event nodes (i.e., those that satisfy the membership requirements of the derived geometry) compute whether they are part of the boundary or the interior of the derived geometry. For some operations, such as `contour`, `common_border`, and `vertices`, additional computation is not required for geometry derivation as these operations return a value of type **lines** or **points**, which do not have an interior. Therefore, in the case of spatial operations which result in a spatial value of type **points** or **lines**, the boundary related information is implicitly set by nodes without the need for further processing along with other information about the derived geometry in GIT. For other operators, details are provided in Section 5.5.2.1.

Recall, from Section 5.3, that an entry in the GIT is a quadruple $\langle$*GeometryID*, *DataType*, *BoundaryNode*, *TTL*$\rangle$. The *geometryID* for the new derived geometry is provided as part the task message. For spatial-valued operations, type inference (i.e., reasoning from the input types) suffices to derive the result data type. For example, taking the `plus` of two geometries of type **regions** results in a new geometry of type **regions**. A node sets the *BoundaryNode* attribute to **true** if it is part of the boundary of the derived geometry, otherwise it sets the attribute value to **false**. The TTL for a new derived geometry is computed as the minimum of TTL of the operands for all binary spatial-valued operations. In the case of unary spatial-valued operations, the TTL of the resulting geometry is set to that of the operand. The membership evaluation process for each of the spatial-valued operation is now explained.

5.5.2.1 Evaluation of Simple Spatial-Valued Tasks on Multi-Element Geometries

An MBR node declares its operation state as **false** if the node does not belong to either of the operands of an operator or if the node does not satisfy the spatial data type requirement for the evaluation of the operator. For many operators (e.g., `plus`, `vertices`), membership evaluation only requires a local GIT look-up to check whether the node satisfies the prerequisites for the operation.

Plus. In the case of the `plus` operator, if an MBR node is part of one or both operands, it declares itself as belonging to the derived geometry, i.e., an event node, by setting its operation state to **true**. Otherwise, an MBR node sets its operation state as **false**. CBNs must perform localized decision-making to compute their edge state. A CBN requests information from its neighbours that have a **true** operation state. Upon reception of the replies, a CBN node makes itself the origin of a circle and partitions its neighbouring event nodes as lying on a quadrant or an axis based on their location. It then uses the modified T-Fit [JF08] boundary detection algorithm to compute its edge state.

A boundary node that belongs to only one operand with a **true** operation state sets its edge state to **true**.

As the `plus` results in the union of two geometries. In the case of `plus`, a boundary node that belongs to only one of the operands checks the entries in its EIT for any neighbour that belongs to that operand. If a node finds such information in the EIT, it appends that information to its EIT, after updating the GID of the tuple with the GID of derived geometry. Recall, an entry in the EIT is a pair $\langle GID, N_{ID} \rangle$. If a node is a CBN, it checks tuples in its EIT that are common to both operands and only appends those tuples to its EIT, after updating the GID of the tuple with the GID of the derived geometry.

Vertices. The `vertices` operation returns a new spatial-valued geometry of type **points**. It is a unary operation. Nodes that belong to the boundary of the **regions** value set their operation state to **true**. Nodes that belong to the interior of the **regions** value set their operation state to **false**.

For other operations (e.g., `minus`, `intersection`, `contour`, `common_border`), the participating nodes engage in localized decision-making to compute whether they belong to the derived geometry.

CommonBorder. The `common_border` operator returns a new geometry of type **lines** containing the common boundary segments of operands. In the case of `common_border`, CBNs perform localized decision-making to compute whether they belong to a CBS. If they do, the CBNs declare themselves to belong to the derived geometry by setting their operation state to **true**.

All other nodes set their state to **false**. The algorithm for the detection of CBS was described in Section 5.4.2.1 for the `border_in_common` operator.

Intersection. In the case of the `intersection` operator, non-CBNs compute their operation state using the information available in their own *GIT*: if the node belongs to the interior of both operands, or to the interior of one and the boundary of the other, then its operation state is **true**, otherwise **false**.

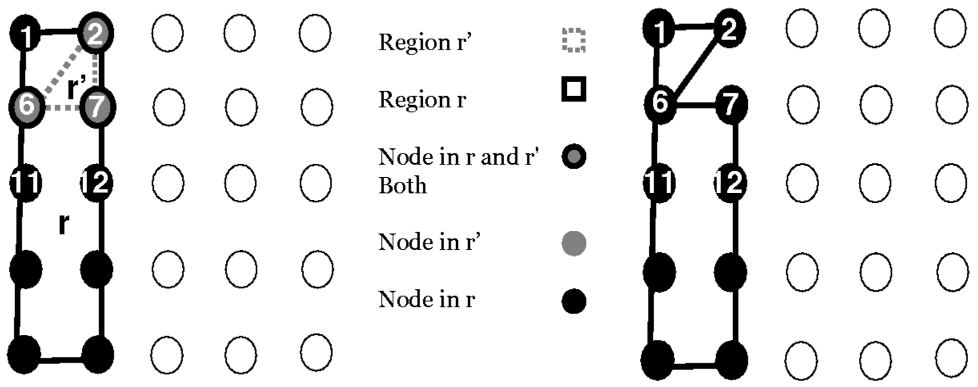
Let the space around a CBN be divided into 12 sectors of 30° each. A CBN transmits an information request to its one-hop neighbours. All one-hop neighbours that belong to one or both operands respond with an indication as to whether they belong to interior or boundary of each of the operands. Upon receiving this information, a CBN assigns them to the proper sector (based on their location) and computes the minimum distance neighbour in each sector. Its operation state is **true** if one or more of its closest one-hop neighbours nodes belong either to the interior of both operands or to the boundary of one operand and the interior of the other. If a CBN has no neighbouring non-CBN belonging to both geometries, then its operation state is **false** provided that it has less than two neighbouring CBNs, otherwise it must consider the existence of a shared minimum unit **regions** value. Let s_1, s_2, s_3 denote CBNs then a LUT between them may form a CLUT. A CBN can compute whether this is the case using the barycentric technique in [CM69]. The reader is referred to Section 5.4.2.1 for the description of how a CLUT is computed. If a CBN belongs to one or more CLUTs, its operation state is **true**, otherwise **false**. A node with a **true** operation state that does not belong to the interior of both operands sets their local edge state attribute to **true**, otherwise to **false**.

After membership evaluation, the nodes with operation state **true** compute whether they lie in the interior or the boundary using the local edge state attribute. If the local edge state attribute is **true**, the node declares itself a boundary node, and enters information about the derived geometry in its GIT, otherwise, it declares itself a non-boundary node.

After computing the local edge state attribute, the nodes that belong to the boundary of the derived geometry engage in local communication. The boundary nodes of the derived geometry having an entry for R or R' for a particular neighbour in its EIT, ask that neighbour whether it is part of the derived geometry. Upon reception of this information, it adds that information about that neighbour using the GID of the derived geometry to its EIT.

Minus. The `minus` returns the difference between two **regions** values. Let the operation be denoted r `minus` r'. A non-CBN node that belongs to r only or to the interior of r and to the boundary of r' sets its operation state to **true**, otherwise to **false**. As for `intersection`, in this case too a CBN must consider CLUTs. Let the space around a CBN be divided into 12 sectors of 30° each. A CBN transmits an information request to its one-hop neighbours. Upon receipt of this information, the CBN assigns them to the proper sector and computes the minimum distance neighbour in each sector. The CBN then computes whether it is part of any CLUT (created with neighbouring CBNs). If it is not and if it has at least one neighbour that belongs only to r amongst its closest neighbours, it sets the operation state to **true**. If the CBN is part of one or more CLUTs (formed with neighbouring CBNs) and if it has at least one neighbour that belongs only to r, and the segment between the CBN and that neighbour does not intersect any of the CLUTs, the CBN sets its state to **true**, otherwise to **false**. Consider the example scenario in Figure 5.16, geometries r and r' have three boundary nodes (2, 6 and 7), that are CBNs. All CBNs request information from their neighbours. Upon receiving request from CBNs, the neighbouring nodes send a reply. As shown in Figure 5.16 (a) CBN 7, has three neighbours (1, 11 and 12) which belongs to r only. The segment between itself and neighbour 1 intersects the CLUT. But the segments with the neighbours 11 and 12 does not intersect any of the CLUT, therefore, CBN 7 sets its operation state as **true**.

A node with a **true** operation state that is a CBN or belongs to the interior of r and the boundary of r', or belongs to r only and to its boundary, sets its edge state to **true**. Nodes that belongs to the interior of the first operand set their boundary state information to **false**.



(a) Example Scenario 01

(b) Derived geometry after evaluating r `minus` r' over example scenario in Figure 5.16

Figure 5.16: Example scenario-1 and associated derived geometry after evaluating operation r `minus` r'

Now the next step is to remove the segments that are created by the boundary nodes of derived geometry, but which should not be part of the derived geometry as they are lying in the spatial area of R'. Let s_i denote a node with the local edge state attribute set to **true**. In the case of the `minus` operation, as part of geometry derivation, an $s_i \in R_{on}$ performs a local EIT-lookup to find any tuple belonging to R. If it finds one or more tuples, it appends the same information in the EIT for the new derived geometry by updating the GID. Recall that a tuple in the EIT contains the *GID*, and the *neighbourID*. Where as, a node $s_i \in R_{on}R'_{on}$ or $s_i \in R_{in}R'_{on}$ needs information from their neighbours to compute which edges are lying in the spatial area of R' and enter such information in its EIT because edges from derived geometry that are created by its closest boundary nodes but are lying in the spatial area of R' must be removed. Consider a complicated example scenario in Figure 5.17(a) and the derived geometry in Figure 5.17(b).

For this purpose a node $s_i \in R_{in}R'_{on}$ also requests information from its neighbours when CBNs request information to compute their event state. Let sector 1 denote the sector in which the angle a node can make with its boundary neighbours is between 0°-30°(inclusive) by making itself the origin and let sector 12 denote the sector in which the angle a node can make with its boundary neighbours is between 330°-360°. Let A denote the sectors from 7 to 9 and B

denotes the sectors from 10 to 12.

A node $s_i \in R_{on}R'_{on}$ or $s_i \in R_{in}R'_{on}$ with neighbours in $R_{on}R'_{on}$ or in $R_{in}R'_{on}$ in sector A or B or both runs the following algorithm to compute among the neighbours (i.e., neighbours $\in R_{on}R'_{on}$ or neighbours $\in R_{in}R'_{on}$) with whom it does not form valid edges. Each s_i computes list of the rejected neighbours (with whom it does not form valid edges) and add this information to its EIT. After such computation if a node has a non-empty rejected edges list it transmits the information about the neighbours with which it not form edges in a message to its neighbours. Upon reception of this information, the neighbours that find their ID in the message remove their edge with the neighbour from which it has received message by adding this information to its EIT. Consider the example scenario in Figure 5.17, nodes 11, 12, 13, 35, 36 and 37 has neighbours that lie in either sector A or B or both.

The algorithm works as follows:

1. If neighbours lying in sector B of s_i forms a total of all neighbours lying in sector A and sector B. Node s_i needs to compute two neighbours with which it is making LUT. In case, s_i makes more than one LUT, it select the LUT in which it creates maximum angle by keeping itself at the origin. After the selection of LUT, it keep both neighbours. It removes edges with all other neighbours by adding their ID to the list of rejected neighbours. Node s_i then adds information from its list of rejected neighbours to its EIT.

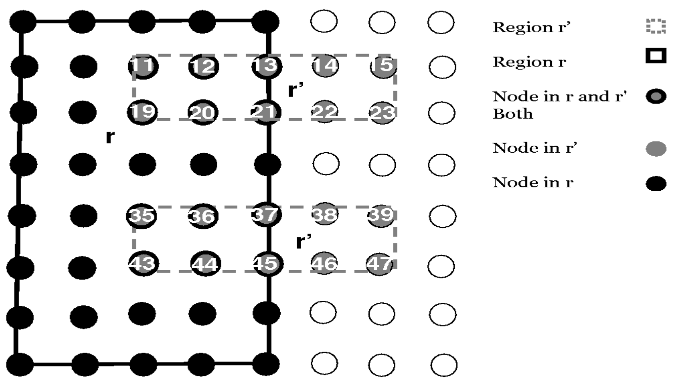
   - In the example scenario Figure 5.17, for example node 11 has three neighbours (12, 19, 20) in sector B that forms a total of all neighbours lying in sector A and sector B. Node 11 is part of three LUTs <11, 12, 19>, <11, 12, 20> and <11, 19, 20>. Among these LUTs, node 11 selects LUT <11,12,19> in which it creates maximum angle by keeping itself at the origin. Therefore, node 11 keeps its segments with node 12 and 19 and removes segment with 20.

2. If s_i does not satisfy condition (1) and has neighbours in sector A. Node s_i selects the neighbour among its neighbouring nodes in sector A with which it forms the minimum angle, and adds all other neighbours ID lying in sector A to its list of rejected neighbours. Node s_i then adds information from its list of rejected neighbours to its EIT.

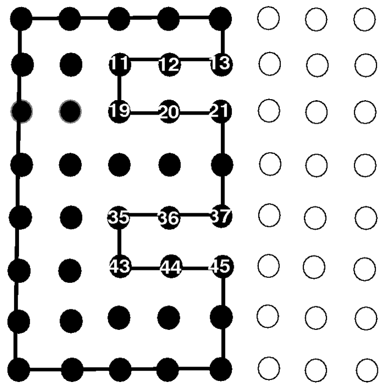
   - In the example scenario Figure 5.17, for example node 13 keeps its segment with node 12 only and removes its segments with nodes 20 and 21, and node 12 keeps its segment with node 11 only and removes its segments with nodes 19 and 20.

3. If s_i does not satisfy condition (1) and has neighbours in sector B, it selects the neighbour from B with which it forms the maximum angle and adds all other neighbour IDs in B to its list of rejected neighbours. A node then adds information from its list of rejected neighbours to its EIT.

   - In the example scenario Figure 5.17, for example node 12 keeps its segment with node 13 and removes its segments with nodes 20 and 21.



(a) Example Scenario 02



(b) Derived geometry after evaluating r `minus` r' over example scenario in Figure 5.17

Figure 5.17: Example scenario-2 and associated derived geometry after evaluating operation r `minus` r'

Contour. The `contour` operator computes the outer boundary of the **regions** value by ignoring inner boundaries. As a **regions** value with hole contains both an outer boundary and an inner boundary, the nodes in the outer boundary declare their operation states to be **true**. All other nodes declare their operation state to be **false**.

Nodes belonging to the operand of the `contour` operator engage in localized decision-making to compute the MBR of the **regions** value. Once the boundary nodes belonging to the **regions** value know about the minimum x-axis and y-axis, and the maximum x-axis and y-axis locations, a boundary node computes the four boundary segments of the MBR. Based on its location, the boundary node computes to which of the four boundary segments of the MBR it lies closest. The MBR segment to which it lies closest is called its *closest MBR segment.*

The boundary nodes then request their neighbours for information related to whether they belong to the interior or the boundary of the operand. Suppose that space around every boundary node is divided into 12 sectors of 30 degrees each. Upon receiving information from the neighbours, it computes the closest neighbour in each sector. The boundary node then computes whether any neighbouring nodes lie closer than itself to the *closest MBR segment.* If it finds that a neighbouring node belonging to the interior lies closer to the *closest MBR segment*, then it declares its operation state to be **true**. If it finds all neighbouring nodes in the direction of the *closest MBR segment* are boundary nodes, then by placing itself at the origin it computes the angle between any of the two neighbours in the adjacent sectors towards the *closest MBR segment.* If the angle is greater than or equal to 90, it declare its operation state to be **true**. It also sets its operation state to be **true** upon finding itself part of *closest MBR segment.*

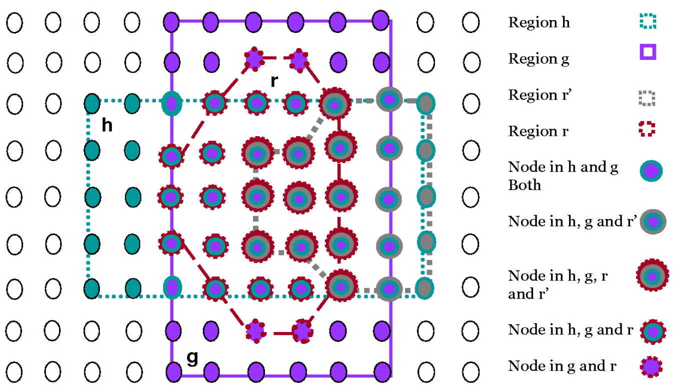
5.5.2.2 Evaluation of Complex Spatial-Valued Tasks

In the case of complex tasks, the explicit derivation of a geometry is performed after the evaluation of every operator in the task. The evaluation of each spatial-valued operator results in an *intermediate derived geometry* (IDG). Each MBR node keeps information about an IDG locally. Each node belonging to an IDG inserts information about that geometry in its GIT. Upon the evaluation of each successive operator in which the IDG is used as operand, all nodes belonging to IDG remove its IDG entry from the GIT if the operation state is **false**. Nodes with **true** operation state also update the attributes in the GIT entry of the IDG. Upon evaluation of the last operation in the task, the nodes with **true** operation state remove the entry for the IDG and create an entry for the final derived geometry.

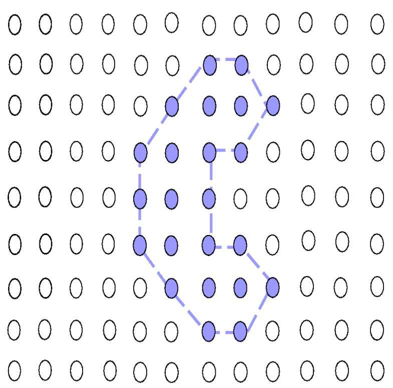
For example, consider the complex spatial-valued task in Figure 5.18. It is semantically equivalent to the task $(((r$ `plus` $h)$ `minus` $r')$ `intersection` $g)$ in infix notation. Let r, h, r' and g denote SEG **regions** values. Figure 5.19(a) provides an example scenario for the evaluation of the complex task in Figure 5.18 and Figure 5.19(b) shows the resultant derived geometry after the evaluation of the complex task over the geometries in Figure 5.19 (a). Figure 5.20 shows the step-by-step execution of the complex task in Figure 5.18 over the geometries in the example scenario in Figure 5.19(a).

```
r h PLUS r’ MINUS g INTERSECTION
```

Figure 5.18: Complex Spatial-valued task in Postfix notation



(a) Example Scenario



(b) Derived geometry after evaluating complex task in Figure 5.18

Figure 5.19: Example scenario and derived geometry after evaluating complex task in Figure 5.18

The task evaluation engine at each MBR node executes the task using a stack-based postfix expression evaluation approach. An element is popped from the expression. If it is an operand, it is pushed into the operand stack. If it is a binary operator, then two operands are popped from the stack. Otherwise, one operand is popped from the operand stack. Operators are then evaluated and the resulting operand (representing the IDG) is pushed into the operand stack. If there is no IDG operand ID on the stack, then the nodes enter the operand ID $TDGID_a$. Otherwise, it enters the IDG operand ID $TDGID_b$. Depending on the complexity of the task, an MBR node can have a maximum of two IDG operand IDs ($TDGID_a$, $TDGID_b$) on the operand stack during its evaluation.

While evaluating the expression in Figure 5.18 from left to right, operands r and h are

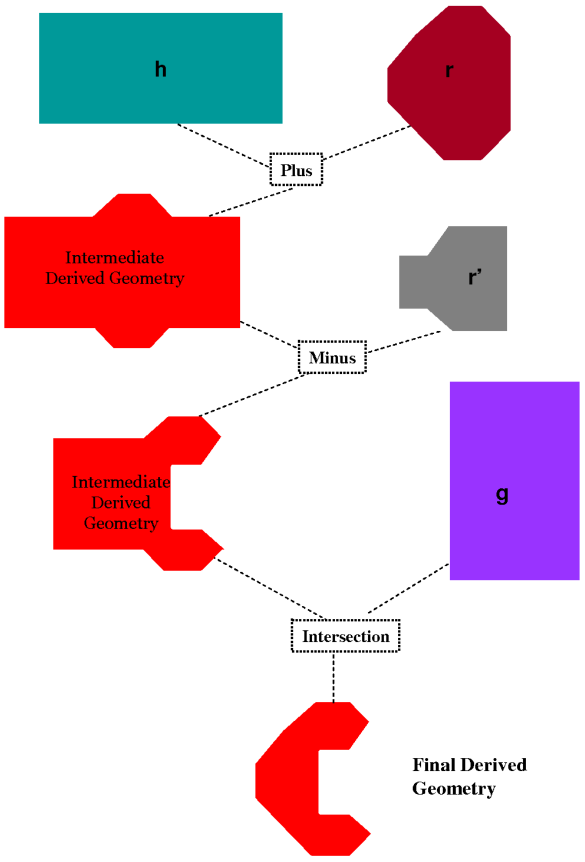


Figure 5.20: Step by step execution of complex task in Figure 5.18 over the geometries in example scenario in Figure 5.19 (a)

pushed into the operand stack. After popping the `plus` operator, operands r and h are popped from the operand stack. The MBR nodes that belong to r or h or both declare their operation state to be **true**. All other MBR nodes set their operation state to **false**.

After membership evaluation, nodes having operation state **true** engage in local communication in order to compute whether their CBNs belonging the boundary or the interior of the IDG. After this, all the nodes whose operation state is **true**, and belong to the boundary, compute the EIT information. All nodes whose operation state is **true**, and belong to the derived geometry, enter the information about the IDG in their GIT. The information includes *GID*, *datatype*, *boundary node* and *TTL*. The *GID* is set as the IDG ID (i.e., $TDGID_a$). For `plus`, type inference suffices to derive the result type, which in this case is **regions**. The node sets the *boundary node* attribute value to **true** or **false** depending on whether it lies in the boundary or the interior of the IDG. The TTL for a new derived geometry is computed as the minimum of the TTL of the operands. All MBR nodes push the operand ID for the IDG into the operand stack before evaluating the rest of the task.

While evaluating the expression from LHS to RHS, all other operations in the task will be evaluated as explained above. The next step is result processing which is now explained.

5.5.3 Result Processing

The nodes that are part of the final derived geometry add an entry for it in their GIT if it does not exists already. Otherwise, these nodes update the entry by resetting the TTL attribute in the GIT table. For the final derived geometry, the nodes set the *GID* to the derived geometry ID received as part of the task message.

5.6 Distributed Algorithms for Spatial Operations

The main research challenge faced is that of devising algorithms to operate efficiently under conditions of extreme resource scarcity and the precarious nature of the shared communication medium. The distributed algorithms presented in this chapter are, for the most part, localized, and hence have desirable complexity in terms of messages complexity as well as in terms of bit complexity, response time, and energy consumption. Then there, are challenges relating to the engineering of the solution and of the expressiveness of the algebra in terms of the complex geometries involved and of the powerful operations supported as well as the compositionality opportunities permitted by the algebra.

The section is structured as follows. Section 5.6.1 presents a model for the distributed algorithms that will be used in this chapter. Section 5.6.2 presents an overview of the task processing system. It describes how the various software components are wired together to build the overall application that runs in a node. Section 5.6.3 provides the algorithm for the stack-based evaluation of Boolean-valued tasks and the algorithms for the Boolean-valued operators. Appendix B.3 presents the algorithm for the stack-based evaluation of spatial-valued tasks and the algorithms for spatial-valued operators.

5.6.1 Model for Distributed Algorithms

This section presents a model for distributed computation of spatial operations and introduces several terms, concepts, and notations that will be used in this section. The model, terminology, concepts and notations are adapted from [San06]. Indeed, most of the structure and content of this section is adapted from [San06]. Note, that following [San06], in this section we mostly refer to nodes as entities, i.e., an element in a distributed computation. This differs from the terminology used in previous chapters.

5.6.1.1 Entity

In a WSN, each node is an entity and acts as unit of the distributed computing environment. Each *entity* $n \in N$ is equipped with local memory, processing, storage, sensing, and communication capabilities, but entities in a WSN are extremely constrained in terms of communication, power, storage and computational resources. In addition, these entities are limited by their short transmission range, and therefore, may have to perform the additional function of forwarding/relaying the data which they receive from neighbouring entities along a routing path.

Entities do not share global memory over the communication network. For the purpose of this section, the local memory of an entity n comprises a set of registers that, conceptually, can be described as follows: a geometric information register, an edge information register, a status register, and an input value register, denoted by *g_state*(n), *e_state*(n),*n_state*(n), *status*(n), *input*(n), respectively. *g_state*(n) contains GIT, *e_state*(n) contains EIT, and *n_state*(n) contains neighbours location information. *status*(n) determines the state that an entity is initially in and can change during the execution of the chore. Examples of such values includes `IDLE`, and `Available`. More details about state values in *status*(n) are provided in Section 5.6.1.3. Whereas *input*(n) contains the values of the local identifiers.

In addition, each entity has available a local timer referred to as an alarm clock. An alarm can be set, stop or reset on a timer. An alarm is fired, upon the completion of a time period.

An entity can perform the following operations: sensing, local storage, local processing, transmission of message, changing the value of the status register, and (re)setting/stoping alarm.

5.6.1.2 Event

An entity is reactive and responds to stimuli called events. An event $e \in E$ can be generated by a hardware or a software component. The completion of a request, or an external trigger, can generate an event. Events generated by hardware are interrupts caused by a timer, a sensor, or a communication device.

An event can be internal or external. We envisage internal events E_i as representations of stimuli that are local to an entity and originate within the entity (e.g., an interrupt indicating the firing of a timer). External events E_o are external to the system (e.g., an incoming radio packet). The set of all events is $E = E_i \cup E_o$.

In the scope of this dissertation, an entity can call a method *Set_Alarm(a_ID, time)* to set an alarm, where *a_ID* denotes the unique alarm ID and *time* denotes the period of time the expiry of which causes alarm *a_ID* to fire. In addition, an entity can turn off and reset a specific alarm by calling methods *Stop_Alarm(a_ID)* and *Reset_Alarm(a_ID)*, respectively. Each timer is associated with a $When$ event. Upon the completion of the time *time*, any action associated with an $Alarm\_Fire(a\_ID)$ event is executed. More details about actions are provided below.

A *Receive* event is associated with incoming radio packets at the receiving entity. An entity can receive more than one message. The sending entity associates with each message to a message ID, specifying which application or process it should be forwarded. Upon the reception of a message, an entity is responsible for forwarding each message to the specific process/component running on it, based on the message ID. *Receive* events are differentiated from each other by their message ID denoted by *Receive*(m_ID).

A *Read* event is associated with an interrupt indicating that new sensor reading has been made available. An entity may be equipped with more than one sensing device. *Read* events are distinguished by the corresponding sensing device ID, denoted by *Read*(s_ID).

A *Spontaneously* event is not associated with an interrupt or external cause such as hardware interrupt. Among the events that are external or internal to the system, *Spontaneously* event has the higher precedence.

The operation **Send** generates a *Receive* event, **Sense** generates a *Read* event and **Set_alarm** and **Reset_alarm** generate a $When$ event.

When an event $e \in E$ occurs, the action block associated with the event is executed. An action block comprises a finite sequence of instructions, called actions. It is possible, that in response to an event, an entity n does not react because no action is associated with event.

5.6.1.3 Status

In the scope of this dissertation, the set of state values $\boldsymbol{S}$ comprises three subsets, viz., S_{INIT}, $S_{INTERMEDIATE}$, and S_{TERM}. S_{INIT} denotes the set of values an entity can hold at the start of the execution; $S_{INTERMEDIATE}$, the set of values that an entity can hold after the start and before the end of the execution; and S_{FINAL}, the set of values that an entity can hold after the execution of an action block has ended.

More formally, the set of state values $\boldsymbol{S}$ can be described as, $\boldsymbol{S} = \boldsymbol{S}_{INIT} \cup \boldsymbol{S}_{INTERMEDIATE} \cup \boldsymbol{S}_{TERM}$

Among the S_{INIT} values, $S_{START} \subset S_{INIT}$ refers to the set of values, which if an entity is in, is responsible for starting the evaluation of the protocol. Typically, it contains a single value $S_{START} = \{\texttt{INITIATOR}\}$. In the scope of this dissertation, if action block evaluation is started on all entities in a specific state, then only the set of states in S_{START} are specified.

S_{TERM} values denotes states that cannot ever be changed by the protocol. Among S_{TERM} state values, there exists a subset of values, called S_{FINAL}, that define the status in which no further activity takes place.

5.6.1.4 Behaviour

The action that an entity $n \in N$ will take depends upon the type of event and the value in $status(n)$ when an event occurs. This can be formally defined as a rule of the form:

$status(n) \times Event \Longrightarrow Action$,

The behaviour of an entity n can be defined as finite set of rules $B(n)$ that an entity n obeys. In other words, actions determine the behaviour of an entity based on specific conditions. For every distinct event and status, there exists one rule that defines the action which an entity should take when that event occurs in that state. A set of rules $B(n)$ is called a protocol or a distributed algorithm for n.

In this dissertation, we make use of the following conventions:

- Rules are grouped by status.
- If the action for a (status, event) pair is *Null*, then no rule is given for it.
- If the action for a (status, event) pair includes a command to change the state, and is followed by a call to the protocol, it executes the protocol in that new state. Otherwise, it simply changes the state and executes the events when they occur in that state.
- If a protocol calls another protocol without passing any arguments, it is simply returning the control back to that protocol.
- The action associated with an *Spontaneously* event is executed implicitly.

- In distributed computations, it is possible that at some specific point in time some entities are still running the protocol while others may have terminated its execution before completion of the overall computation; and also some entities based on the *status*(n) or the *g_state*(n) register may be ready to participate in the execution of protocol while others may not.

5.6.1.5 Communication

A WSN can be considered a bidirectional graph $G(V, E)$, with the entities as the set of vertices V, and the wireless communication links as the set of edges E. The entities that an entity can receive from and transmit to directly are its one-hop neighbours. Consider two entities s_j and $s_k \in V$, if s_j is within the propagation area of s_k, there is an edge (s_j, s_k) $\in E$.

In a WSN, messages can be lost as well as corrupted. Packets can be lost due to communication links going down or to collisions. Packet collision occurs when an entity receiving a packet hears one or more additional transmissions over the same channel [KM07, MV05]. The result of a collision is generally a garbled message. In general, packet collision occurs when two or more entities send data at the same time over the same channel. When a collision is detected, the message is rejected and the result is that no message is delivered.

WSNs require some sort of mechanism either to prevent collisions altogether or to recover from collisions when they occur. In WSNs, the role of the *medium access control* (MAC) protocol is to coordinate access to and transmission over a shared wireless medium [WC01]. In the networking literature, numerous MAC protocols have been proposed, which are usually grouped in two different broad categories: contention-based and reservation-based [BDWL10, KM07, YH03].

The communication system may deliver duplicated messages. In addition, communication delays may occur due to message failure, network congestion and processing overhead. By *communication delay* we mean any additional overhead above the normal expected time that elapses between the event at the sending entity which transmitted the message and the event that processes the message at the receiving entity.

5.6.1.6 Messages

For the distributed execution of an application logic, entities communicate with each other by passing messages over the communication channels.

In this dissertation, the following conventions will be used. The message comprises a finite number of bytes and its size is bounded by the underlying system. A message is usually transmitted by an entity as a pair (*m_ID*, *tuple*), where *m_ID* denotes the message ID and *tuple* denotes a sequence of length k of data fields (*f1*, *f2* .... *fk*), where each field contains some amount of payload information. An entity can transmit the message directly to its neighbours by specifying their ID as destination. For transmitting a message, the *Send* operation is used. The syntax of the *Send* operation is as follows:

Send (**m_ID**, f_1, f_2 .... f_k) **to** *destination*

In the case of a message broadcast, $\mathbf{N}(n)$ is specified as *destination*, denoting the local neighbourhood of an entity n. In the case of transmission to a specific neighbour, the entity ID of that neighbour is specified as the *destination*.

5.6.1.7 Protocol

The content of all the registers of an entity n and the values of its alarms constitute the internal state of an entity. Some of the registers are initialized before the start of the protocol, while others are initialized during the execution.

In this section, the initial and final conditions of each protocol are defined in terms of $\{\mathrm{P}_{INIT}, \mathrm{P}_{FINAL}\}$ are provided in the algorithm pseudocode for a protocol. P_{INIT} and P_{FINAL} are predicates on the internal state of the entities, and R defines set of restrictions. Let t_0 denote the time before the entities start evaluating the protocol and t_f the time at which the evaluation of protocol ends. P_{INIT} at t_0 defines the conditions the entities are in before the start of the protocol, P_{FINAL} at time t_f defines the conditions the entities are in after evaluating the protocol. The set of R restrictions for all the protocols described in this section are the same, as now described.

5.6.1.8 Restrictions

The distributed algorithms defined in this section, make several assumptions. These assumptions define the knowledge and restrictions under which these algorithms operate. Some restrictions regarding the network are as follows.

1. Bidirectional communication links
2. An entity may receive duplicate messages. Some of the possible causes include loss of acknowledgement messages (during routing and tree construction), in response to which the transmitter retransmits the message after a fixed interval; broadcast communication (e.g., during task dissemination inside MBR), as a result of which an entity receives the same message from different sources.
3. An entity may not receive a message due to collision. One cause of packet collision is the hidden terminal problem [TL08].
4. Physical time plays a crucial role in many WSN applications. Time is crucial not only for data fusion but also for intra-network coordination among different sensor entities. Time synchronization is a well studied research problem [SKPM06, EE01], and in the scope of this dissertation, it is assumed that all entities are time synchronized.
5. Delay in communication may occur, e.g., due to different workloads. For example, if an entity is busy with aggregation and receives a message to be sent towards the first-level leader, then the relaying of the message may be delayed. This situation can particularly affect tasks involving MEG operands because of their size and complexity. In such cases, the aggregation process may complete at different times in different operands. Another cause of communication delay is a transmission failure, in which case the sender waits for an acknowledgement and retransmits if none arrives. Finally, network congestion may cause communication delay as well.

5.6.2 Description of the Task Processing System

A simple component diagram of the task processing system that runs on entities is given in Figure 5.21. It describes the high-level software components and the interfaces to those components. Components may both provide and require interfaces. An interface defines a cohesive set of behaviours. A connector from one component to another denotes that the former provides the interface that the latter requires. The component that provides the interface exhibits a lollipop (the symbol with a complete circle at its head), whereas the component that requires the interface exhibits a socket (the half-circle symbol). To avoid clutter, we have not included interface titles in Figure 5.21. The inner structures of the components *Boolean-Valued Operator (Type1)* and *Boolean-Valued Operator (Type2)* are shown in Figure 5.22. The inner structures of the components *Spatial-Valued Operator (Type1)* and *Spatial-Valued Operator (Type2)* are shown in Figure 5.23.

The *radio*, *receive* and *transmit* components constitute the communication layer. The *transmit* component is responsible for tagging the message with an ID and transmitting it over the radio. The *receive* component is responsible for the reception of messages destined to an entity and forwarding it to a specific component on the basis of the message ID. The *radio* component is responsible for turning the radio on and off and starting the radio either in *full* mode or in *low power listening* mode. In *low power listening* mode, the entity cycles between the *sleep* state and the *awake* state [MM06, LWG05].

The architecture of the task processing system consists of two main components, viz., the software running on the motes and the software that runs on the gateway. A WSN is associated with a gateway that acts as the source for disseminating the spatial analysis tasks and as the sink for receiving the corresponding outcomes. Spatial tasks are translated in the gateway into a compact byte representation in order to eliminate the need for a sophisticated parser on each mote and to reduce the size of the task message. The contents of this data structure are defined in Appendix B.1.1. Once a task has been disseminated, it is handed over to the task processing system, which is responsible for the evaluation of the task conveyed in the task message. Each entity that belongs to the task MBR evaluates the task specified in the task message. These entities also cooperate in the execution of the task. Task processing consists of two phases, viz., preprocessing; and evaluation. Preprocessing is performed once, after the reception of the task message. Evaluation is performed repeatedly, once per evaluation period. Preprocessing is performed by the *evaluate* component, which is responsible for checking the validity of the task and forwarding it to a specific evaluator on the basis of the type of task. For example, if the task comprises Boolean operators, then it is forwarded to the *Evaluate Boolean-Valued Task* component for evaluation. Due to the limited capabilities of an entity, most of the validation is performed on the gateway.

The evaluation logic of a task depends upon the type of task. The *Evaluate Boolean-Valued Task* component in each entity is responsible for the stack-based evaluation of the Boolean-valued tasks. The *Neighbour GIT-lookup* component is used for getting geometric information from the neighbouring entities.

The *Evaluate Spatial-Valued Task* component in each entity is responsible for the stack-based evaluation of spatial-valued tasks. The *Evaluate Induce Geometry* component in each

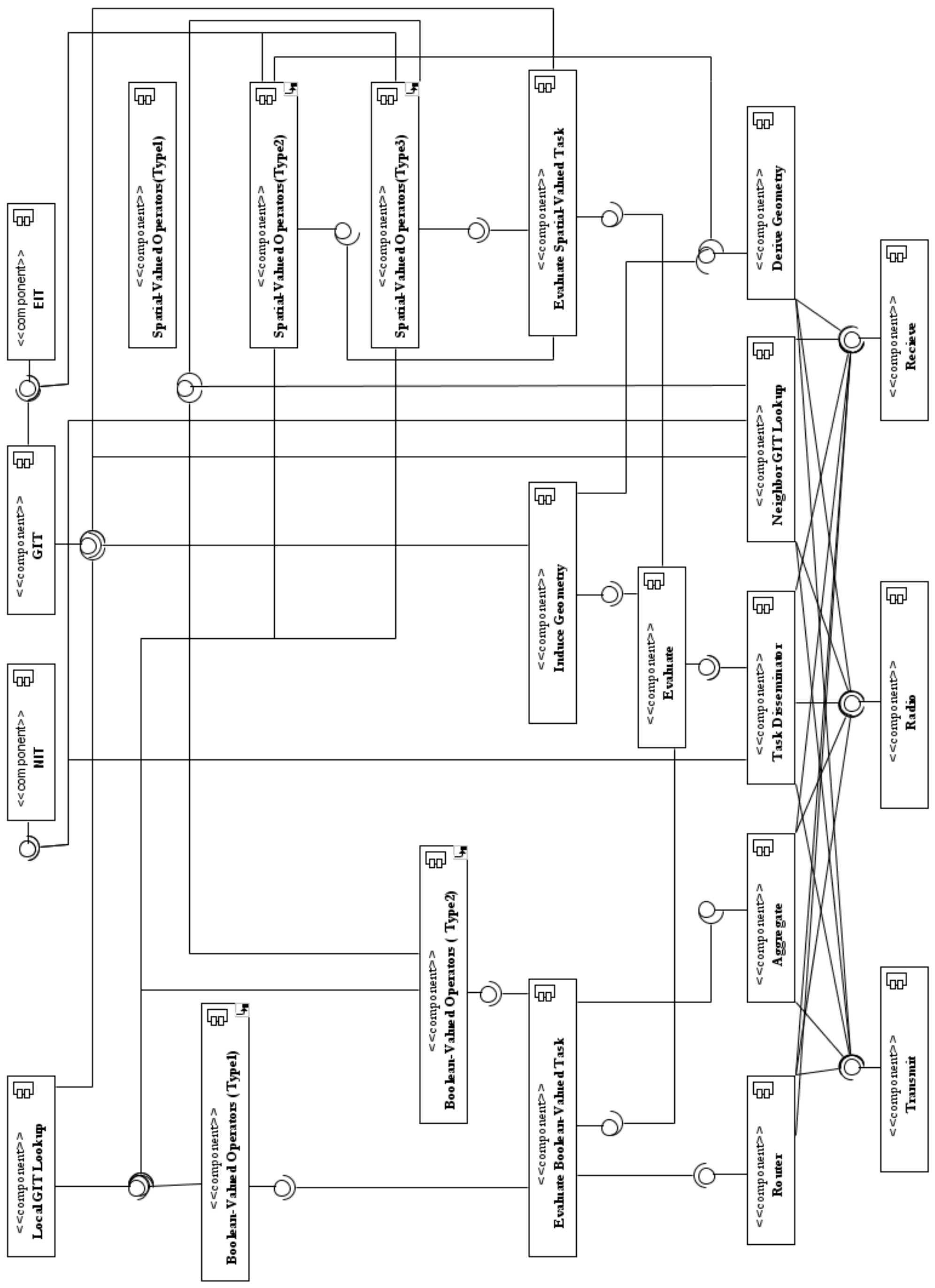

<<component>> EIT
<<component>> GIT
<<component>> NIT
<<component>> Local GIT Lookup
<<component>> Spatial-Valued Operators(Type1)
<<component>> Spatial-Valued Operators(Type2)
<<component>> Spatial-Valued Operators(Type3)
<<component>> Evaluate Spatial-Valued Task
<<component>> Induce Geometry
<<component>> Evaluate
<<component>> Boolean-Valued Operators (Type2)
<<component>> Boolean-Valued Operators (Type1)
<<component>> Evaluate Boolean-Valued Task
<<component>> Derive Geometry
<<component>> Neighbor GIT Lookup
<<component>> Task Disseminator
<<component>> Aggregate
<<component>> Router
<<component>> Recieve
<<component>> Radio
<<component>> Transmit

Figure 5.21: Task Processing System component diagram

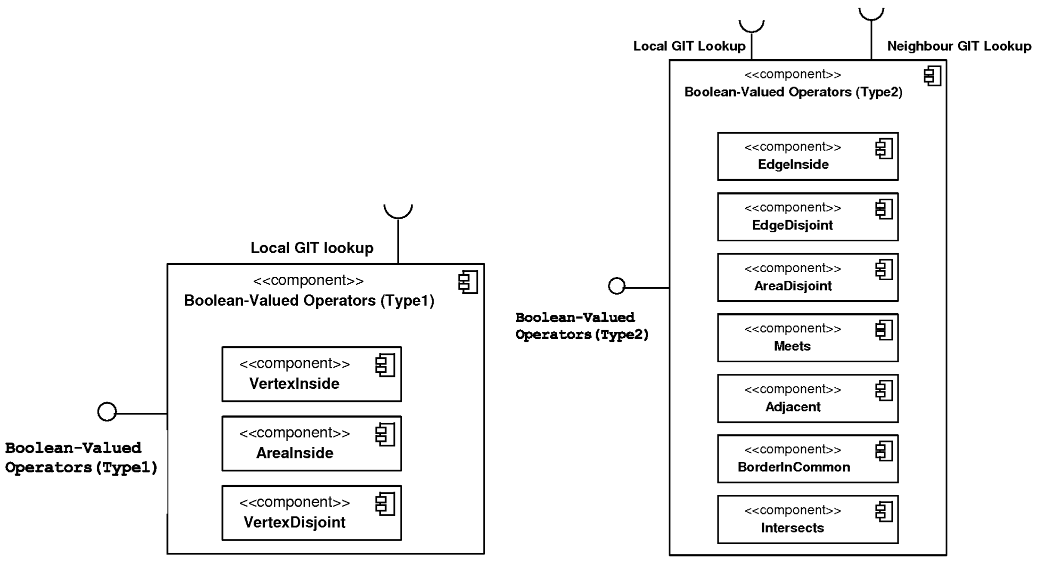
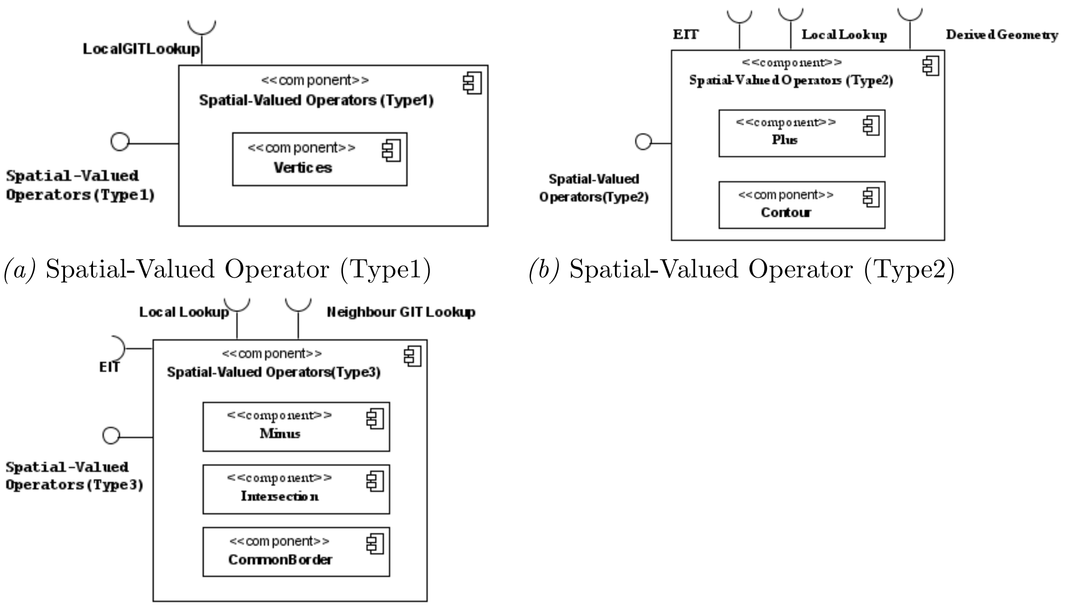


Figure 5.22: Task Processing System components *(a)* Boolean-Valued Operator (Type1) *(b)* Boolean-Valued Operator (Type2) inner structure is composed of other components



(a) Spatial-Valued Operator (Type1) *(b)* Spatial-Valued Operator (Type2)

(c) Spatial-Valued Operator (Type3)

Figure 5.23: Task Processing System component's *(a)* Spatial-Valued Operator (Type1) *(b)* Spatial-Valued Operator (Type2) *(c)* Spatial-Valued Operator (Type3) inner structure is composed of other components

entity is responsible for the derivation of induced geometries. The *Derive Geometry* component is responsible for the derivation of geometries.

The *GIT* component is responsible for the insertion, look-up, modification, and deletion in the GIT of tuples with information about the geometries an entity belongs to. This component is equipped with an alarm that fires periodically. At each such event, the validity time of tuple is decreased by one time unit.

Recall, from Section 5.4.1, that during task dissemination a task message is forwarded towards the task MBR using a greedy forwarding method that requires each entity to be aware of its neighbours location. Let the neighbours location be stored in a *neighbour information table* (NIT).

The *EIT* component is responsible for the insertion, look-up, modification, and deletion, of information about segments formed with neighbours that do not belong to the geometries the entity belongs to.

5.6.3 Concrete Distributed Algorithms for Spatial Operations

As discussed in Section 5.6.2, the *task disseminate* and *evaluate* components participate in the evaluation of any type of task. Appendix B discuss the protocols for both the *task disseminate* and *evaluate* components.

Recall, from Section 5.4, that the Boolean-valued task evaluation process comprises two phases, viz., *distributed task evaluation*, and *result processing*. The description of the distributed task evaluation protocol is provided later in this section, and description of the result processing protocol is provided in Appendix B.

5.6.4 Distributed Task Evaluation

This section, discuss the *EvaluateBooleanValuedTask* protocol in Figure 5.24. It uses a stack-based evaluation approach. The protocol supports re-evaluation of tasks, up to a fixed number of times. This protocol is only responsible for the evaluation of Boolean-valued operators not Boolean-valued connectives. Boolean-valued connectives are applied by the first-level leader on the finally aggregated operation state (line 100 in Figure B.25 in Appendix B).

Each operation in the task is evaluated. In order to compress the operation states into an entity-level task state, the procedure *ComputeTaskState* is invoked. The *SatisfyTaskOP* procedure is responsible for checking whether an entity satisfies the condition for participation, at least one operation in the task. If an entity does so, the *ComputeAggregationOperand* procedure is invoked to select the aggregation operand, as discussed in Section 5.4.2.5. After the selection of the aggregation operand, the *Aggregation* protocol in Figure B.23 (Appendix B) is invoked. After the aggregation process is performed, the *Result Processing* protocol (explained in Appendix B) routes the messages from each SEG leader to the first-level leader and then from the first-level leader to gateway.

The description of the `vertex_inside` protocol is provided in this section, and that of other protocols are provided in Appendix B.

VertexDisjoint

For the computation of this operation, the information required by an entity is available in its own GIT. The procedures used in the `vertex_disjoint` protocol (Figure 5.25) include *CheckVDDataTypeValidity* and *LocalGITLookup_Search*. *CheckVDDataTypeValidity* is responsible for computing whether the operands are valid i.e., it returns **true** if an entity satisfies the operand's data type, which it is a member of. *LocalGITLookup_Search* is responsible for local GIT-lookup and for returning the address of the tuple in GIT, if it exists, otherwise it returns **null**.

5.7 Summary

In this chapter, the algorithmic strategy for in-network processing of spatial task has been presented. As mentioned, there exists related work in the area of topological change detection in WSNs and extracting information that represent topological features. However, there is no related work which provides a general framework or a distributed spatial algebra for supporting complex spatial algebraic expressions.

This chapter described the various types of complex tasks supported by the task processing system. It also explained the attributes of the supporting structures. In addition, the main contribution of this chapter has been the construction of a generic technique for evaluating each spatial operator in a distributed manner on WSNs. The algorithmic strategy for evaluating Boolean-valued and spatial-valued tasks over multi-element geometries has also been described.

The approach presented is comprehensive as it covers a broad range of task evaluation aspects. The empirical evaluation in Chapter 6 will show the approach used for the distributed evaluation of spatial tasks scale well in number of nodes involved. In addition, it also scales well on task complexity measured as the number of operators comprising the task, and the number of multi-element operands, with and without holes increases. Consequently, the approach contributed in this chapter is appropriate for distributed environments with large numbers of mote-level nodes.

This chapter has also presented the architecture of a task processing system as well as distributed algorithms for Boolean-valued operations for in-network spatial analysis. The contributed algorithms are novel and have been designed with specific consideration to the distributed nature of the execution platform and in full awareness of the extreme resource scarcity in WSN platforms.

```
Protocol EvaluateBooleanValuedTask(postfixTask, reEvalPeriod, duration)
// $P_{INIT}$ ≡ "All entities in the MBR can evaluate Boolean-valued task at time $t_0$" ≡
//             {∀ n ∈ N: n ∈ MBR ⇒ n can evaluate the task at time $t_0$}
// $P_{FINAL}$ ≡ "Final result is received at the sink at time $t_f$" ≡
//             {∃ n ∈ N: n has task result message at $t_f$ ⇒ n is the sink }

Status Values: S= {TASK_EVALUATING, SEGLEADER, IDLE, OPERATION_EVALUATING, AVAILABLE}
$S_{START}$= {TASK_EVALUATING}
$S_{INTERMEDIATE}$={IDLE, SEGLEADER, OPERATION_EVALUATING}
$S_{TERM}$ = {AVAILABLE}
TASK_PREPROCESSING
    Spontaneously
        countOP = 0
        Stack s
        taskstate = 0
        taskstateSEG = 0
        leaderSEG= false
        operationState = 0
        isAggregationDone = false
        PushTaskOnStack(postfixTask, s)
        Set_Alarm(a_t20, reEvalPeriod)
        Become(TASK_EVALUATING)
TASK_EVALUATING
    Spontaneously
        while(not s_empty())
            leftArg = s_pop()
            rightArg = s_pop()
            op = s_pop()

            if (op = vertex_inside):
                // VertexInside protocol return the state of the operation in stateOP
                Become(OPERATION_EVALUATING)
                VertexInsideOP(leftArg, rightArg, &operationState)
            elseif (op =area_inside):
                Become(OPERATION_EVALUATING)
                AreaInsideOP(leftArg, rightArg, &operationState)
            elseif (op = edge_inside):
                Become(OPERATION_EVALUATING)
                EdgeInsideOP(leftArg, rightArg, &operationState)
            elseif (op = area_disjoint):
                Become(OPERATION_EVALUATING)
                AreaDisjointOP(leftArg, rightArg, &operationState)
            elseif (op = vertex_disjoint):
                Become(OPERATION_EVALUATING)
                VertexDisjointOP(leftArg, rightArg, &operationState)
            elseif (op = edge_disjoint):
                Become(OPERATION_EVALUATING)
                EdgeDisjointOP(leftArg, rightArg, &operationState)
            elseif (op = meets):
                Become(OPERATION_EVALUATING)
                MeetsOP(leftArg, rightArg, &operationState)
            elseif (op = adjacent):
                Become(OPERATION_EVALUATING)
                AdjacentOP(leftArg, rightArg, &operationState)
            elseif (op = equals):
                Become(OPERATION_EVALUATING)
                EqualsOP(leftArg, rightArg, &operationState)
            elseif (op = not_equals):
                Become(OPERATION_EVALUATING)
                NotEqualsOP(leftArg, rightArg, &operationState)
            elseif (op = intersects):
                Become(OPERATION_EVALUATING)
                IntersectsOP(leftArg, rightArg, &operationState)
            elseif (op = on_border_of):
                Become(OPERATION_EVALUATING)
                OnBorderOfOP(leftArg, rightArg, &operationState)
            elseif (op = border_in_common):
                Become(OPERATION_EVALUATING)
                BorderInCommonOP(leftArg, rightArg, &operationState)

            if (operationState = NOT_PART_OF_OPERANDS) or
            (operationState = OPNOTSUPPORTED)):
                operationState = OP_NOT_APPLICABLE

            taskState = ComputeTaskState(taskState,operationState)

```

Figure 5.24: Protocol *EvaluateBooleanValuedTask*

```
    if (s_empty()):
        if (isAggregationDone = false):
            satisfyOP = SatisfyTaskOP(taskState)
            isAggregationDone = true

            if (satisfyOP = true):
                gmtryID = ComputeAggregationOperand(postfixTask)
                // After Aggregation each entity gets partial aggregated task state
                // and information whether it is selected as SEG leader
                taskStateSEG = taskState
                leaderSEG = n.ID
                Become(IDLE)
                Aggregation(gmtryID, &taskStateSEG, &LeaderSEG)
            else
                Become(AVAILABLE)
        else :
            if (leaderSEG= n.ID):
                Become(SEGLEADER)
                ResultProcessing(taskStateSEG, gmtryID)
            else :
                Become(AVAILABLE)
                ResultProcessing(taskStateSEG, gmtryID)

AVAILABLE
When(a_t20)
    evalPeriod = ComputeReEvalPeriod(evalPeriod, reEvalPeriod)
    if (evalPeriod < duration)
        Become(TASK_PREPROCESSING)
    else :
        Become(AVAILABLE)

```

Figure 5.24 (continued)

```
Protocol VertexDisjoint(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the vertex_disjoint operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true or false at time $t_f\}$

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        stateOP = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckVDDataTypeValidity(tupleG1, tupleG2)

            if (isOperandsDTValid):
                if ((tupleG1 != null) and (tupleG2 != null)):
                    stateOP = false
                elseif ((tupleG1 != null) or (tupleG2 != null)):
                    stateOP = true
            else :
                stateOP = OP_NOT_SUPPORTED

        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure 5.25: Protocol *VertexDisjoint*

Chapter 6

Experimental Validation

In Chapter 5, an algorithmic strategy for computing Boolean-valued and spatial-valued operations, and, building on that strategy, algorithms for the distributed evaluation of each such operation, have been described. These algorithms have been implemented over the de facto standard simulation environment for WSNs. This chapter describes a series of experiments that aim to evaluate the degree to which the distributed algorithms described in Chapter 6 meets the desiderata placed upon. The aim of the experiments is to investigate the benefits of in-network processing of spatial-analytical tasks.

We performed experiments using simulations of an actual distributed platform, viz., TinyOS [LMG^{+}04] and studied four main performance indicators: bytes transmitted, messages transmitted, energy consumption and response time.

The structure of this chapter is organized into six sections as follows: Section 6.1 describes the implementation environment. Section 6.2 explains the experimental context used for all the experiments. Section 6.3 summarizes the preprocessing steps required to run the experiments over the simulator. Section 6.4 reports on an experimental evaluation of spatial analysis out-of-network. Section 6.5 reports on an experimental evaluation of geometry induction. Section 6.6 presents an experimental evaluation of in-network spatial analysis. Finally, Section 6.7 concludes the chapter.

6.1 Simulation Environment

TinyOS [LMG^{+}04] is an open-source operating system designed for mote-level WSNs and nesC is the special purpose language designed to embody the structural concepts and execution model of TinyOS [LG09]. All of the algorithms contributed in Chapter 5.6 were implemented in nesC/TinyOS 1.x. The algorithms in this dissertation were evaluated using detailed simulations over the TOSSIM [LLWC03] TinyOS simulator. TOSSIM is an event-driven simulator, for application execution in a controlled, reproducible environment. It simulate, nesC/TinyOS code and can scale to large number of nodes. TOSSIM provides a number of features including run-time configurable debugging output and radio modeling but it does not allow the estimation of energy consumption. Energy consumption is an important factor in determining the life time of a WSN. Therefore, to estimate the total energy consumed by the sensor nodes, the PowerTOSSIM [SHrC+04] extension to TOSSIM is used to provide a per-node estimation of energy consumption that scales to larger networks.

TOSSIM does not allow for modeling real-world behaviours such as a specific network layout, mobility of sensor nodes or the assignment of specific readings to sensor nodes. It provides a socket-based API for other applications to allow such modeling. One such application that can be used for the purpose of complex real-world modeling is TinyViz [Tin], a graphical user interface that communicates with TOSSIM over the socket API and allows to control and analyze the simulation. TinyViz provides several mechanisms for interacting with the network, e.g., packet traffic monitoring and a set of plugins that provide basic debugging and analysis capabilities (examples include a plugin that displays all debug messages in list format, a plugin that changes radio connectivity based on distances between nodes, and another that displays the data in radio packets). In addition, it also supports a plugin interface allowing developers to implement their own application-specific code within the TinyViz engine, and thereby to extend the GUI's functionality.

A scripting language, Tython (also called, Tinython) [DLJ+05], provides a scripting interface to TOSSIM. Tython is based on Jython, a Java implementation of the Python language. Users can use TinyViz, a Tython console, or both simultaneously, to interact with a running simulation.

TOSSIM MAC is a contention-based MAC protocol, providing transmission and reception of information using CSMA/CA-based contention-avoidance schemes [BDWL10, KM07, YH03].

6.2 Experimental Context

The tasks used for experiments in this report are based on the motivating example in Section 1.4. As described in Section 1.4, mildew and other bacteria develop under closely-related conditions that can compromise the desired productivity, so monitoring the moisture and temperature in the soil is crucial to improve crop yield and quality. If needed, chemicals can be applied under those conditions. As a result, it is important to monitor the temperature and soil moisture with temporal and spatial precision in order to decide when and where to apply the chemicals. The vine crop is influenced by the physical conditions where it is growing (e.g., location, topography, soil type etc.), and the climatic environment. Grape growers can take advantage of natural factors, by selecting suitable sites for particular grape varieties. Wine makers can achieve greater control over the product by selecting, fermenting and blending batches of grapes with suitable ripeness and flavors. The tasks for the experiments are generated based upon the detailed study of the requirements of the sensor network deployments for precision agriculture [Ulr08, PE08, A05, MCP+02, MGZ+09, KR04].

6.3 Preprocessing Steps

For the experiments in this chapter, different graphics packages have been used to draw and subsequently transform the drawing of geometric shapes into textual representation of geometric shapes corresponding to hypothetical fields in the example application as well as induced and derived geometries. Scripts were written in Python, that make use of these data files to generate files containing location and GIT information for the nodes.

TinyViz was used for modeling network behaviour. Several plugins have been written to make each sensor node aware of specific information including location, one-hop neighbourhood, and initial GIT tuples.

For further automating the whole system, scripts were written in Tython, that automatically invoke the required plugins in the TinyViz engine and run the task processor code. Once the initial loading of static information is completed, and each node is equipped with basic information about location, and the initial GIT tuples, initial EIT tuples, the nodes are ready to evaluate tasks that are complex expressions in the algebra described in this dissertation.

6.4 Out-of-Network Spatial Analysis Approach

At the time of writing, there is no comparably expressive, implemented platform for in-network spatial analysis that we might compare our implementation with. We therefore compared our approach with an out-of-network approach in which all data is sent back to the gateway which provides a baseline comparable to a warehousing approach to sensor data, and a less expensive an out-of-network approach in which only boundary information of induced regions is sent back to the gateway.

6.4.1 Experimental Setup

The specification of the sensor nodes we have simulated is [Type = MICA2, Radio = CC1000, Energy Stock = 31,320,000 mJ (2 Lithium AA batteries)]. The radio range is set such that the nodes can form a one-hop neighbourhood. The radio connectivity between nodes is based on distances between nodes and, therefore, the maximum cardinality of the one-hop neighbourhood of a node is set to eight in our experiments unless specified otherwise. In all experiments, it is assumed that the tasks related to detection of induced geometries run over the whole network and the cost associated with the induction of induced geometries is not included in the experimental cost. In all experiments presented in this section, it is assumed that each node knows its parent, to which it has to transmit message, and, to receive messages from, its children. A node tries to send a packet to its parent up to four times, upon not receiving the acknowledgement from parent. Otherwise, it broadcasts the message to its neighbours. A node also checks not to send duplicate packets.

Size of the networks for the experiments are selected by not only considering the number of nodes in the real-world deployments [Ulr08, PE08, A05, MCP+02, MGZ+09, KR04, KLS+10], but also to demonstrate that the approach contributed in this dissertation is appropriate for distributed environments with large numbers of mote-level nodes. The number of nodes deployed in some of these applications are as follows: Camalie vineyards [Ulr08] 25 nodes, NAV (Network Avanzato per il Vigneto) system [MGZ+09] 45 nodes for precision viticulture, LofarAgro project [A05] around 150 nodes to fight fungal-disease in the field.

6.4.2 Experiments

6.4.2.1 Experiment O1: Every sensor node senses and every reading is sent back to the gateway

The purpose of the experiment is to measure the cost in terms of messages transmitted, energy consumed and response time of sending all the measurement from every node back to the gateway, so that the gateway can generate a snapshot of the induced geometry based on the received data and allow the evaluation of spatial-analytic tasks. In this case, all data is centrally held in the gateway and centralized algorithms could be used. The experiment is run over the scenario in Figure 6.1. Here, and elsewhere in this chapter, we plot the energy consumed by the CPU and by the radio components separately. The results are depicted in Figure 6.2.

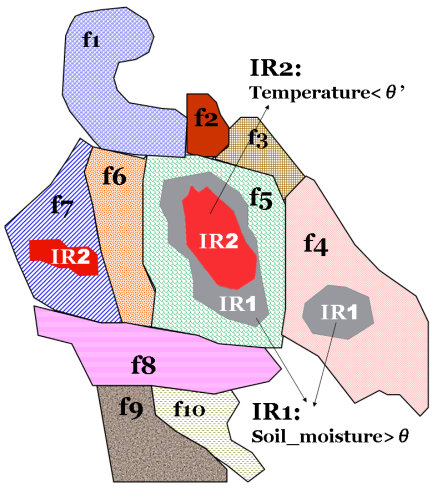
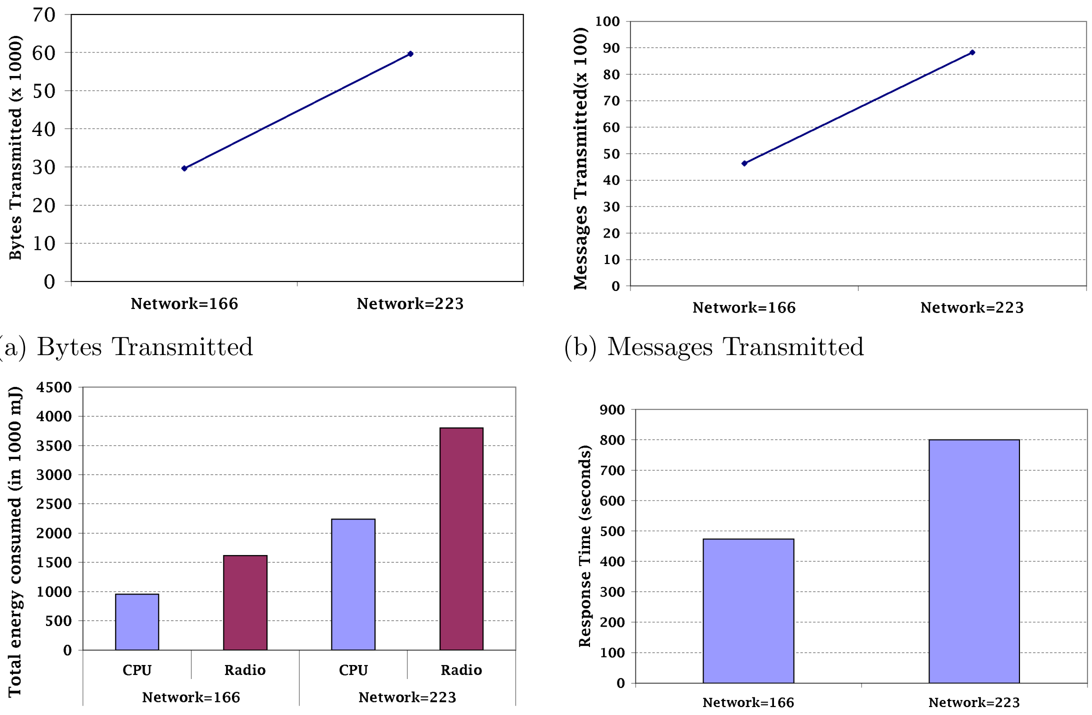


Figure 6.1: Scenario (Snapshot of vineyard)

In this experiment we have only plotted the result for the two of the network sizes, because, in this approach, apart from network longevity, scalability is an issue as it result in increase in bandwidth requirements, increasing risks of packet loss due to collisions, the need to increase buffer sizes and the effect that nodes that are closer to sink run out of energy stock sooner than other nodes. For energy efficiency, techniques for congestion control [WEC03], flow control [CHS+09] and scheduling algorithms [PBM+05] are needed. To avoid collision at the physical layer, transmission events must be made to occur less frequently. For energy efficiency, transmission events must be made to occur more frequently in order to maximize the radio power-off interval and reduce the radio listening interval for the incoming packets.

In this experiment, for a network size of 166 nodes, the amount of energy consumed by CPU and radio is close to 2,573 kmJ. If each node is powered by two AA batteries, the initial energy stock per node is 31,320,000 mJ. The total energy stock inside the network is then the network size times 31,320,000. This implies that each evaluation episode consumes between 0.05% and 0.09% of the total energy stock for networks containing between 166 and 223 nodes, respectively. However, note that the distribution of energy consumption is not uniform: nodes that are closer to the sink will deplete their energy stock much faster.



(a) Bytes Transmitted (b) Messages Transmitted

(c) Energy Consumed (d) Response Time

Figure 6.2: Experiment O1: Behaviour w.r.t. Network Size (Sensed measurements are sent back to the gateway)

6.4.2.2 Experiment O2: Boundary information of induced regions are sent back to the gateway

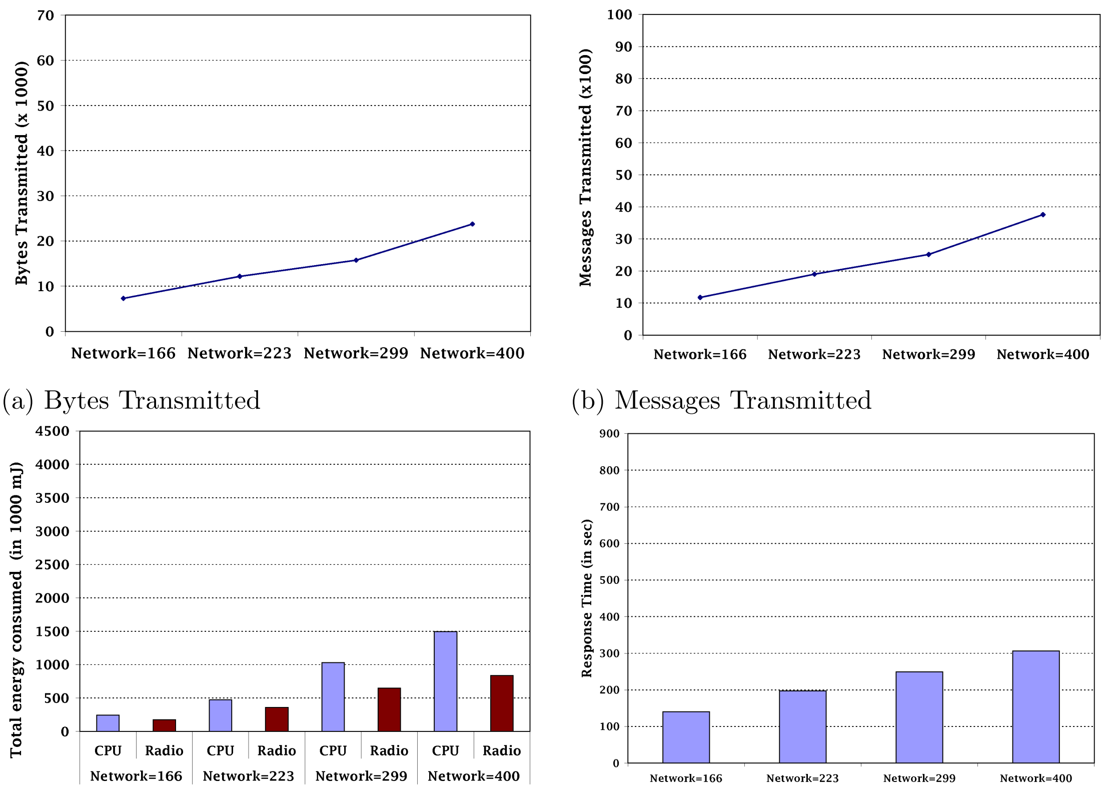
The purpose of this experiment is to measure the cost in terms of messages transmitted, energy consumed and response time of sending only the event information from boundary nodes (in contrast to Experiment O1 in which all measurements from all nodes were sent) back to the gateway for performing spatial analysis outside the network, so that the gateway can construct the snapshot of induced geometry to allow the processing of spatial-analytic tasks outside the network. The experiment is run over the scenario in Figure 6.1. The results obtained are depicted in Figure 6.3. The size of induced geometries are given in Table 6.1.

Network Size	IR1 in f5	IR2 in f5	IR1 in f4	IR2 in f7
166	28	11	12	6
223	37	15	18	10
299	51	20	22	14
400	68	28	27	18

Table 6.1: Size in number of nodes of the Induced Geometries in Experiment O2

In this experiment, for the network containing 166 nodes the amount of energy consumed by CPU and radio together is close to 4,22 kmJ. This implies that each evaluation episode consumes between 0.008% and 0.04% of the total energy stock for networks containing between

166 and 400 nodes, respectively. However, note that the distribution of energy consumption is not uniform: nodes that are closer to the sink will deplete their energy stock much faster. In comparison with Experiment 1, the growth rate is slower and the magnitudes much smaller.



(a) Bytes Transmitted (b) Messages Transmitted

(c) Energy Consumed (d) Response Time

Figure 6.3: Experiment O2: Behaviour w.r.t. Network Size (Boundary information of induced regions is sent back to the gateway)

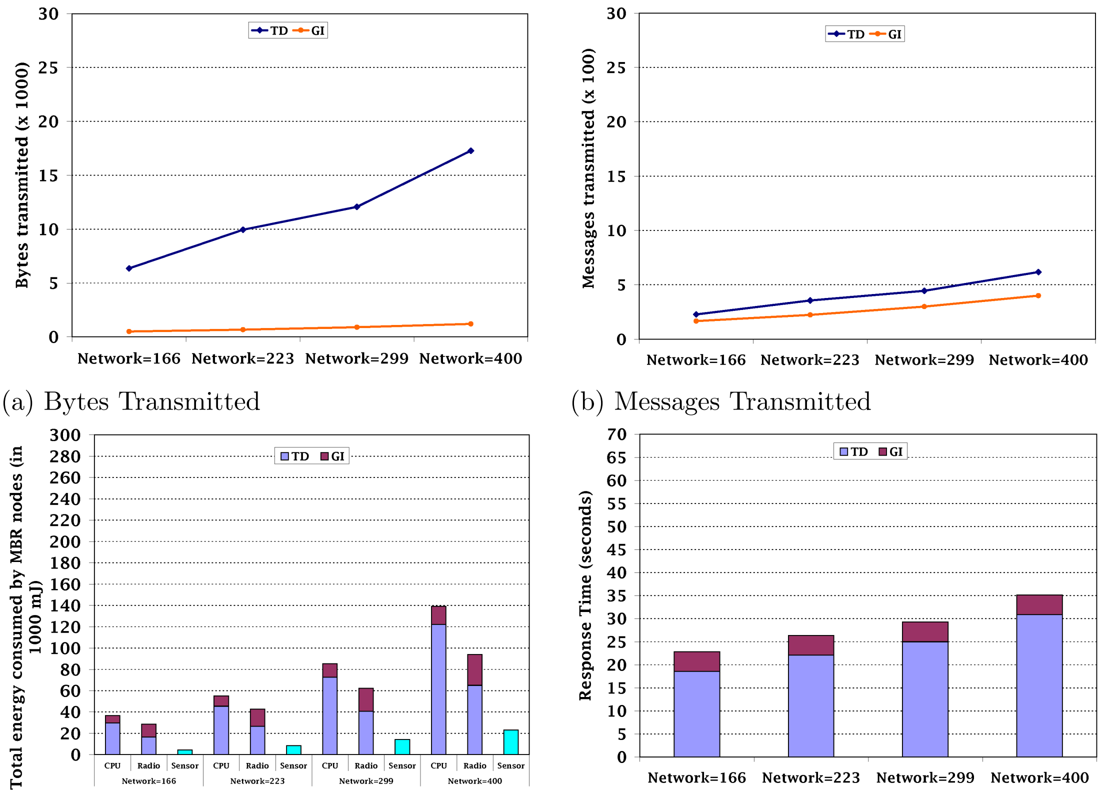
6.5 Geometry Induction

In this section, an experimental evaluation of geometry induction task is presented. In this experiment, a task requiring geometry induction is run over the whole network. Figure 6.1 shows an experimental scenario in which we have induced geometry, viz., an induced region `IR1` defined by the event predicate `Temperature`$<\theta$. For computing the induced geometry CFEBD algorithm [RZL06] discussed in Section 3.4.1 is applied. The task used in this experiment is:

```
Temperature < 10
```

The results obtained are depicted in Figure 6.4. Figure 6.4 gives the projection of the cost associated with inducing temperature geometry over the network of varying sizes. By 'TD' is meant task dissemination; by 'GI', geometry induction.

As explained in Section 3.4.2, that each node upon acquiring the reading from attached sensing device computes whether they are part of event geometry. CFEBD algorithm requires



(a) Bytes Transmitted (b) Messages Transmitted

(c) Energy Consumed (d) Response Time

Figure 6.4: Experiment GI1: Geometry Induction Behaviour w.r.t. Network Growth

each event and non-event node to transmit its event state to its neighbours. The event nodes upon reception of information from neighbours compute whether they are part of the boundary or of the interior of an induced geometry. The energy consumed by the CPU, radio and sensor components are plotted separately in Figure 6.4 (c). As can be seen from Figure 6.4, geometry induction algorithm exhibits slow rates of growth in terms of messages transmitted, energy consumed and response time as the network size grows.

6.6 In-Network Spatial Analysis Approach

In this section, the experimental evaluation of Boolean-valued and spatial-valued tasks is presented. This section comprises three subsections: Section 6.6.1 describes the experimental setup, Section 6.6.2 presents experiments related to evaluation of Boolean-valued tasks, and Section 6.6.3 presents experiments related to evaluation of spatial-valued tasks. In all experiments, it is assumed that the tasks related to detection of induced geometries are run over the whole network and the cost associated with the induction of induced geometries are not included in the experimental cost.

As sensor nodes have limited amount of memory, to avoid potential exhaustion of memory resources, possible strategies have been taken. First and foremost, spatial tasks are translated in the gateway into a compact byte representation in order to eliminate the need for a sophisticated parser on each mote and to reduce the size of the task message. Secondly, the task processing system at each node allows it for the stack-based evaluation of complex tasks. In case of, Boolean-valued tasks as described in Section 5.6.4, there is no need of maintaining the operations state stack, each operation in the complex task, results in increase in the task state size by 03 bits. Similarly, in case of spatial-valued tasks as described in Appendix B.3, there is no need of maintaining the operations state stack. Thirdly, an entry in the node GIT is a quadruple $\langle GeometryID, DataType, BoundaryNode, TTL\rangle$. Where $GeometryID$ and $DataType$ attributes are of type byte (i.e., short int), BoundaryNode of type Boolean and TTL of type long int (i.e., 04 bytes). Based on an application scenario, a node can be at maximum part of two asserted geometries (i.e., if lying on the boundary of two asserted geometries), and to none, one or more induced or derived geometries at each evaluation period.

6.6.1 Experimental Setup

The experimental setup is the same as in Section 6.4.1. The main difference is now discussed. As communication is the most expensive operation, it must be dealt with carefully. There have been many proposals for network management and for handling *media access control* (MAC).

Radio duty cycling is one of the techniques used to improve the longevity of the network. Low-Power-Listening MAC protocols are characterized by radio duty cycling. The idea behind is to prevent idle waste of energy and, thereby, meet the energy constraints of the application. Idle listening refers to the time when a node is listening on the radio to receive the messages, but no messages are being received. This is a WSN-specific MAC problem to a certain extent, as idle listening consumes energy. Therefore, in order to conserve energy, after task message dissemination and distributed membership evaluation, the nodes which are part of the task MBR move to a low power listening state [KM07, MM06, LWG05], from which they emerge to route packets if required. Since, it is not known in advance which of the MBR nodes will be SEG leaders and which of the nodes will act as relay nodes, putting all the MBR nodes in the Idle listening state would have been a very energy consuming option at power-up. Therefore, in order to conserve energy, after the distributed membership evaluation stage, all nodes that are in the task MBR move into a low power listening state. The TinyOS radio component for MICA2 motes is used to achieve this purpose as it supports a low power listening state. For task dissemination, and distributed membership evaluation, the node turns on its radio in full transmit/recieve mode whenever required.

The experimental results shown in this section always plot the average computed over ten runs of each experiment.

6.6.2 Boolean-Valued Operations

The experiments in this section provide evidence that the algorithms for Boolean-valued operations scale well in terms of bit and message complexity, energy consumption and response time.

Note that all experimental results are broken down into four components corresponding to the evaluation phases, viz., TD for task dissemination, DME for distributed membership evaluation, AGG for aggregation, RP for result processing. In the first two experiments, the task is evaluated using tree-based as well as gossip-based aggregation schemes. The motivation is twofold, viz., to show the cost associated using each of the aggregation scheme, and to demonstrate that more than one aggregation scheme can be used with the algorithmic strategy proposed for the evaluation of spatial-analysis tasks. The best aggregation policy to use depend on the application requirements. In experiments where tree-based aggregation scheme is used, TC denotes tree construction.

The time required for aggregation and routing is split into two parts, viz., AGGSGE, i.e., the amount of time taken to complete the aggregation at all SGEs, and AGGMGE, i.e., the time taken to compute the final result at the first-level leader. The first-level leader waits for a response from all SGEs before computing the final result. In the case of MEGs, the size of a geometry element may be such as to cause the nodes complete at different times in different elements so that the partial aggregated result is forwarded by the SEG leader of that element towards the MEG leader, while the aggregation process is still going on other elements. This may result in a overlap of the time line, as expected in any truly distributed computation. Thus, (AGGMGE + RP) denotes the time taken for the aggregation to complete at the last SEG plus that taken for the result to be received at the sink, and AGGSGE denotes the time taken for the aggregation to complete at the last SEG that completes.

6.6.2.1 Experiment I1: As the network grows in terms of number of sensor nodes

This experiment explores the behaviour of the implemented algorithms in evaluating a given task as the number of nodes in the network grows. The task used is one that, in terms of our motivating example, identifies whether spraying is required in a given field. Figure 6.5 shows field `f5` and two induced geometries, viz., an induced region `IR1` defined by the event predicate `soil_moisture`$>\theta$ and another induced region `IR2` defined by the event predicate `temperature`$<\theta'$. The task used in this experiment is:

```
(NOT ((IR1 AreaDisjoint f5) AND (IR2 AreaDisjoint f5)
                              AND (IR2 VertexDisjoint IR1)))
```

Our aim here is also to convey the expressiveness of the algebra which is why the task is expressed in a more complex form than needed (it could be more simply specified using a single operator as follows: `((IR1 intersects f5) and (IR2 intersects f5) and (IR2 intersects IR1)))`. The results obtained after evaluating the task on scenario in Figure 6.5 are depicted in Figure 6.6 (using tree-based aggregation) and Figure 6.7 (using gossip-based aggregation). The size of the geometries involved in the task are given in Table 6.2.

It can be seen in Figure 6.7(c) that gossip-based aggregation involves additional communication cost compared to the tree-based aggregation results in Figure 6.6(c). In addition, the rate of growth is much sharper for messages and bytes transmitted in gossip-based aggregation in Figure 6.7. The growth rate of energy consumption is less acute compared to that of the messages (and bytes) transmitted. The reason is that, in the case of tree-based aggregation the nodes have to wait to receive messages from child nodes [MFHH02] and the cost of idle listening

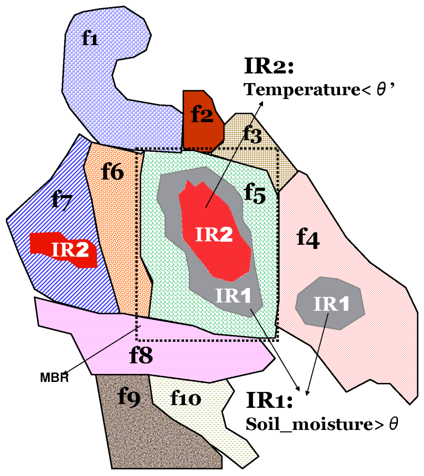


Figure 6.5: Experiment I1: Scenario (Intersection of two induced MEGs IR1 and IR2)

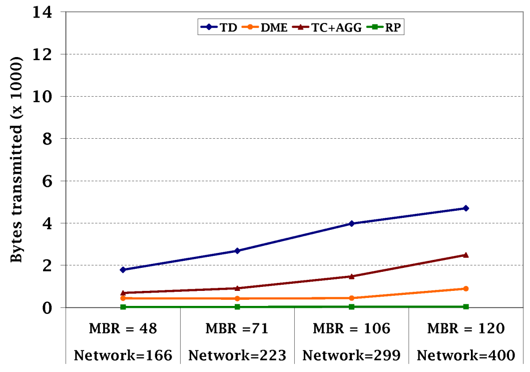
Network Size	f5	IR1 in f5	IR2 in f5	IR1 in f4	IR2 in f7
166	48	28	11	12	6
223	63	37	15	18	10
299	86	51	20	22	14
400	106	68	28	27	18

Table 6.2: Size of **regions** in Experiment I1

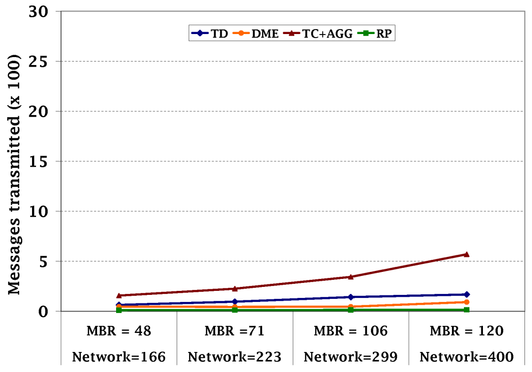
(when the radio is listening to the channel for upcoming messages) is significant and comparable to the energy cost of receiving data [DKSD09]. For example, in MICA2 nodes the ratio for radio power consumption, during idle-listening and during message receipt and transmission is 1:1:1.41 at 433MHz with *radio frequency* (RF) signal power of 1mW in transmission mode [DKSD09, RSZ04]. According to the power model for MICA02 motes given by PowerTOSSIM [SHrC$^+$04], the radio listening cost for incoming messages and for receiving messages is 7 mA and cost for transmitting the message to the distance of 6 to 8 meters is 11.6 mA.

As expected, a comparison of Figure 6.7(c) with Figure 6.6(c) shows that tree-based aggregation incurs slightly lower energy cost compared to gossip. Note that there is an inherent trade-off between tree-based and gossip-based aggregation. The tree-based algorithm is more energy efficient but is not as robust against node and link failures. On the other hand, the gossip-based approach is resilient to node and link failures but comparatively less energy efficient. The network user must choose the appropriate aggregation computation approach based on the application requirements and the hostility of the environment in which the sensor network has to be deployed.

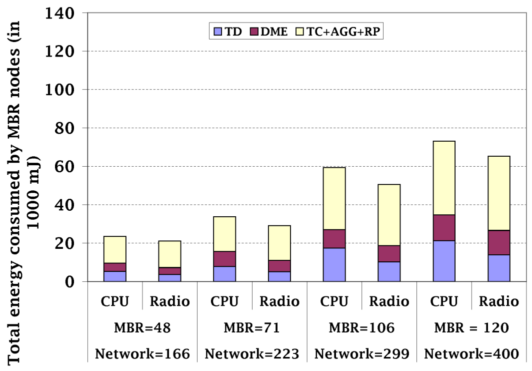
As can be seen from Figure 6.7 and Figure 6.6, our algorithms exhibits slow rates of growth in terms of messages transmitted, energy consumed and response time as the network size grows except for bytes and messages transmitted in the case of gossip-based aggregation. As expected, the most onerous phases are TD and AGG, as they are responsible for the majority of communication events. The energy consumption, in particular, seems to exhibit good behaviour.



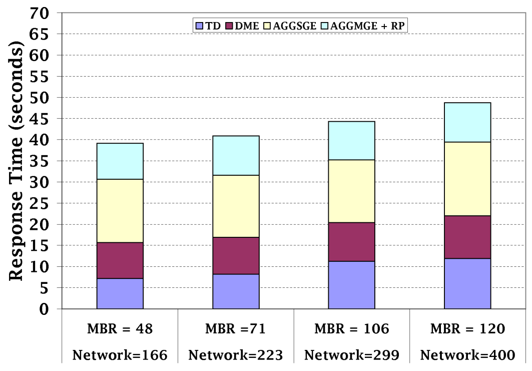
(a) Bytes Transmitted



(b) Messages Transmitted



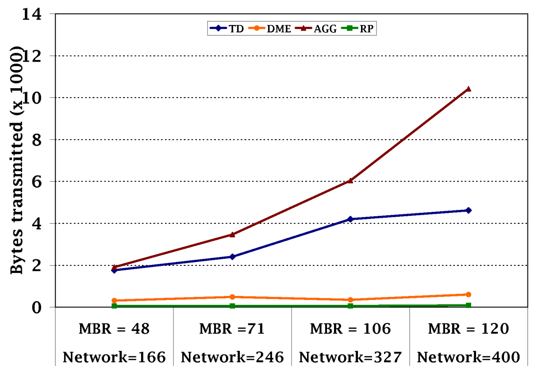
(c) Energy Consumed



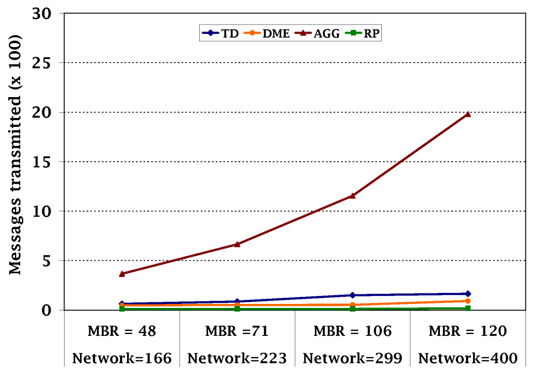
(d) Response Time

Figure 6.6: Experiment I1: Behaviour w.r.t. Network Growth (Tree Aggregation)

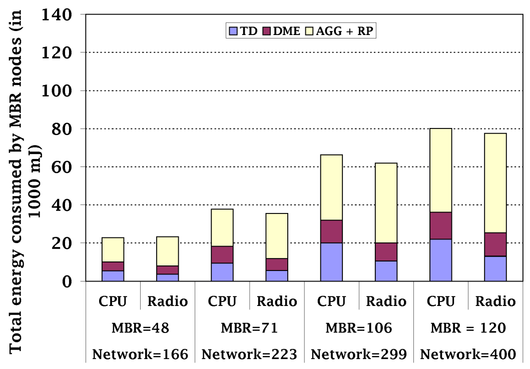
For the largest MBR (containing 120 nodes) the amount of energy consumed by CPU and radio together is 138,481 mJ (tree-based aggregation Figure 6.6 (c)) and 157,691 mJ (gossip-based aggregation Figure 6.7). If each node is powered by two AA batteries, the initial energy stock per node is 31,320,000 mJ. The total energy stock inside the MBR is then 120*31,320,000. This implies that each evaluation episode consumes around .0036% (tree-based aggregation) and 0.0041% (gossip-based aggregation) of the total energy stock. Even if the depletion is not uniform across different nodes, the overall amount depleted is very low. This is low enough that adding the energy required to induce the event geometries (not counted in Figure 6.6 and 6.7) is unlikely to significantly detract from the force of these conclusions (see Section 6.5). It can be seen that all costs for out-of-network processing Figure 6.3 and Figure 6.2 are very high compared to in-network implementation. By comparing the results of this experiment with the out-of-network processing approach in Figure 6.3, it can be seen that the in-network implementation transmits no more than half the messages, and often much less; it consumes no more than 2% of the energy and requires no more than 8% of the time to respond.



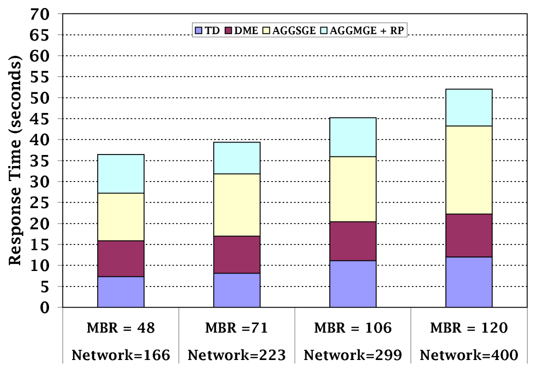
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.7: Experiment I1: Behaviour w.r.t. Network Growth (Gossip-based Aggregation)

6.6.2.2 Experiment I2: As the topology of the underlying transient phenomena undergoes significant dynamic changes between evaluation episodes

This experiment explores the behaviour of the implemented algorithms when, given a network consisting of 223 nodes, a transient physical phenomenon evolves through six stages. In terms of our motivating example, we note that water infiltration varies over space and time and hence influences soil moisture levels differently across monitoring episodes. It is, therefore, an example of an evolving phenomenon giving rise to transient geometries that differ between excessive evaluation episodes.

The six pictures in Figure 6.8 show an induced geometry `IR1` (defined by the event predicate `soil_moisture`>θ) as it undergoes six changes: at Eval1, it is a single-element geometry; at Eval2, the induced geometry has grown and moved; at Eval3, it has grown and changed shape; at Eval4, it has split and become a multi-element geometry; at Eval5, one of the component elements has developed a hole; at Eval6, one of the component elements has grown, and acquired a hole inside it. The task used in this experiment is:

```
((IR1 Intersects f5) AND
   ((IR1 BorderInCommon f6) OR (IR1 AreaInside f6)))
```

The size of geometries used in the task are given in Table 6.3.

Evaluations	Network	f5	f6	IR1		
	Size			IR1-1	IR1-2	IR1-2 hole
Eval 1	223	63	35	15	-	-
Eval 2	223	63	35	24	-	-
Eval 3	223	63	35	37	-	-
Eval 4	223	63	35	14	31	-
Eval 5	223	63	35	14	27	5
Eval 6	223	63	35	14	35	10

Table 6.3: Size of **regions** in Experiment I2

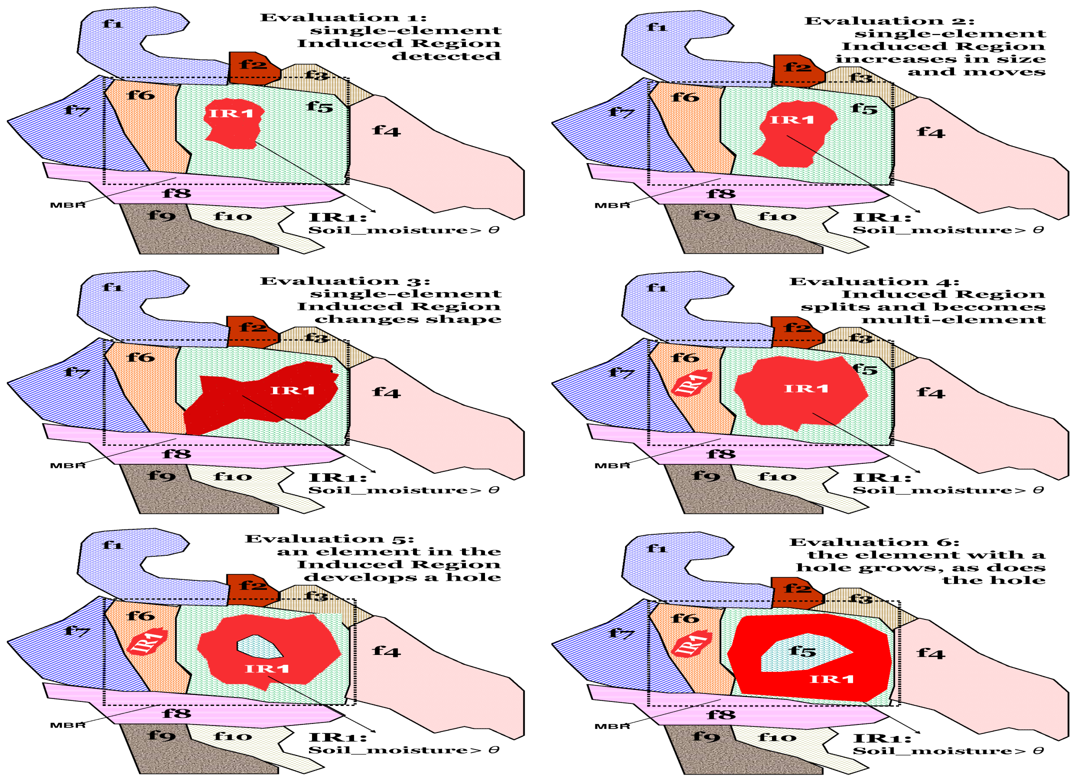
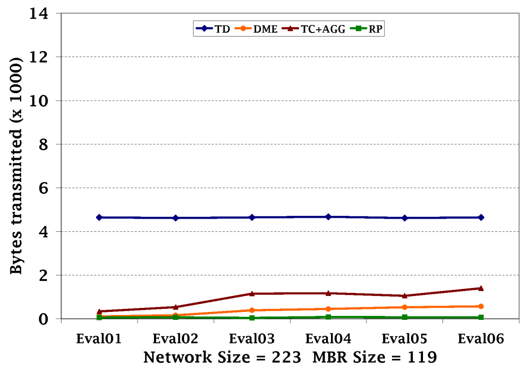

Evaluation 1: single-element Induced Region detected
Evaluation 2: single-element Induced Region increases in size and moves
Evaluation 3: single-element Induced Region changes shape
Evaluation 4: Induced Region splits and becomes multi-element
Evaluation 5: an element in the Induced Region develops a hole
Evaluation 6: the element with a hole grows, as does the hole
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
IR1
MBR
IR1: Soil_moisture> θ

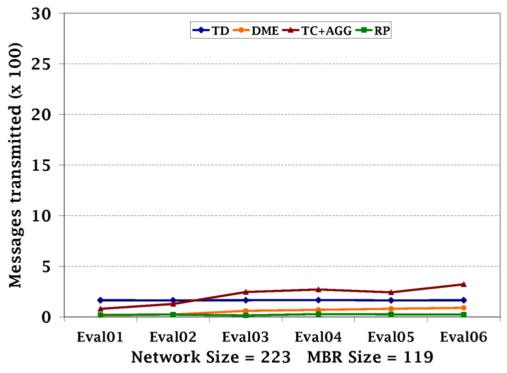
Figure 6.8: Experiment I2: Scenario (Six stages of an evolving phenomenon)

This section discusses the results obtained by running the same task over the experimental scenarios in Figure 6.8, using the two different aggregation methods. The results obtained using tree-based aggregation are depicted in Figure 6.9, those obtained using gossip-based aggregation method in Figure 6.10.

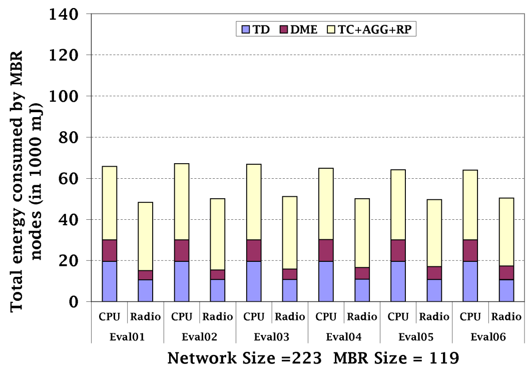
The results show that the algorithms yield roughly constant cost in terms of messages transmitted, energy consumed and response time as the geometries that characterize the evolving phenomena change over time. The increase in complexity, and the sparser, more spread-out nature of the geometry element that contains a hole, as expected, implies a more onerous AGG phase in terms of messages transmitted. However, as Fig.6.9- 6.10(**b**)-(**c**) show, there is very little variation in energy consumption and response time across the evaluation episodes, although, as already pointed out, gossip-based aggregation shows slow, linear-like growth, as expected.



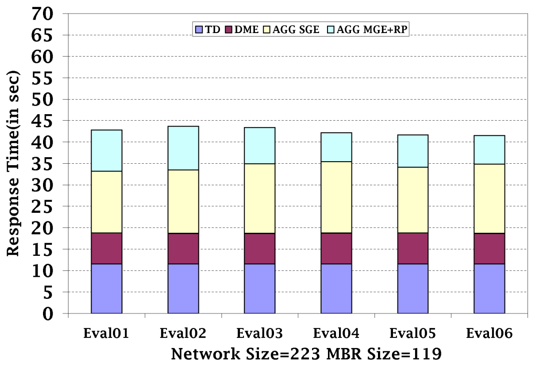
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.9: Experiment I2: Behaviour w.r.t. Topology of the underlying transient phenomenon undergoes changes between evaluation episodes (Tree-based Aggregation)

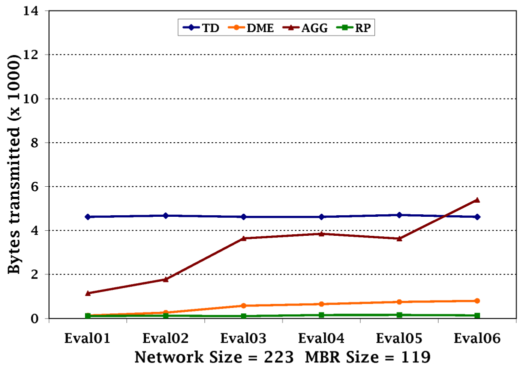
This is satisfying, as it suggests that in-network processing approach is stable in terms of energy consumed even as a transient phenomenon under observation undergoes significant topological change.

6.6.2.3 Experiment I3: As the spatial-analytic task complexity grows

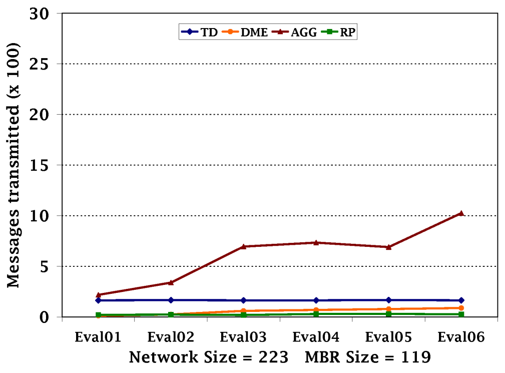
This experiment explores the behaviour of the implemented algorithms when, given a network (consisting of 223 nodes) and a particular set of geometries (see below), the task complexity grows. In terms of our motivating example, it seems likely that, whenever an indication is produced that action may be needed in one area of the vineyard, one would want to carry out more complex analysis to obtain a more complete indication as to which actions are needed where. In other words, there will be circumstances when a more exploratory mode of investigation will be needed, with the user issuing a sequence of tasks of increasing refinement and complexity.

This experiment uses the geometries depicted in Figure 6.11. The MBR is enclosing the regions $f1$, $f2$, $f3$, and $f6$. It contains one induced multi-element geometry `IR1` defined by the event predicate `soil_moisture`$>\theta$. The size of the geometries involved in the task are given in Table 6.4.

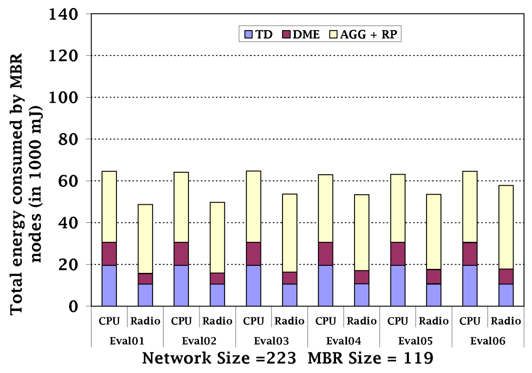
Four tasks of growing complexity in terms of the number of occurrences of an operator are



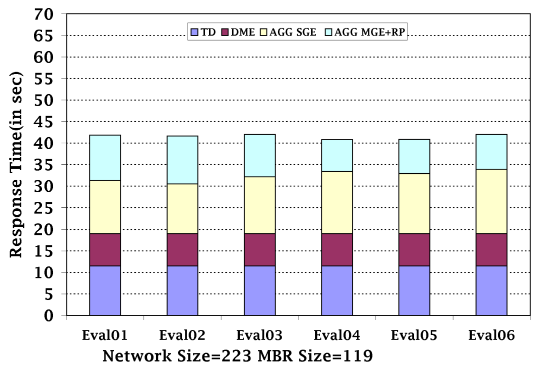
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.10: Experiment I2: Behaviour w.r.t. Topology of the underlying transient phenomenon undergoes changes between evaluation episodes (Gossip-based Aggregation)

Network Size	f1	f2	f3	f6	IR1	
					IR1-1	IR1-2
223	31	9	14	35	27	24

Table 6.4: Size of **regions** in Experiment I3

applied. The chosen operation, viz., `adjacent`, is amongst the more complex in the algebra in terms of its evaluation costs. The four tasks are:

```
Eval01: (IR1 Adjacent f1)

Eval02: (IR1 Adjacent f1) AND (IR1 Adjacent f2)

Eval03: (IR1 Adjacent f1) AND (IR1 Adjacent f2) AND
        (IR1 Adjacent f3)

Eval04: (IR1 Adjacent f1) AND (IR1 Adjacent f2) AND
        (IR1 Adjacent f3) AND (IR1 Adjacent f6)
```

The results obtained are depicted in Figure 6.12. As expected in Figure 6.12(c) the number of

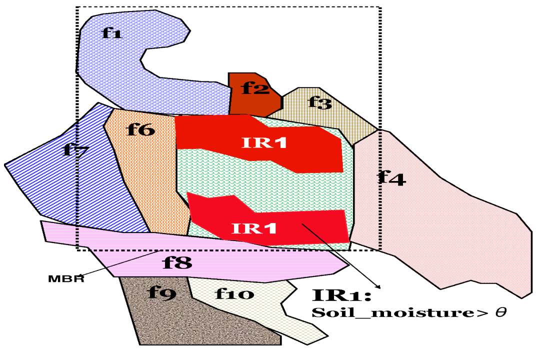
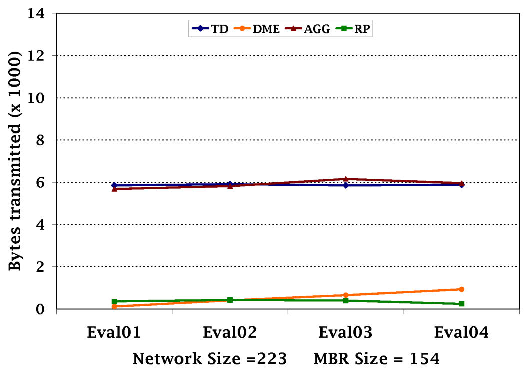
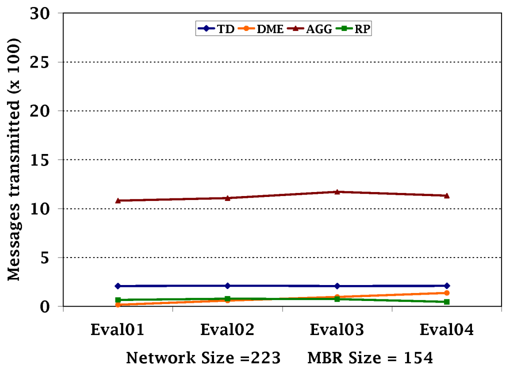


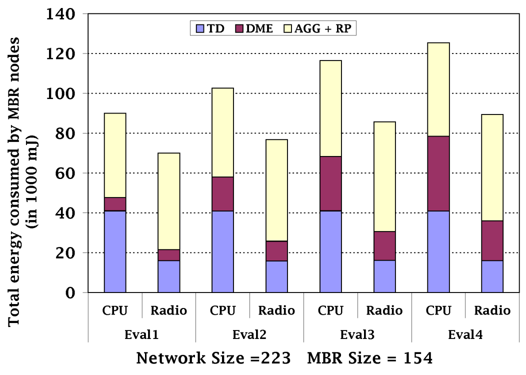
Figure 6.11: Experiment I3: Scenario (Event region with disjoint element geometries)



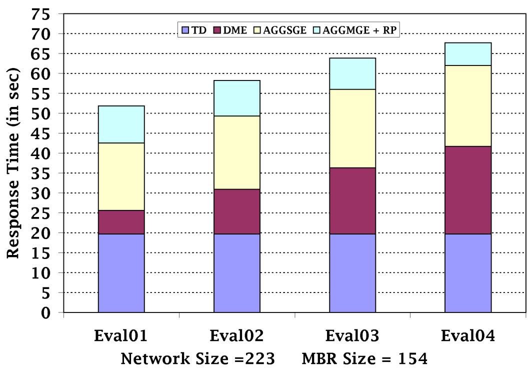
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.12: Experiment I3: Behaviour w.r.t. Spatial-analytic task grows more complex

messages transmitted during the DME phase increases with the addition of adjacent operation. As one more adjacent operation is added, the CPU energy grows by approx. 11% and radio energy by 10%. Once more, the implementation exhibits slow rates of growth in terms of bytes transmitted, messages transmitted and energy consumption as tasks grow in complexity from Eval1 to Eval4.

6.6.2.4 Experiment I4: As the induced geometry grows larger

This experiment explores the behaviour of the implemented algorithms when, given a network consisting of 223 nodes and a particular set of geometries, the size of the multi-element induced geometry grows. Experiment I2 explores the behaviour of the implemented algorithms as the topology of the underlying transient phenomena undergoes significant dynamic changes (i.e., increase/decrease in size, self-split, formation of hole, and increase in the size of hole) between evaluation episodes. The aim of this experiment is to show the approach used for the distributed evaluation of spatial tasks scale well in terms of number of nodes involved as part of the multi-element induced geometry.

This experiment uses the geometries depicted in Figure 6.13. The task used in this experiment is:

```
((IR1 VertexInside f5) OR (IR1 Intersects f5) OR (IR1 Meets f6))
```

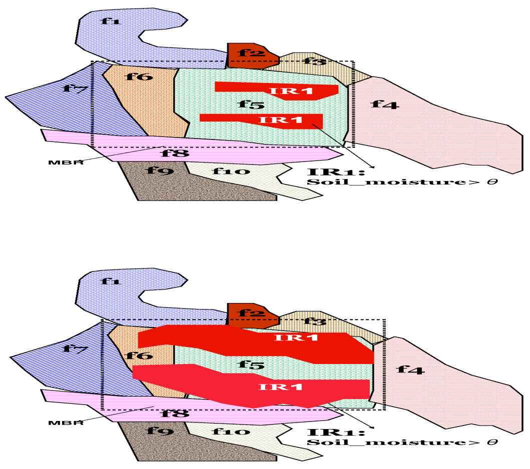


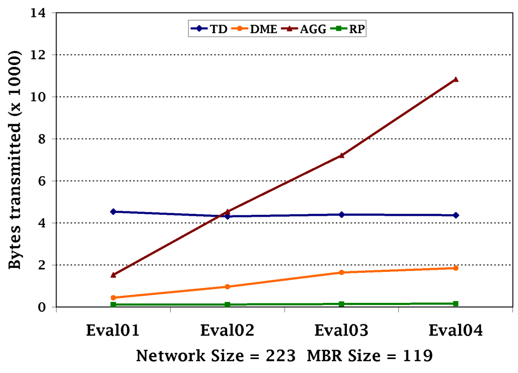
Figure 6.13: Experiment I4: Scenario (Four stages of an evolving phenomenon that grows)

The task involves induced multi-element geometry `IR1` defined by the event predicate `soil_moisture`>θ. The sizes of the geometries involved in the task are given in Table 6.5. The results obtained are depicted in Figure 6.14.

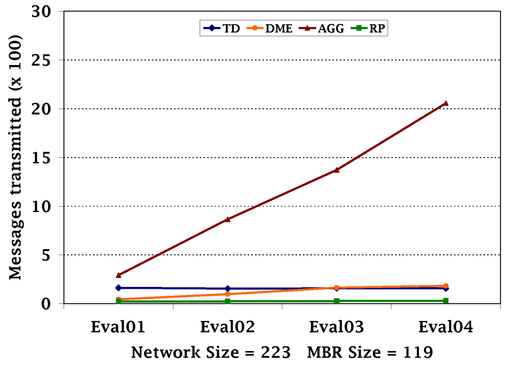
Network Size	f5	f6	IR	
			IR1-1	IR1-2
223	63	35	12	10
223	63	35	26	18
223	63	35	35	31
223	63	35	44	40

Table 6.5: Size of **regions** in Experiment I4

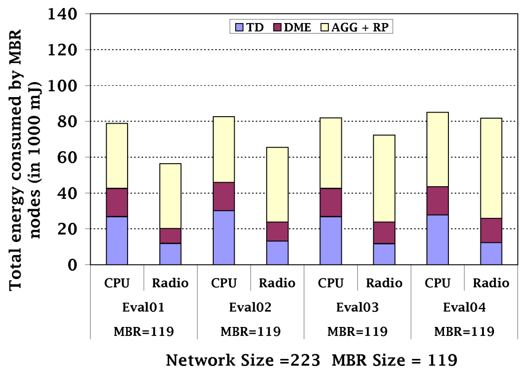
As expected, the increase in size of the induced geometry, implies a more onerous AGG phase in terms of messages transmitted and in turn in energy consumption, but the increase in size of the induced geometry does not affect response time by much.



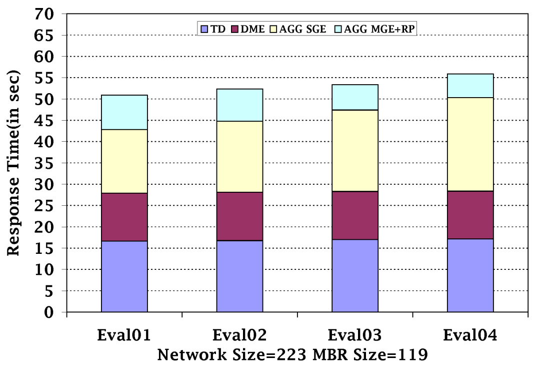
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.14: Experiment I4: Behaviour w.r.t. Induced geometry growth

6.6.2.5 Experiment I5: As the average node-neighbourhood cardinality grows

This experiment explores the behaviour of the implemented algorithms when, given a network consisting of 223 nodes and a particular set of geometries (see below), the size of the node neighbourhood grows. This experiment uses the geometries depicted in Figure 6.5. The task used in this experiment is:

```
(not ((IR1 EdgeDisjoint f5) and (IR2 VertexDisjoint f5))) or (IR1 EdgeInside IR2)
```

The experimental scenario contains two induced multi-element geometries `IR1` and `IR2` defined by the event predicate `soil_moisture`$>\theta$ and `temperature`$>\theta$'. Table 6.6 depicts the size of the geometries involved in the task and the neighbourhood size, per evaluation episode. The neighbourhood size is computed by increasing the radio range by factor of 1, 1.25, 1.5, and 1.75, while keeping the distance between nodes constant.

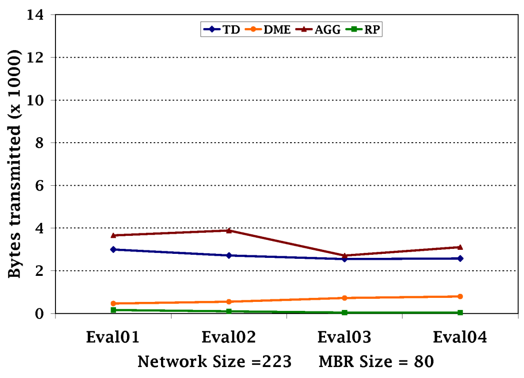
The results obtained are depicted in Figure 6.15. The increase in size of the radio range (i.e., node neighbourhood), as expected, results in a reduction of messages transmitted in the TD phase and an increase in messages transmitted during DME. The reason for the decrease in messages is due to the fact that by increasing the radio range the neighbourhood of a node increases. By increasing the radio range, the task dissemination message sent by a node is received by a larger number of neighbours and as a result, upon finding themselves part of the

Eval	Neighbourhood Size	f5	IR1 in f5	IR2 in f5	IR1 in f4	IR2 in f7
Eval01	8	63	37	15	18	10
Eval02	12	63	37	15	18	10
Eval03	20	63	37	15	18	10
Eval04	24	63	37	15	18	10

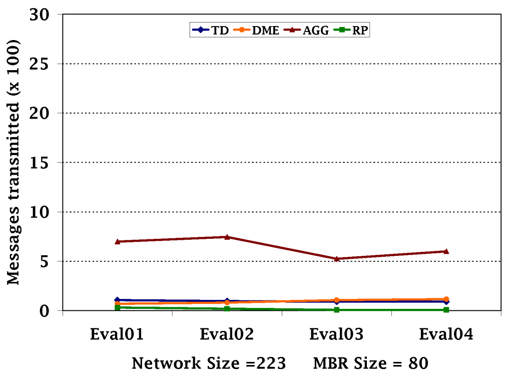
Table 6.6: Size of **regions** in Experiment I5

task MBR, they start participating in task dissemination. In this way, the task is disseminated much faster in the region of interest.

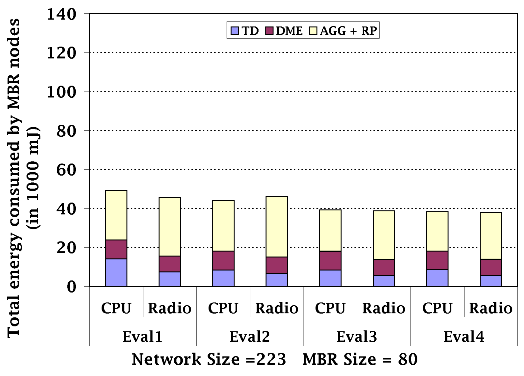
The increase in radio range, therefore, decreases the depth of the tree (i.e., the number of hops required decreases) that is built during task dissemination. As already explained in Section 5.4.2, after receiving the task, an MBR node may need to wait until the estimated (on the basis its location in the task MBR) time needed for the task message to be received by the furthest node in the MBR. Therefore, an MBR node needs to wait less after receiving the task.



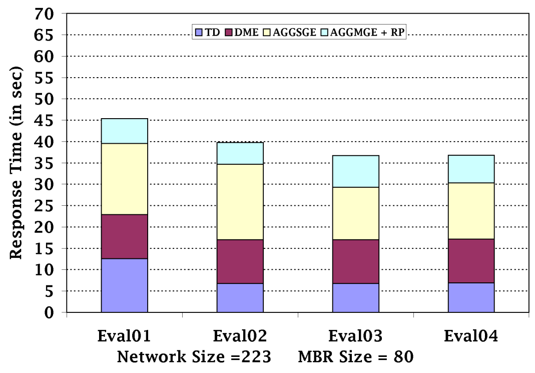
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.15: Experiment I5: Behaviour w.r.t. Average node-neighbourhood cardinality growth

6.6.2.6 Experiment I6: As the network (regularly deployed over a irregular grid) grows in terms of number of sensor nodes

This experiment explores the behaviour of the implemented algorithms as the network (regularly deployed over a irregular grid) grows in terms of number of sensor nodes. The task used in this experiment is:

```
((IR1 Intersects f7) and (IR1 Intersects IR2))
```

The experimental scenario is given in Figure 6.16. In this experiment, apart from the fields we have an additional application-specific geometry lake (represented by a white blob). As described in Section 3.2.1, our framework allows the overall disposition of nodes in the two dimensional Euclidean plane to be regular (i.e., grid-like) or not. In this experiment, no nodes are deployed where the lake is situated, i.e., it gives rise to a WSN with a hole in it. Therefore, we cannot induce geometries or associate asserted geometries to this part of the Euclidean plane. This is more flexible than the original conception of the ROSE algebra. The underlying *realm* is a regular discrete point grid.

Table 6.7 show the size of the geometries involved in the task. The results obtained are depicted in Figure 6.17.

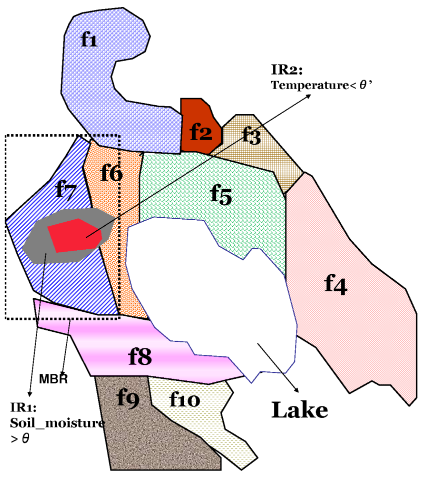


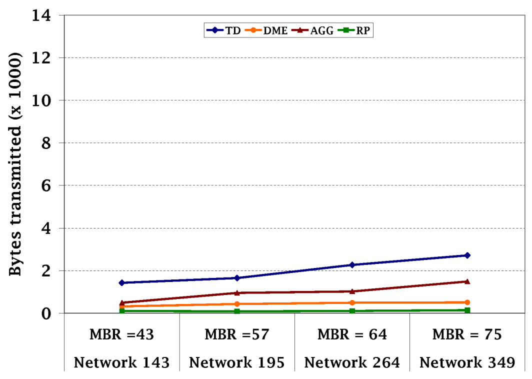
Figure 6.16: Experiment I6: Scenario (Network regularly deployed over a irregular grid)

6.6.2.7 Experiment I7: Spatial-analytic task involving operands other than of type Regions

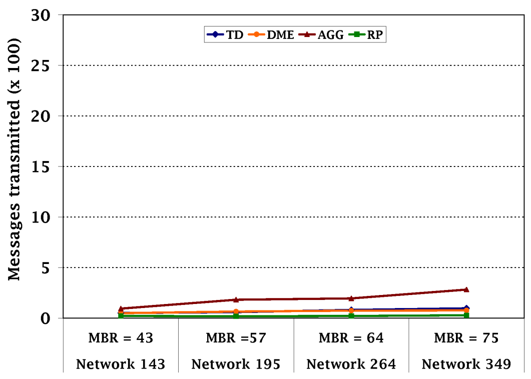
This experiment explores the behaviour of operations on operands of types other than **regions**. Given a network consisting of 223 nodes, spatial-analytic tasks involving one or more operands of type **regions**, **lines**, or **points** are evaluated. These tasks are:

Network Size	f7	IR1	IR2
143	26	10	7
195	34	13	7
264	42	17	9
349	56	25	14

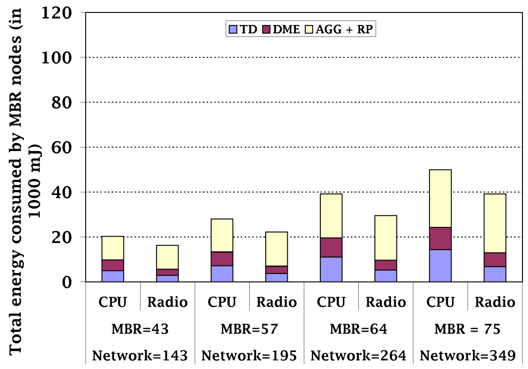
Table 6.7: Size of **regions** in Experiment I6



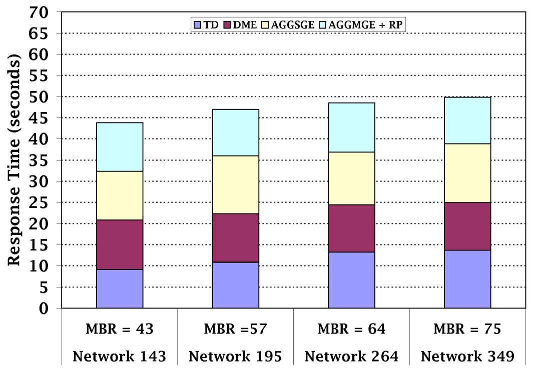
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.17: Experiment I6: Behaviour w.r.t. Network (regularly deployed over a irregular grid) growth in terms of number of sensor nodes

```
Eval01: (IR1 Intersects  l2)

Eval02: (p2 OnBorderOf IR1)

Eval03: (l2 AreaInside IR1)

Eval04: (l1 Meets l2)

Eval05: (l2 BorderInCommon IR1)
```

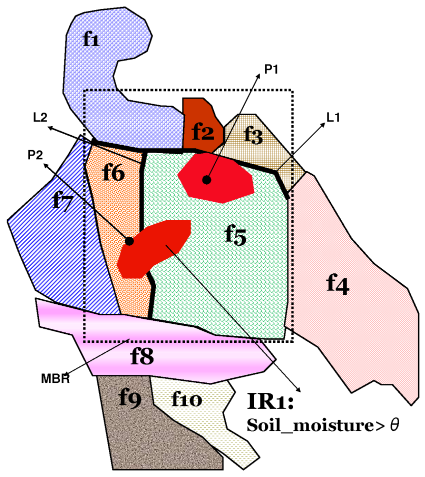


Figure 6.18: Experiment I7: Scenario (Distinct Geometries)

The tasks involve an induced multi-element geometry IR1 (defined by soil_moisture>θ), two geometries, p1 and p2, of type **points** and two geometries, l1 and l2, of type **lines**. This experiment uses the geometries depicted in Figure 6.18. Table 6.8 gives the size of the geometries involved in the task. Figure 6.19 shows the experimental results.

Network Size	f5	f6	l1	l2	p1	p2	IR1	
							IR1-1	IR1-2
223	63	35	10	8	1	1	15	12

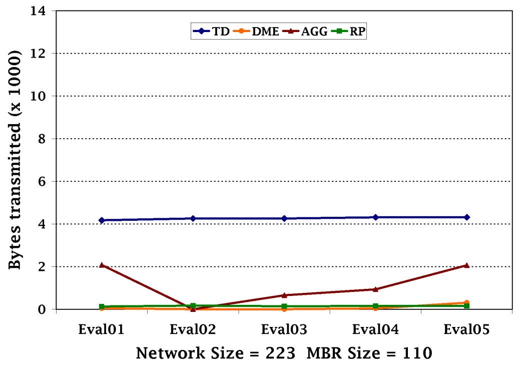
Table 6.8: Size of geometries in Experiment I7

In Figure 6.19(b) at Eval02, the number of messages in AGG phase drops to zero as the operand $p2$ value is of type **points**, and it declares itself SEG leader and transmits its membership state towards MEG leader. In 6.19(c), the radio cost for DME phase in Eval02 and Eval03 is minimal as operations involved in these two evaluations require only local GIT lookup and do not require any information from neighbours.

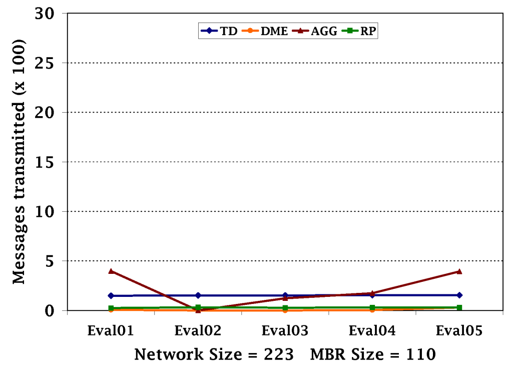
6.6.2.8 Experiment I8: Randomly generated tasks

Experiment I3 explores the behaviour of the implemented algorithms when, the task complexity grows (i.e., in terms of the number of operators and the operands involved) at each evaluation period. Experiment I2 explores the behaviour when, the task remains the same but one of the operands topology complexity grows. Experiment I7 explores the behaviour of spatial-analytic task involving operands other than of type Regions.

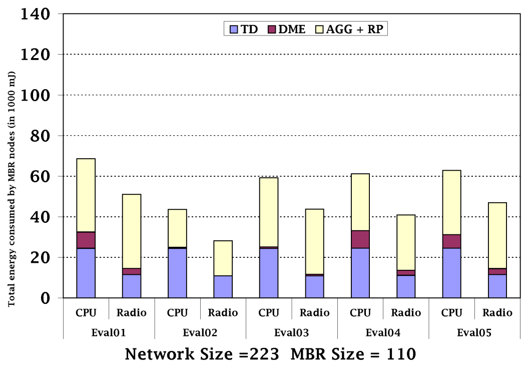
The aim of this experiment is to measure the relative performance of randomly generated Boolean-valued tasks. The aim is to show some experimental evidence that the algorithms produce consistent results even when the experimenter did not decide on the task to be submitted. Given a network (consisting of 166 nodes), spatial-analytic tasks involve one or more operands of type **regions**, **lines**, or **points**. Fifteen randomly generated tasks of varying complexity, are applied. The fifteen tasks are:



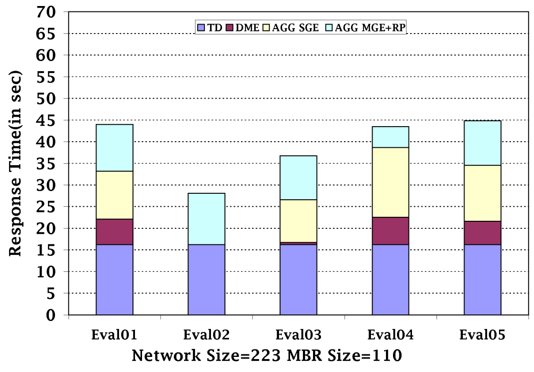
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.19: Experiment I7: Behaviour w.r.t. Boolean task varies in terms of operation involving one or more operands other than of type **regions**

```
Eval01: (f1 AreaDisjoint l2)

Eval02: (IR1 BorderInCommon f5)

Eval03: (f5 VertexInside IR1) AND NOT(f1 VertexInside f9)

Eval04: (f6 BorderInCommon f7) OR  NOT(f6 Meets f8)

Eval05: NOT(f2 AreaDisjoint f3)

Eval06: (f10 VertexDisjoint f7)

Eval07: (f7 VertexDisjoint f5) AND (f3 VertexInside IR1) OR
        (f5 AreaInside f4) OR (f3 VertexDisjoint l1)

Eval08: (f10 Equals f9) OR (f3 VertexDisjoint f1) OR
        (f10 NotEquals f9) OR (l2 Meets f5)
```

```
Eval09: (p1 VertexDisjoint f6) OR (l2  BorderInCommon IR2) OR  (f3
Meets l2) AND (l2 Intersects f5)

Eval10: (f9 Adjacent IR2) OR (f8  AreaInside f10) AND  (l1  VertexDisjoint f5)

Eval11: NOT(f6  Meets  IR1)

Eval12: (l2 Intersects IR2) AND (f8  Meets l1)

Eval13: (l1 BorderInCommon f10) AND (f8 Adjacent f1)  AND (f7
AreaInside f6)

Eval14: (f5 AreaInside f2)

Eval15: NOT((f4 VertexInside f2) AND  (f9  EdgeInside l1) OR (f6
AreaDisjoint f8))
```

The experimental scenario is given in Figure 6.20. The size of geometries used in the task is given in Table 6.9. Figure 6.21 shows the experimental results. From the experimental results, it can be seen that the algorithms behave sensibly with the increase in the complexity of the task. It can be seen that cost associated with task evaluation varies based on task complexity.

Recall, from Section 5.4.2.4, that *Task complexity* is based on the number of operations in the task, whether the operation only needs a local GIT lookup or, additionally, the collection of GIT information from the node's neighbours, the operands are SEG or MEG and, finally, how independent, in terms of the sensor space, the operand elements of an operation are with respect to other operations in the complex task. In Figure 6.20(b) at Eval07, it can be seen that the number of messages transmitted in AGG phase and TD phase are much greater as compared to other evaluations. The reason for the increase in cost is the increase in complexity of the task. The task involve operands that are independent, in terms of the sensor space, with respect to other operations in the task and having large number of nodes.

6.6.3 Spatial-Valued Operations

The experiments in this section provide evidence that the algorithms for spatial-valued operations scale well in terms of bit and message complexity, energy consumption and response time.

6.6.3.1 Experiment I9: As the network grows in terms of number of sensor nodes.

This experiment explores the behaviour of the implemented algorithms when, given a task, the number of nodes in the network grows. The task used is one that, in terms of our motivating example, derives the geometry of an area in need of spraying. Figure 6.5 shows field `f5` and two induced geometries, viz., an induced region `IR1` defined by the event predicate `soil_moisture`$>\theta$

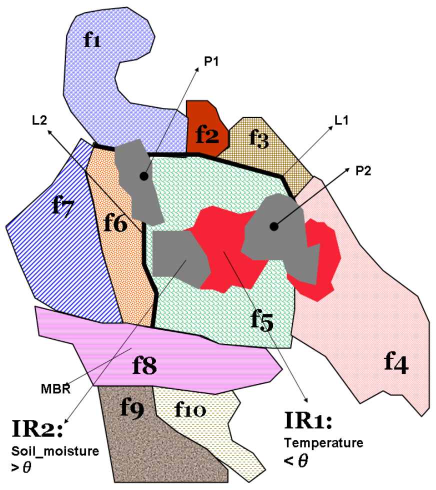


Figure 6.20: Experiment I8: Scenario (Distinct Geometries)

Geometry ID	Number Of Nodes		
f1	23		
f2	5		
f3	9		
f4	34		
f5	42		
f6	14		
f7	27		
f8	25		
f9	13		
f10	14		
IR1	12	11	
IR2	6	7	10
l1	8		
l2	7		
p1	1		
p2	1		

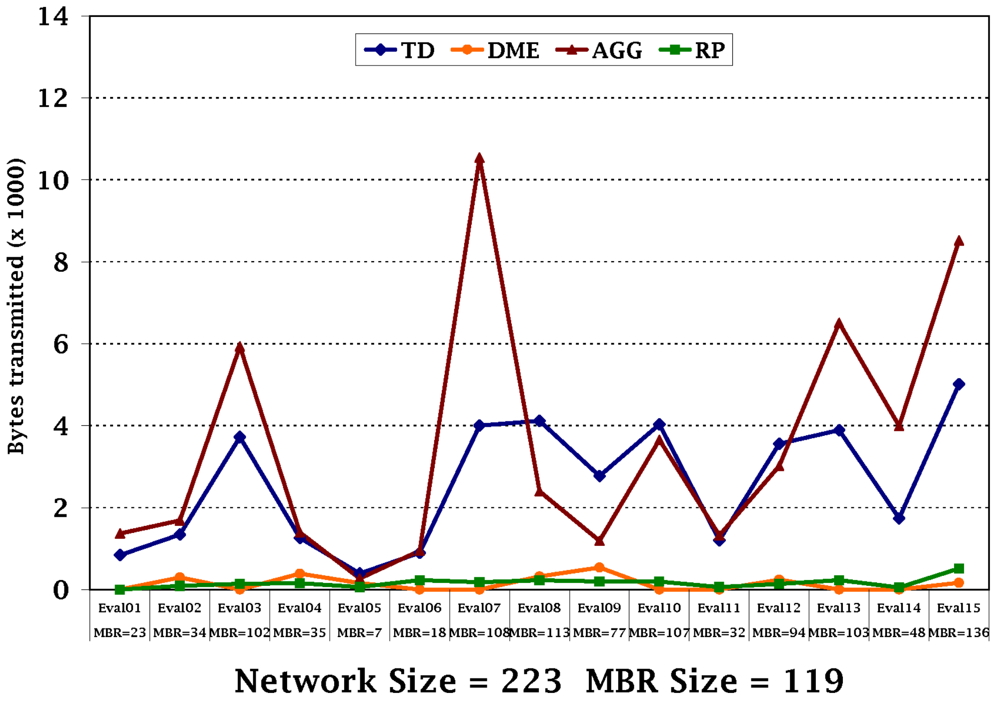
Table 6.9: Size of geometries in Experiment I8

and another induced region `IR2` defined by the event predicate `temperature`$<\theta'$. The task used in experiment is:

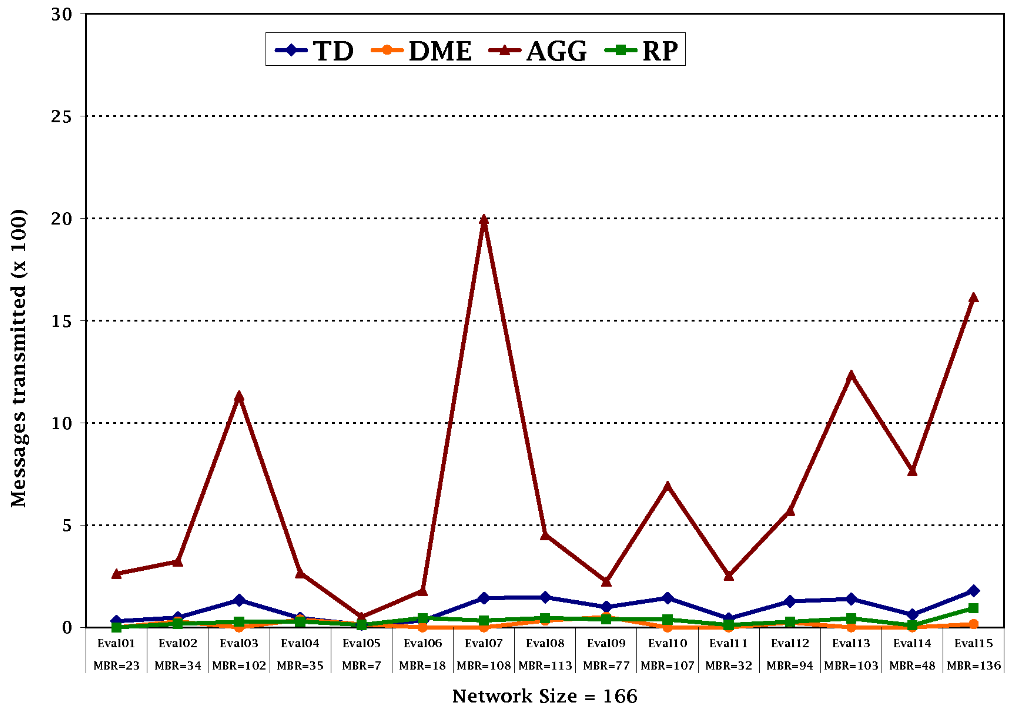
```
(f5 Intersection IR1) Intersection IR2)
```

The size of the geometries were given above, in Section 6.6.2.1, Table 6.2. The results are given in Figure 6.22.

In contrast with experiment 6.6.2.1, this experiment does not involve aggregation and routing. As a result of the evaluation of the task, nodes that are part of derived geometry store information about the derived geometry in their GIT and EIT.

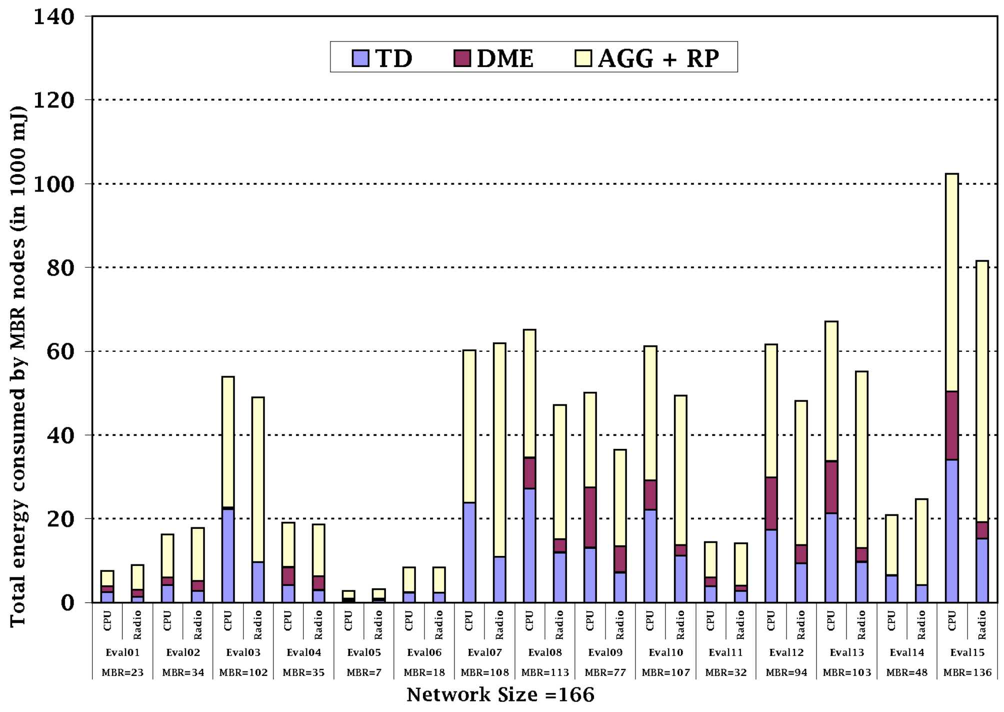


(a) Bytes Transmitted

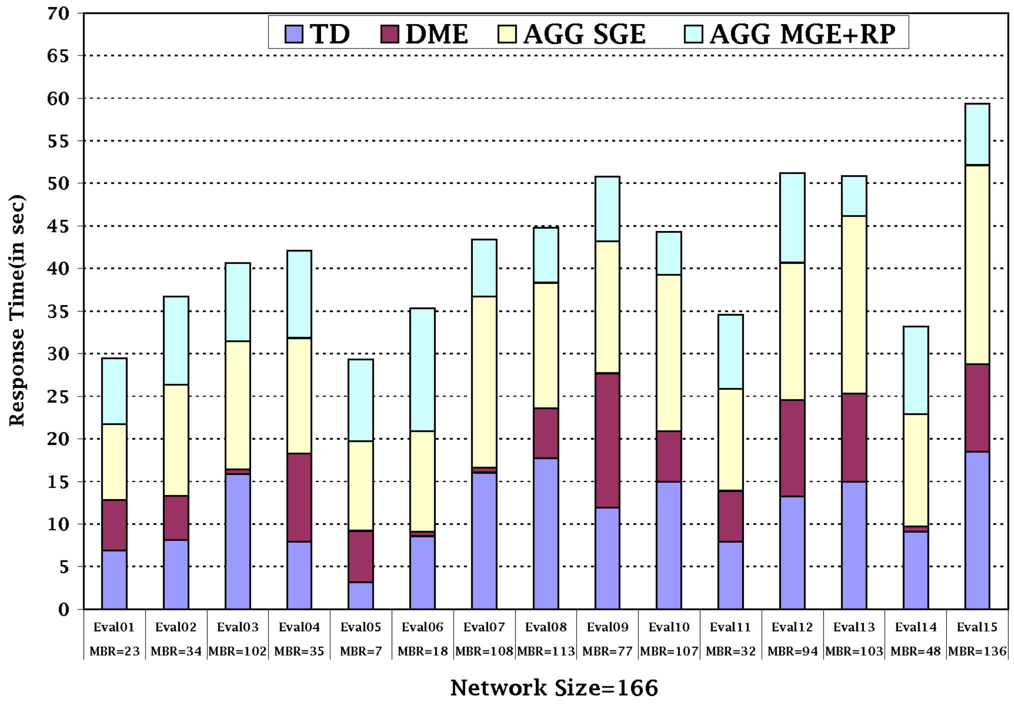


(b) Messages Transmitted

Figure 6.21: Experiment I8: Behaviour w.r.t. Randomly generated spatial-analytical tasks

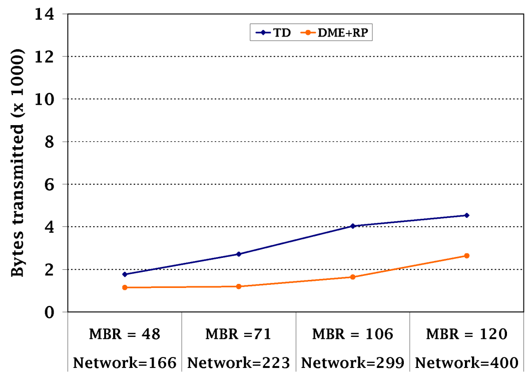


(c) Energy Consumed

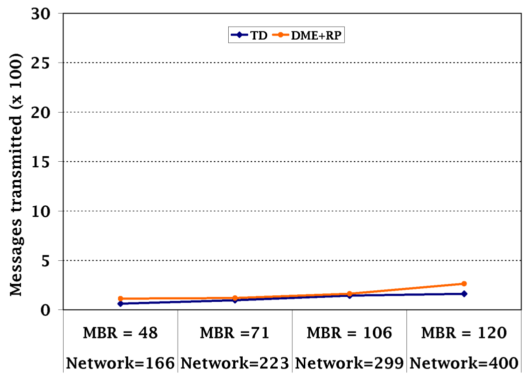


(d) Response Time

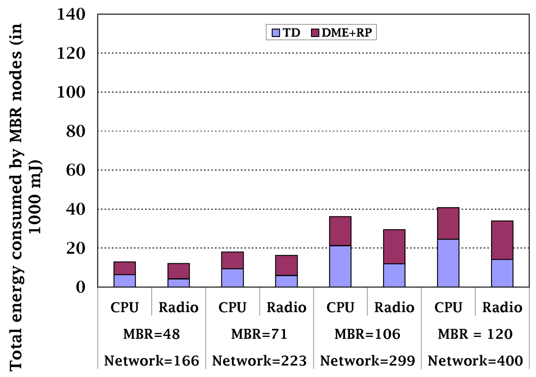
Experiment I8: Behaviour w.r.t. Randomly generated spatial-analytical tasks (Figure 6.21 continued)



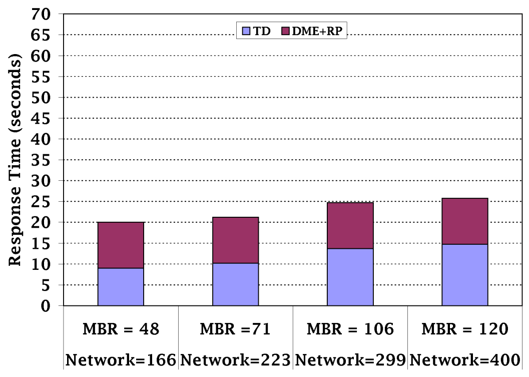
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.22: Experiment I9: Behaviour of Spatial-valued task w.r.t. Network Growth

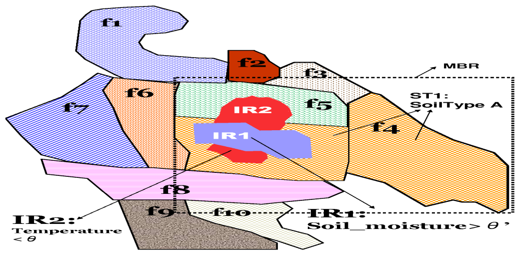
6.6.3.2 Experiment I10: As the topology of the underlying transient phenomena undergoes significant dynamic changes between evaluation episodes

This experiment explores the behaviour of the implemented algorithms when, given a task, the topology of the underlying transient phenomena undergoes significant dynamic changes between evaluation episodes. The purpose of the task is to derive a region that has *soil_type A* (i.e., $ST1$) and within which neither the temperature is below a threshold nor the soil moisture is above another threshold, or else either the temperature is below threshold, or the moisture is above threshold. In terms of our motivating example, water infiltration and temperature influences soil moisture levels differently based on type of soil. Wine makers achieve greater control over the product by defining sub-areas within the vineyard. Batch selection can then be decided based upon spatial locality, and the type of soil, along with other factors.

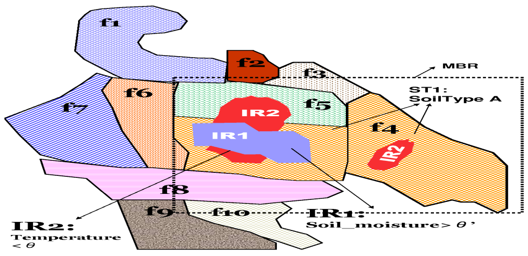
The four pictures in Figure 6.23 show two induced geometries, viz., an induced region `IR1` defined by the event predicate `soil_moisture`$>\theta$ and another induced region `IR2` defined by the event predicate `temperature`$<\theta'$ undergoes four changes. The task used in the experiment is:

```
(((f5 INTERSECTION ST1) PLUS f4) MINUS (IR1 INTERSECTION IR2))
```

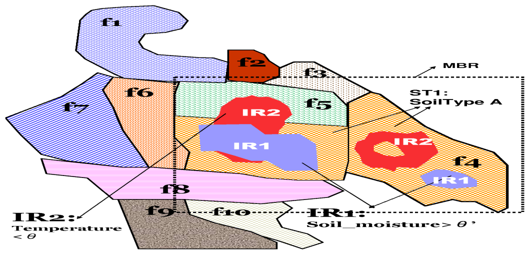
The size of geometries used in the task is given in Table 6.10. The experimental results are presented in Figure 6.24.



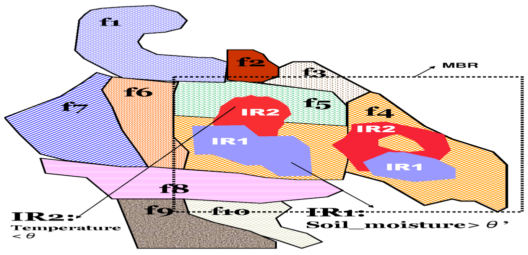
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

Figure 6.23: Experiment I10: Scenario (Four stages of evolving phenomena)

Evaluations	Network Size	f5	f5 with $ST1$	f4	IR1			IR2	
					IR1-1	IR1-2	IR1-2 hole	IR2-1	IR2-2
Eval 1	223	63	39	40	31	-	-	17	-
Eval 2	223	63	39	40	31	9	-	24	-
Eval 3	223	63	39	40	31	15	7	30	7
Eval 4	223	63	39	40	31	24	7	30	17

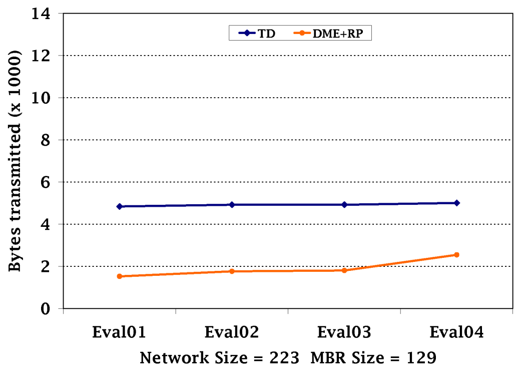
Table 6.10: Size of **regions** in Experiment I10

In Figure 6.24(b) it can be seen that the number of messages transmitted only slightly increases with the increase in complexity of the induced geometry.

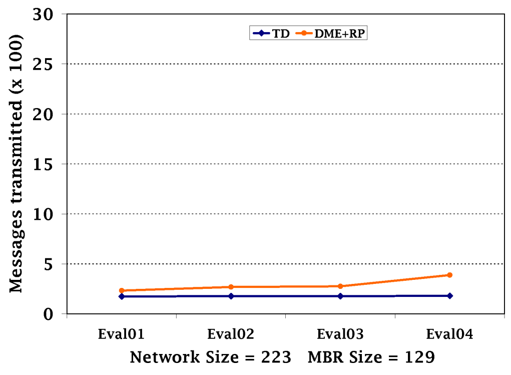
6.6.3.3 Experiment I11: Spatial-analytic tasks with one or both operands of a type other than Regions

This experiment explores the behaviour of operations on operands of type other than **regions**. Given a network consisting of 223 nodes, spatial-analytic tasks involving one or more operands other than of type **lines**, or **points** are run. The tasks are:

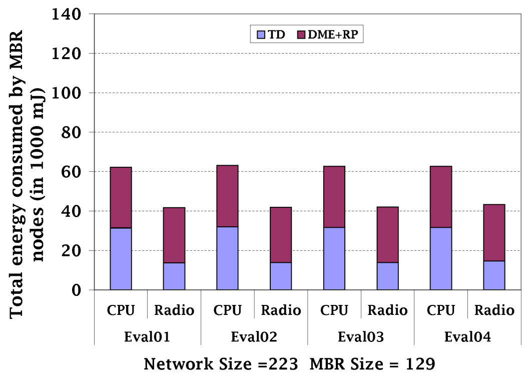
```
Eval01: (IR1 Intersection  l2)

Eval02: (l1 PLUS l2)

Eval03: (Vertices IR1)

Eval04: (l1 Intersection l2)

Eval05: (IR1 CommonBorder l2)
```



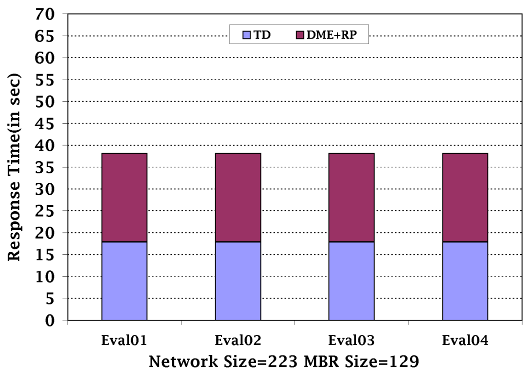
(a) Bytes Transmitted



(b) Messages Transmitted



(c) Energy Consumed



(d) Response Time

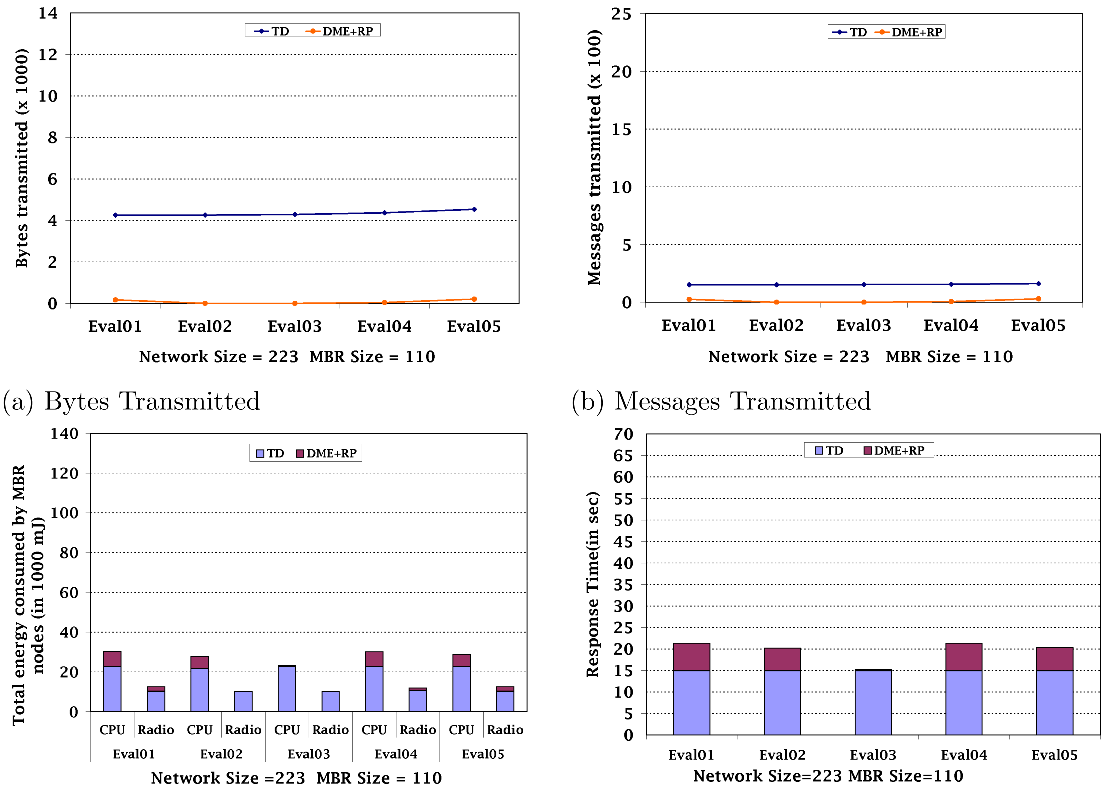
Figure 6.24: Experiment I10: Behaviour of Spatial-valued task w.r.t. Topology of the underlying transient phenomena undergo changes between evaluation episodes

The tasks involve an induced multi-element geometry, `IR1`, where `soil_moisture`>θ, two geometries `p1` and `p2` of type **points**, and two geometries `l1` and `l2` of type **lines**. This experiment uses the geometries depicted in Figure 6.18. Table 6.8 gives the size of the geometries involved in the task. Figure 6.25 shows the experimental results.

In Figure 6.25(b) at Eval02 and Eval03, the number of messages in DME phase drops to zero because for the computation of the membership state for the operators `plus` and `vertices` the nodes require information available in their local GIT. Therefore, in 6.19(c) the radio cost for Eval02 and Eval03 is null.

6.7 Summary

This chapter described experiments that constitute empirical evidence for the claim of an efficient implementation of the algebra and demonstrated that efficient in-network processing in WSNs is possible. The algorithms are, for the most part, localized, and were shown in our experiments to have very desirable behaviour in terms of byte, message complexity, response time and energy consumption. Experimental results show that in-network evaluation of spatial analytic tasks results in substantial savings in terms of energy consumption and response time



(a) Bytes Transmitted (b) Messages Transmitted

(c) Energy Consumed (d) Response Time

Figure 6.25: Experiment I11: Behaviour w.r.t. Spatial-analytic task varies in terms of operation involving one or more operands other than of type **regions**

(as well as, as expected, in message and bit complexity) compared to sending all sensed measurements or boundary information back to the sink. These results strongly suggest that our contributions advance the state-of-the-art towards the goal of enabling the high-level specification of expressive spatial analysis over WSNs.

Chapter 7

Conclusions and Future Work

This chapter concludes the dissertation. A review of the research motivation and the aims of this dissertation is given in Section 7.1. The main results and contributions to distributed spatial analysis are summarised in Section 7.2. Directions for future work are discussed in Section 7.3. Section 7.4 summarizes the chapter.

7.1 Overview

The dissertation has focused on the challenges involved in performing in-network spatial analysis using WSNs. It has argued that WSNs can be used effectively as spatial information systems, allowing management decisions about processes in the physical world to be made on the basis of sensed data. The aim of this dissertation has been to contribute to the enabling of in-network distributed spatial analysis of spatial phenomena that can be sensed using WSNs.

This aim has been pursued through the definition of a framework for representing geometries over WSNs, thereby allowing the representation of asserted, induced and derived geometries. The dissertation then showed how a centralized spatial algebra can be adapted and redefined over this spatial framework. The resulting operations enable the expression of sophisticated spatial analysis over WSNs.

The focus of the research reported has been on developing algorithms for efficient in-network evaluation of complex tasks that are expressions in the algebra mentioned above. This dissertation has described distributed implementations of both Boolean- and spatial-valued algebraic operations over the geometries represented by the framework, thereby enabling new geometries to be derived from induced and asserted ones, and relationships to be checked between spatially-referenced entities.

The research contributions in this dissertation demonstrate that in-network spatial analysis in WSNs can be effective and efficient. The evaluation results support this claim in that the implemented-algorithms were shown, empirically, to exhibit desirable behaviour regarding bit and message complexity, energy consumption and response time to an extent that justifies their use in practice. Equipping each node with a task processing engine allows for the specification of declarative spatial-analytical tasks that the WSN can compute the results for. Given that, as argued in Chapter 1, programming WSNs requires specialized knowledge on several grounds scarcity of the resources puts a tight limit on code size and debugging is cumbersome, this algebraic approach provides a cost-effective way of interacting with WSNs. It allows more application requirements to be specified at high-level and expressively enough to enable sophisticated WSN applications, such as environmental monitoring and precision agriculture.

7.2 Significance of Major Results

The major results of this dissertation are:

- The definition of a framework for distributed spatial analysis over WSN. It is to the best of our knowledge, the first generic framework capable of underpinning an algebraic approach to spatial analysis in WSNs as a distributed platform for in-network processing.

- The definition of spatial algebra over the geometries representable by the framework above. The algebra builds upon the Schneider centralized ROSE algebra but this has adapted it to a completely different computational environment requiring a different approach to representing geometries and to perform operations over them. Moreover, we have identified problems with the original formalizations and found solutions that have improved it.

- Distributed in-network algorithms for spatial and Boolean-valued operations in the contributed spatial algebra over the geometries that can be represented over the contributed framework. The algorithms for these operations are divided into logically-cohesive components and are specifically tailored for power-efficient in-network execution. This dissertation showed that the algorithms for topological operations map the problem of distributed computation of complex algebraic expressions that involve multi-element geometries to the problem of first computing a node-level task state and then aggregating that at two levels using a new bit-string-based approach. The dissertation studied two aggregation approaches for handling this distributed aggregation problem and contributes modifications to these approaches that yield good performance. To the best of our knowledge this is the first proposal for an expressive spatial algebra for distributed, in-network processing in WSNs.

- A task processing system that allows for the in-network evaluation of complex tasks that are algebraic expressions in the contributed algebra. The system is the first to allow exploratory interaction of spatial dynamically evolving phenomena in WSNs. In area such as environmental studies and precision agriculture this capability is a significant step towards the effort of realizing the vision of WSNs as macroscopes.

- A comparison of the in-network approach to the evaluation of spatial tasks with two out-of-network evaluation approaches with the results showing that the in-network approach results in substantial savings in terms of energy consumption and response time as well as, as expected, in message and bit complexity.

7.3 Future Research Directions

The contributions made in this dissertation give rise to several potential directions for future work.

Extensions to the Spatial Algebra. The dissertation presents distributed in-network algorithms for a spatial algebra enabling new geometries to be derived from induced and asserted ones and topological relationships between geometries to be identified. It would be beneficial to extend the spatial algebra to handle metric notions including *area*, *perimeter*, *diameter* of region, and *distance* between two spatial values etc. There exists research on the computation of the *area* [GKCE04] and the *perimeter* of a region [KOA09].

Setting Multiple Spatial-Analytic Task. The research contributions made in this dissertation assume the compilation of a single spatial-analytic task. It would be beneficial to add the ability to pose multiple spatial-analytical tasks against a WSN.

Integration with Existing WSN Query Engines. The current implementation of task processing suffers from the fact that it is not integrated with any existing WSN query processors. If, in the future, high-level, declarative approaches become the primary way in which users interact with WSNs, better integration would be beneficial. Current WSN query processors allow users to pose declarative queries but lack support for spatial analysis. The integration of spatial-analytic task processing with existing WSN query processors would provide users with a more comprehensive solution.

Spatial-Extension to Query Language. The work presented in this dissertation supports expressive algebraic abstractions as spatial types with expressive operations upon them. It would be beneficial for spatial extensions to be expressible at the level of a query language.

Multiple Gateways. Another limitation of the current task processing system is that all of the results for a particular task are delivered to a single end-point. It would be useful to extend the techniques proposed in this dissertation to support multiple gateways. Since, this would mitigate the risk of creating hotspots where time to depletion is much shorter.

Event and Edge Detection. Task processing system currently includes only basic support for event and edge detection. More sophisticated in-network event and edge detection techniques would likely to be more energy efficient, providing true edges and would enable handling faulty measurements.

Supporting 3D Spatial Analysis. The research presented in this dissertation works for the analysis in *two dimensions.* It would be useful to extend the techniques proposed in this dissertation to support 3D analysis. There exists related work on 3D spatial analysis [CS09], but, as is the norm, there has been no attempt at distributed evaluation over WSNs.

Extension to Handle Fuzzy Regions. In some applications in geographic analysis, spatial entities cannot be represented with crisp boundaries. Spatial values of such entities are called fuzzy. Examples includes land features with continuously changing properties (such as population density, soil quality, vegetation, pollution, air pressure), oceans, deserts, or mountains and valleys (e.g., the transition between a valley and a mountain usually cannot be exactly ascertained) [Sch01a, Sch01b]. Research has already been conducted in this area including how to model such types and to compute the topological relationships between such values [SDCCK08, Zha97, Sch01a, Sch01b], but, again, there has been no attempt at distributed evaluation over WSNs.

Temporal Extension and Need of Storage Manager. Our contributions enable spatial analysis in WSN. However, there are many environmental monitoring applications (such as those concerned with emergencies such as oil spills, or forest fires) where the addition of a temporal dimension to yield an in-network spatio-temporal task processing over WSNs would be beneficial. There exists work in the area of spatio-temporal query processing in WSN [HSLA05, CSN05]. For supporting in-network spatio-temporal queries it may be necessary to materialize the information related to induced geometries, in order to support the evaluation of historical queries. For supporting materialization, it would be necessary to store data persistently in sensor nodes. As sensor nodes are equipped with limited persistent storage, there would be a need that each node to be equipped with a sophisticated storage manager. There exists related research on storage managers [MDGS06, Bla10], but there is still a need to develop more sophisticated storage managers that can fulfil all the requirements for supporting historical spatio-temporal queries. In addition, there would be a need for extensions to the query language and algebra.

7.4 Summary

In this chapter, we summarized the work presented in this dissertation. We also outlined some future directions which can be explored to advance the research in spatial analysis in wireless sensor network.

Bibliography

[A05] Baggio A. Wireless sensor networks in precision agriculture. In *Workshop on Real-World Wireless Sensor Networks. REALWSN'05, Stockholm, Sweden*, 2005.

[ABC$^+$04] Tarek F. Abdelzaher, Brian M. Blum, Qing Cao, Y. Chen, D. Evans, J. George, S. George, Lin Gu, Tian He, Sudha Krishnamurthy, Liqian Luo, Sang Hyuk Son, Jack Stankovic, Radu Stoleru, and Anthony D. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS*, pages 582–589, 2004.

[AS09] C. Ayday and S. Safak. Application of wireless sensor networks with GIS on the soil moisture distribution mapping. In *Proceedings of Symposium GIS Seamless Geoinformation Technologies, Ostrava, Czech Republic*, 2528 January 2009.

[ASSC02a] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[ASSC02b] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102 – 114, August 2002.

[AY03] Kemal Akkaya and Mohamed Younis. An energy-aware QoS routing protocol for wireless sensor networks. *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, 0:710 – 715, 2003.

[AY05a] Kemal Akkaya and Mohamed Younis. Energy and QoS aware routing in wireless sensor networks. *Cluster Computing*, 8(2-3):179–188, 2005.

[AY05b] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3:325–349, 2005.

[BBB04] Jenna Burrell, Tim Brooke, and Richard Beckwith. Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive Computing*, 3:38–45, 2004.

[BDWL10] Abdelmalik Bachir, Mischa Dohler, Thomas Watteyne, and Kin Leung. MAC essentials for wireless sensor networks. *IEEE Communications Surveys & Tutorials*, 2010.

[BE02] D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *WSNA 02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 22–31. ACM Press, 2002.

[BGA+05] S. Boyd, Ghosh, A., B. Prabhakar, and D Shah. Gossip algorithms: design, analysis and applications. In *INFOCOM 2005: 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1653–1664, 2005.

[BGS00] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the physical world. *IEEE Journal of Selected Areas in Communications*, 7(5):10–15, October 2000.

[BGTD06] Kun Bi, Naijie Gu, Kun Tu, and Wanli Dong. Neighborhood-based distributed topological hole detection algorithm in sensor networks. *IET Conference Publications*, 2006(CP525):21–21, 2006.

[BHE00] Nirupama Bulusu, John Heidemann, and Deborah Estrin. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, October 2000.

[Big09] Jeff Bigham. ALERT: Automated local evaluation in real time. `http://www.alertsystems.org/`, 2009.

[Bla10] Rincon research corporation, blackbook. `http://tinyos.cvs.sourceforge.net/tinyos/tinyo-1.x/contrib/rincon/apps/Blackbook5/`, 2010.

[BMSU01] Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7:609–616, 2001.

[BTG+06] Kun Bi, Kun Tu, Naijie Gu, Wan Lin Dong, and Xiaohu Liu. Topological hole detection in sensor networks with cooperative neighbors. *proceedings of International Conference on Systems and Networks Communications (ICSNC'06), Tahiti*, 0:31–31, October 2006.

[CFC06] Shigang Chen, Guangbin Fan, and Jun-Hong Cui. Avoid "void" in geographic routing for data aggregation in sensor networks. *Int. J. Ad Hoc Ubiquitous Comput.*, 1(4):169–178, 2006.

[CG03] Krishna Chintalapudi and Ramesh Govindan. Localized edge detection in sensor fields. *Ad Hoc Networks*, 1(2-3):273–291, 2003.

[CHLS07] Caixia Chi, Dawei Huang, David Lee, and XiaoRong Sun. Lazy flooding: a new technique for information dissemination in distributed network systems. *IEEE/ACM Trans. Netw.*, 15:80–92, February 2007.

[CHS+09] Jiming Chen, Shibo He, Youxian Sun, Preetha Thulasiraman, and Xuemin (Sherman) Shen. Optimal flow control for utility-lifetime tradeoff in wireless sensor networks. *Comput. Netw.*, 53(18):3031–3041, 2009.

[CIE00] R. Govindan C. Intanagonwiwat and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 56–67, Boston, MA USA, 2000.

[CLJ06] Hongju Cheng, Qin Liu, and Xiaohua Jia. Heuristic algorithms for real-time data aggregation in wireless sensor networks. In *IWCMC*, pages 1123–1128, 2006.

[CM69] Coxeter and H. S. M. Barycentric coordinates. *Introduction to Geometry*, 2:216–221, 1969.

[CMS03] S. Cugati, W. Miller, and J. Schueller. Automation concepts for the variable rate fertilizer applicator for tree farming. In *The Proceedings of the 4th European Conference in Precision Agriculture, Berlin, Germany*, 2003.

[CNS04] Alexandru Coman, Mario A. Nascimento, and Jörg Sander. A framework for spatio-temporal query processing over wireless sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 104–110, New York, NY, USA, 2004. ACM.

[CPX05] Jen-Yeu Chen, Gopal Pandurangan, and Dongyan Xu. Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[cro] Crossbow technologies. `http://www.crossbow.com`.

[cro10a] Crossbow technology. imote2 high performance wireless sensor network node datasheet. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf`, 2010.

[cro10b] Wireless measurement system. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/mica2_Datasheet.pdf`, 2010.

[CS09] Tao Chen and Markus Schneider. Data structures and intersection algorithms for 3d spatial data types. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 148–157, New York, NY, USA, 2009. ACM.

[CSA04] Arnab Chakrabarti, Ashutosh Sabharwal, and Behnaam Aazhang. Multi-hop communication is order-optimal for homogeneous sensor networks. In *IPSN*, pages 178–185, 2004.

[CSN05] Alexandru Coman, Jorg Sander, and Mario A. Nascimento. An analysis of spatio-temporal query processing in sensor networks. In *Proceedings of the 21st International Conference on Data Engineering Workshops*, ICDEW '05, pages 1190–, Washington, DC, USA, 2005. IEEE Computer Society.

[CT04] Jae-Hwan Chang and L. Tassiulas. Maximum lifetime routing in wireless sensor networks. *Networking, IEEE/ACM Transactions on*, 12(4):609 – 619, August 2004.

[CTC06] Tzung-Shi Chen, Hua-Wen Tsai, and Chih-Ping Chu. Gathering-load-balanced tree protocol for wireless sensor networks. In *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing - Vol 2 - Workshops*, pages 8–13, Washington, DC, USA, 2006. IEEE Computer Society.

[DBN03] Budhaditya Deb, Sudeept Bhatnagar, and Badri Nath. Stream: Sensor topology retrieval at multiple resolutions. *Kluwer Journal of Telecommunications Systems*, 26:285–320, 2003.

[DCXC05] M. Ding, D. Chen, K. Xing, and X. Cheng. Localized fault-tolerant event boundary detection in sensor networks. In *INFOCOM*, pages 902–913, 2005.

[DDHG05] Jitender S. Deogun, Saket Das, Haitham S. Hamza, and Steve Goddard. An algorithm for boundary discovery in wireless sensor networks. In *HiPC*, pages 343–352, 2005.

[DFN06] Wenliang Du, Lei Fang, and Peng Ning. LAD: localization anomaly detection for wireless sensor networks. *J. Parallel Distrib. Comput.*, 66(7):874–886, 2006.

[DHS02] Stefan Dulman, Paul Havinga, and Faculty Of Computer Sciences. Wave leader election protocol for wireless sensor networks. In *Proceedings of the 3rd International Symposium on Mobile Multimedia Systems & Applications*, pages 43–50, 2002.

[DK06] Tassos Dimitriou and Ioannis Krontiris. Gravity: Geographic routing around voids in sensor networks. *International Journal of Pervasive Computing and Communications*, 2:351–361, 2006.

[DKSD09] Antonios Deligiannakis, Yannis Kotidis, Vassilis Stoumpos, and Alex Delis. Building efficient aggregation trees for sensor network event-monitoring queries. In *GSN '09: Proceedings of the 3rd International Conference on GeoSensor Networks*, pages 63–76, Berlin, Heidelberg, 2009. Springer-Verlag.

[DLJ$^+$05] Michael Demmer, Philip Levis, August Joki, Eric Brewer, and David Culler. Tython: A dynamic simulation environment for sensor networks. Technical report, UC Berkeley, 2005.

[EE01] Jeremy Elson and Deborah Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 186–, Washington, DC, USA, 2001. IEEE Computer Society.

[EG02] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, New York, NY, USA, 2002. ACM.

[FZWN08] Christopher Farah, Cheng Zhong, Michael F. Worboys, and Silvia Nittel. Detecting topological change using a wireless sensor network. In *GIScience*, pages 55–69, 2008.

[GBJ$^+$09] Ixent Galpin, Christian Y. A. Brenninkmeijer, Farhana Jabeen, Alvaro A. A. Fernandes, and Norman W. Paton. Comprehensive optimization of declarative sensor network queries. In *SSDBM*, pages 339–360, 2009.

[GFP$^+$01] Tony Griffiths, Alvaro A. A. Fernandes, Norman W. Paton, Keith T. Mason, Bo Huang, Michael F. Worboys, Chris Johnson, and John G. Stell. Tripod: A comprehensive system for the management of spatial and aspatial historical objects. In *ACM-GIS*, pages 118–123, 2001.

[GKCE04] Ben Greenstein, Eddie Kohler, David Culler, and Deborah Estrin. Distributed techniques for area computation in sensor networks. *Local Computer Networks, Annual IEEE Conference on*, 0:533–541, 2004.

[GM95] Leonidas J. Guibas and David H. Marimont. Rounding arrangements dynamically. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 190–199, New York, NY, USA, 1995. ACM.

[GM04] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3:46–55, 2004.

[GS93] Ralf Hartmut Güting and Markus Schneider. Realms: A foundation for spatial data types in database systems. In David J. Abel and Beng Chin Ooi, editors, *Advances in Spatial Databases, Third International Symposium, SSD'93, Singapore, June 23-25, 1993, Proceedings*, volume 692 of *Lecture Notes in Computer Science*, pages 14–35. Springer, 1993.

[GS95] Ralf Hartmut Gting and Markus Schneider. Realm-based spatial data types: The rose algebra. *The VLDB Journal*, 4:243–286, 1995. 10.1007/BF01237921.

[GY86] Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:143–152, 1986.

[HAG$^+$07] Thomas C. Harmon, Richard F. Ambrose, Robert M. Gilbert, Jason C. Fisher, Michael Stealey, and William J. Kaiser. High-Resolution River Hydraulic and Water Quality Characterziation using Rapidly Deployable Networked Infomechanical Syestems (NIMS RD). *Environmental Engineering Science*, 24(2):151–1459, 2007.

[HBC$^+$09] Wen Hu, Nirupama Bulusu, Chun Tung Chou, Sanjay Jha, Andrew Taylor, and Van Nghia Tran. Design and evaluation of a hybrid sensor network for cane toad monitoring. *ACM Trans. Sen. Netw.*, 5:4:1–4:28, February 2009.

[HF08] Xiaoxia Huang and Yuguang Fang. Multiconstrained QoS multipath routing in wireless sensor networks. *Wirel. Netw.*, 14(4):465–478, 2008.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

[HHT02] Matthias Handy, Marc Haase, and Dirk Timmermann. Low energy adaptive clustering hierarchy with deterministic cluster-head selection. In *IEEE MWCN*, pages 368–372. IEEE Computer Society, 2002.

[HKB99] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185, Seattle, WA USA, 1999.

[HL05] Gregory Hartl and Baochun Li. infer: A bayesian inference approach towards energy efficient data collection in dense sensor networks. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 371–380, Washington, DC, USA, 2005. IEEE Computer Society.

[HM06] Jane K. Hart and Kirk Martinez. Environmental Sensor Networks: A revolution in the earth system science? *Earth-Science Reviews*, 78:177–191, 2006.

[HSLA05] Tian He, John A. Stankovic, Chenyang Lu, and Tarek F. Abdelzaher. A spatiotemporal communication protocol for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 16:995–1006, 2005.

[HYS04] F. Ye H. Yang and B. Sikdar. A dynamic query-tree energy balancing protocol for sensor networks. In *Proceedings of IEEE WCNC, Atlanta, GA*, 2004.

[HYW$^+$06] John Heidemann, Wei Ye, Jack Wills, Affan Syed, and Yuan Li. Research challenges and applications for underwater sensor networking. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, 2006.

[IEGH02] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 457–, Washington, DC, USA, 2002. IEEE Computer Society.

[JA07] Inwhee Joe and Taewon Ahn. An efficient bi-directional routing protocol for wireless sensor networks. In *Proceedings of the 2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking&Services (MobiQuitous)*, pages 1–4, Washington, DC, USA, 2007. IEEE Computer Society.

[JF08] Farhana Jabeen and Alvaro A. A. Fernandes. Impact on accuracy of deployment trade-offs in localized sensor network event detection. In *Second International Workshop on Localized Algorithms and Protocols for Wireless Sensor Networks (LOCALGOS), in conjuction with IEEE DCOSS*, 2008.

[JJJES07] Adil Jaffer, Muhammad Jaseemuddin, Mandana Jafarian, and Hesham El-Sayed. Event boundary detection using autonomous agents in a sensor network. In *AICCSA*, pages 217–224, 2007.

[JJPSW+09] Zhang Jin, Yu Jian-Ping, Zhou Si-Wang, Lin Ya-Ping, and Li Guang. A survey on position-based routing algorithms in wireless sensor networks. *Algorithms*, 2(1):158–182, 2009.

[JK09] Gyanendra Prasad Joshi and Sung Won Kim. A distributed geo-routing algorithm for wireless sensor networks. *Sensors*, 9(6):4083–4103, 2009.

[JLY+08] Sangsu Jung, Dujeong Lee, Sangyoon Yoon, Jaehwi Shin, Youngwoo Lee, and Jeonghoon Mo. A geographic routing protocol utilizing link lifetime and power control for mobile ad hoc networks. In *FOWANC '08: Proceeding of the 1st ACM international workshop on Foundations of wireless ad hoc and sensor networking and computing*, pages 25–32, New York, NY, USA, 2008. ACM.

[JN06] Guang Jin and Silvia Nittel. NED: An efficient noise-tolerant event and event boundary detection algorithm in wireless sensor networks. In *MDM*, page 153, 2006.

[JW08] Jixiang Jiang and Michael F. Worboys. Detecting basic topological changes in sensor networks by local aggregation. In *GIS*, page 4, 2008.

[JW09] Jixiang Jiang and Michael Worboys. Event-based topology for dynamic planar areal objects. *Int. J. Geogr. Inf. Sci.*, 23(1):33–60, 2009.

[JWN09] Jixiang Jiang, Michael Worboys, and Silvia Nittel. Qualitative change detection using sensor networks based on connectivity information. *GeoInformatica*, pages 1–24, 2009. 10.1007/s10707-009-0097-0.

[JWZ+09] Zhang Junguo, Li Wenbin, Yin Zhongxing, Liu Shengbo, and Guo Xiaolin. Forest fire detection system based on wireless sensor network. In *Industrial Electronics and Applications, 2009. ICIEA 2009*, pages 520–523, 2009.

[KDG03] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 482–, Washington, DC, USA, 2003. IEEE Computer Society.

[KI03] Bhaskar Krishnamachari and S. Sitharama Iyengar. Efficient and fault-tolerant feature extraction in wireless sensor networks. In *Proceedings of the 2nd international conference on Information processing in sensor networks*, IPSN'03, pages 488–501, Berlin, Heidelberg, 2003. Springer-Verlag.

[KI04] Bhaskar Krishnamachari and Sitharama Iyengar. Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks. *IEEE Trans. Comput.*, 53(3):241–250, 2004.

[KK00] Brad Karp and H. T. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 243–254, New York, NY, USA, 2000. ACM.

[KK03] Bradley Koch and Rajiv Khosla. The role of precision agriculture in cropping systems. *Journal of Crop Production*, 9(1/2 (17/18)):361–381, 2003.

[KLS$^+$10] Maria Kazandjieva, Jung Woo Lee, Marcel Salath, Marcus W. Feldman, James H. Jones, and Philip Levis. Experiences in measuring a human contact network for epidemiology research. In *ACM Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets)*, 2010.

[KM07] Kurtis Kredo, II and Prasant Mohapatra. Medium access control in wireless sensor networks. *Comput. Netw.*, 51(4):961–994, 2007.

[KOA09] Ahmed M. Khedr, Walid Osamy, and Dharma P. Agrawal. Perimeter discovery in wireless sensor networks. *Journal of Parallel and Distributed Computing*, 69(11):922 – 929, 2009.

[KPSK09] Young Il Ko, Chang-Sup Park, In Chul Song, and Myoung Ho Kim. An efficient void resolution method for geographic routing in wireless sensor networks. *J. Syst. Softw.*, 82(6):963–973, 2009.

[KR04] Cornelia Kappler and Georg Riegel. A real-world, simple wireless sensor network for monitoring electrical energy consumption. In *EWSN*, pages 339–352, 2004.

[LCN$^+$05] Jie Lian, Lei Chen, Kshirasagar Naik, M. Tamer Özsu, and G. Agnew. Localized routing trees for query processing in sensor networks. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 259–260, New York, NY, USA, 2005. ACM.

[LG09] Philip Levis and David Gay. *TinyOS Programming.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[LKH05] Chun-Han Lin, Chung-Ta King, and Hung-Chang Hsiao. Region abstraction for event tracking in wireless sensor networks. In *ISPAN*, pages 274–281, 2005.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David E. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys*, pages 126–137, 2003.

[LMG$^+$04] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer, and David E. Culler. The emergence of networking abstractions and techniques in TinyOS. In *NSDI*, pages 1–14, 2004.

[LRS02] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M. Sivalingam. Data gathering algorithms in sensor networks using energy metrics. *IEEE Trans. Parallel Distrib. Syst.*, 13(9):924–935, 2002.

[LWG05] O. Landsiedel, K. Wehrle, and S. Gotz. Accurate prediction of power consumption in sensor networks. In *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 37–44, Washington, DC, USA, 2005. IEEE Computer Society.

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[MDGS06] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 195–208, New York, NY, USA, 2006. ACM.

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[MFHH05] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[MGZ+09] A. Matese, S.F. Di Gennaro, A. Zaldei, L. Genesio, and F.P. Vaccari. A wireless sensor network for precision viticulture: The NAV system. *Computers and Electronics in Agriculture*, 69(1):51 – 58, 2009.

[MM06] Umberto Malesci and Samuel Madden. A measurement-based analysis of the interaction between network layers in TinyOS. In *EWSN*, pages 292–309, 2006.

[MP06] Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *DCOSS*, pages 150–168, 2006.

[MV05] Matthew J. Miller and Nitin H. Vaidya. A MAC protocol to reduce sensor network energy consumption using a wakeup radio. *IEEE Transactions on Mobile Computing*, 4(3):228–242, 2005.

[NM03] Robert Nowak and Urbashi Mitra. Boundary estimation in sensor networks: theory and methods. In *Proceedings of the 2nd international conference on Information processing in sensor networks*, IPSN'03, pages 80–95, Berlin, Heidelberg, 2003. Springer-Verlag.

[NN03] Dragos Niculescu and Badri Nath. Trajectory based forwarding and its applications. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 260–272, New York, NY, USA, 2003. ACM.

[OE08] Susan A O'Shaughnessy and Steven R Evett. Integration of wireless sensor networks into moving irrigation systems for automatic irrigation scheduling. In *American Society of Agricultural and Biological Engineers Annual International Meeting*, volume 1, pages 464–484, 2008.

[PBM$^+$05] Sofie Pollin, Bruno Bougard, Rahul Mangharam, Francky Catthoor, Ingrid Moerman, Ragunathan Rajkumar, and Liesbet Van der Perre. Optimizing transmission and shutdown for energy-efficient real-time packet scheduling in clustered ad hoc networks. *EURASIP J. Wirel. Commun. Netw.*, 2005:698–711, October 2005.

[PE08] F. J. Pierce and T. V. Elliott. Regional and on-farm wireless sensor networks for agricultural systems in eastern washington. *Comput. Electron. Agric.*, 61(1):32–43, 2008.

[PK00] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[PKRVJ05] A V U Phani Kumar, Adi Mallikarjuna Reddy V, and D. Janakiram. Distributed collaboration for event detection in wireless sensor networks. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8, New York, NY, USA, 2005. ACM.

[PR00] Romulo G. Pizana and Roland E. Ramos. Triangle graphs with maximum degree at most 3. In *Proceedings of the Third Asian Mathematical Conference*, pages 451–454, 2000.

[RSZ04] C. S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati, editors. *Wireless sensor networks*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[RZL06] Kui Ren, Kai Zeng, and Wenjing Lou. Fault-tolerant event boundary detection in wireless sensor networks. In *GLOBECOM '06: Proceedings of the Global Telecommunications Conference*, pages 1 –5, 2006.

[San06] Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2006.

[SBAS04] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, New York, NY, USA, 2004. ACM.

[SBLC04] Mohamed A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, and Panos K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *VLDB J.*, 13(4):384–403, 2004.

[Sch97] Markus Schneider. *Spatial Data Types for Database Systems: Finite Resolution Geometry for Geographic Information Systems*, volume 1288 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997.

[Sch01a] Markus Schneider. A design of topological predicates for complex crisp and fuzzy regions. In *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, pages 103–116, London, UK, 2001. Springer-Verlag.

[Sch01b] Markus Schneider. Fuzzy topological predicates, their properties, and their integration into query languages. In *GIS '01: Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, pages 9–14, New York, NY, USA, 2001. ACM.

[SDCCK08] Steven Schockaert, Martine De Cock, Chris Cornelis, and Etienne E. Kerre. Fuzzy region connection calculus: Representing vague topological information. *Int. J. Approx. Reasoning*, 48(1):314–331, 2008.

[Sel06] S. Selvakennedy. An Energy Efficient Event Processing Algorithm for Wireless Sensor Networks. In *MSN*, pages 588–599, 2006.

[SHrC$^+$04] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys*, pages 188–200, 2004.

[SHS04] Radu Stoleru, Tian He, and John A. Stankovic. Walking GPS: A practical solution for localization in manually deployed wireless sensor networks. In *LCN*, pages 480–489, 2004.

[SKG05] Amir Soheili, Vana Kalogeraki, and Dimitrios Gunopulos. Spatial queries in sensor networks. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 61–70, New York, NY, USA, 2005. ACM.

[SKPM06] Kee-Young Shin, Jin Won Kim, Ilgon Park, and Pyeong Soo Mah. Wireless sensor networks: A scalable time synchronization. In *ICCSA (4)*, pages 509–518, 2006.

[SL01] Ivan Stojmenovic and Xu Lin. Loop-free hybrid single-path/flooding routing algorithms with guaranteed delivery for wireless networks. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1023–1032, 2001.

[SN05] Y. Shiraishi S. Nittel, G. Jin. In-networks spatial query estimation in sensor networks. In *IEICE Transactions (A), Vol.J88-A, No.12*, pages pp.1413–1421, 2005.

[Sri] Prasanna Sridhar. Scalability and performance issues in deeply embedded sensor systems. *Smart Sensing And Intelligent Systems*, 2(1):1–14.

[SSC$^+$07] Amarjeet Singh, Michael J. Stealey, Victor Chen, William J. Kaiser, Maxim Batalin, Yeung Lam, Bin Zhang, Amit Dhariwal, Carl Oberg, Arvind Pereira, Gaurav S. Sukhatme, Beth Stauffer, Stefanie Moorthi, Dave Caron, and Mark Hansen. Human Assists Robotic Team Campaigns for Aquatic Monitoring. *Journal of Field Robotics*, 24(11):969–989, 2007.

[Sto02] I. Stojmenovic. Position-based routing in ad hoc networks. *Communications Magazine, IEEE*, 40(7):128 –134, jul 2002.

[SZHK04] Karim Seada, Marco Zuniga, Ahmed Helmy, and Bhaskar Krishnamachari. Energy-efficient forwarding strategies for geographic routing in lossy wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 108–121, New York, NY, USA, 2004. ACM.

[Tin] Simulating tinyos applications in tossim. `http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson5.html`.

[TK84] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 32(3):246–257, 1984.

[TL08] Athanasia Tsertou and David I. Laurenson. Revisiting the hidden terminal problem in a csma/ca wireless network. *IEEE Trans. Mob. Comput.*, 7(7):817–831, 2008.

[TPS$^+$05] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David E. Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *SenSys*, pages 51–63, 2005.

[TYIM05] Wataru Tsujitaa, Akihito Yoshinoa, Hiroshi Ishidab, and Toyosaka Moriizumia. Gas sensor network for air-pollution monitoring. *Sensors and Actuators B: Chemical*, 110(2):304–311, October 2005.

[Ulr08] Thomas Ulrich. Wireless network monitors H2O: System saves resources, increases yield in cabernet vineyard. *Wines and Vines Magazine*, July 2008.

[VTP$^+$08] G. Vellidis, M. Tucker, C. Perry, C. Kvien, and C. Bednarz. A real-time wireless smart sensor array for scheduling irrigation. *Comput. Electron. Agric.*, 61(1):44–50, 2008.

[WALW$^+$06] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[WC01] Alec Woo and David E. Culler. A transmission control scheme for media access in sensor networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 221–235, New York, NY, USA, 2001. ACM.

[WD06] Michael F. Worboys and Matt Duckham. Monitoring qualitative spatiotemporal change for geosensor networks. *International Journal of Geographical Information Science*, 20(10):1087–1108, 2006.

[WEC03] Chieh Y. Wan, Shane B. Eisenman, and Andrew T. Campbell. Coda: congestion detection and avoidance in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 266–279, New York, NY, USA, 2003. ACM.

[Wel04] Matt Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. *Computer Communication Review*, 34(1):119–124, 2004.

[WMHF03] Jörg Widmer, Martin Mauve, Hannes Hartenstein, and Holger Fü*β*ler. Position-based routing in ad hoc wireless networks. *The handbook of ad hoc wireless networks*, pages 219–232, 2003.

[WZW06] Ning Wang, Naiqian Zhang, and Maohua Wang. Review: Wireless sensors in agriculture and food industry-recent development and future perspective. *Comput. Electron. Agric.*, 50(1):1–14, 2006.

[XcL05] Yingqi Xu and Wang chien Lee. PSGR: priority-based stateless geo-routing in wireless sensor networks. In *Proc. IEEE Conf. Mobile Ad-hoc and Sensor Systems*, pages 7–10. IEEE Press, 2005.

[YCL$^+$01] Fan Ye, Alvin Chen, Songwu Lu, Lixia Zhang, and Fan Ye Alvin Chen. A scalable solution to minimum cost forwarding in large sensor networks. In *IEEE International Conference on Computer Communications and Networks (ICCCN), Scottsdale, Arizona, USA*, pages 304–309, 2001.

[YF04] Ossama Younis and Sonia Fahmy. Heed: A hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing*, 3:366–379, 2004.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[YGE01] Y. Yu, R. Govindan, and D. Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report UCLA/CSD-TR-01-0023, UCLA Computer Science Department, 2001.

[YH03] Wei Ye and John Heidemann. Medium access control in wireless sensor networks, 2003.

[ZGE03] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *First IEEE International Workshop on Sensor Network Protocols and Applications*, page 139148, 2003.

[ZGFL08] Ming Zhang, Chenglong Gong, Yuan Feng, and Chao Liu. Energy-predicted shortest routing tree algorithm in wireless sensor networks. In *ISICA '08: Proceedings of the 3rd International Symposium on Advances in Computation and Intelligence*, pages 117–124, Berlin, Heidelberg, 2008. Springer-Verlag.

[ZGTS03] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Inf. Syst.*, 28(1-2):61–84, 2003.

[Zha97] F. Benjamin Zhan. Topological relations between fuzzy regions. In *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing*, pages 192–196, New York, NY, USA, 1997. ACM.

[ZLS07] Gang Zhao, Xiangqian Liu, and Min-Te Sun. Anchor-based geographic routing for sensor networks using projection distance. In *Wireless Pervasive Computing, 2007. ISWPC '07. 2nd International Symposium on*, 5-7 2007.

[ZRLM06] Kai Zeng, Kui Ren, Wenjing Lou, and Patrick J. Moran. Energy-aware geographic routing in lossy wireless sensor networks with environmental energy supply. In *Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks*, QShine '06, New York, NY, USA, 2006. ACM.

[ZSGM08] Xianjin Zhu, Rik Sarkar, Jie Gao, and Joseph S. B. Mitchell. Light-weight contour tracking in wireless sensor networks. In *INFOCOM*, pages 1175–1183, 2008.

[ZVPS08] Yujie Zhu, Ramanuja Vedantham, Seung-Jong Park, and Raghupathy Sivakumar. A scalable correlation aware aggregation strategy for wireless sensor networks. *Inf. Fusion*, 9(3):354–369, 2008.

[ZYACS07] Demetrios Zeinalipour-Yazti, Panayiotis Andreou, Panos K. Chrysanthis, and George Samaras. Senseswarm: a perimeter-based data acquisition framework for mobile sensor networks. In *DMSN '07: Proceedings of the 4th workshop on Data management for sensor networks*, pages 13–18, New York, NY, USA, 2007. ACM.

Appendix A

Rose Algebra

The Appendix is structured as follows. Section A.1 defines the data types offered by the ROSE algebra. Section A.2 describes the definitions of the spatial operations.

A.1 Spatial Data Types

The ROSE algebra offers three data types called **points**, **lines**, and **regions**, whose values are *realm*-based. To describe these values, the notions of an *R_block*, *R_unit*, *R_cycle* and *R_face* are introduced.

A **points** data type value is a finite set of *R_points*. For **lines** two representations have been defined, one as a finite set of *R_segments* and another as a finite set of pairwise disjoint *R_blocks*. An *R_block* is a set of maximal connected *R_segments*. Two *R_blocks* $b1$ and $b2$ are `disjoint`: $\Leftrightarrow$ $\forall$ $s_1 \in S(b_1)$ $\forall$ $s_2 \in S(b_2)$: s_1 and s_2 are `disjoint`, where $S(b1)$ refers to the segments in block $b1$ and $S(b2)$ to the segments in $b2$.

An *R_cycle* is a simple polygon and denotes a cycle in the planar graph interpretation of a *realm*. Cycles represent basic entities for the definitions of **regions**. A value of type **regions** is a finite set of `edge_disjoint` *R_faces*. By `edge_disjoint` is meant that the values are `area_disjoint` (i.e. the interior of either **regions** value does not intersect the other **regions** value) and have no common boundary segment. A value f of type *R_face* is represented as a pair (c, H) where c is an *R_cycle* as the outer boundary and H is a (possibly empty) set of holes $\{h^1, ..., h^n\}$. An *R_face* is an *R_cycle* that possibly encloses other `edge_disjoint` *R_cycles*, which correspond to holes (see the example in Figure A.1 taken from [Sch97]). The holes, if they exist, are required to be `edge_inside` the outer *R_cycle*. By `edge_inside` is meant that the hole lies inside the outer *R_cycle* and that they should not have any common boundary segment. The reader is referred to Section A.2.1 for the description of the `edge_inside` and `edge_disjoint` operators. Each of the cycles in the set of segments of f (i.e., $S(f)$) is either equal to c or to one of the cycles in H. *R_units* are the smallest **regions** values that exist over a *realm*. They are also called minimal *R_faces*. A **regions** value can, therefore, be viewed as set of *R_faces* or, equivalently, as set of *R_units*. The boundary and the exterior of a **regions** value with holes is allowed to be disconnected.

An *R_cycle* c divides grid points into three subsets $P_{in}(c)$, $P_{on}(c)$, and $P_{out}(c)$, where $P_{in}(c)$ denotes the set of points lying in the interior of c; $P_{on}(c)$, the set of points lying on the outer boundary of the c; and $P_{out}(c)$, the set of points lying outside the c. The set of points that are part of a *R_cycle* c can, therefore, be defined as $P(c) := P_{on}(c) \cup P_{in}(c)$. The grid points that belong to an *R_face* $f = (f_0, \overline{F})$ (where f_0 is an *R_cycle* as the outer boundary and $\overline{F}$ is a (possibly empty) set of holes $\{f^1, ..., f^n\}$) is defined as $P(f) := P(f_0)/ \bigcup_{i=1}^{n} P_{in}\ (\overline{f}_i)$. Figure A.1 illustrates some example values of type **points**, **lines** and **regions**.

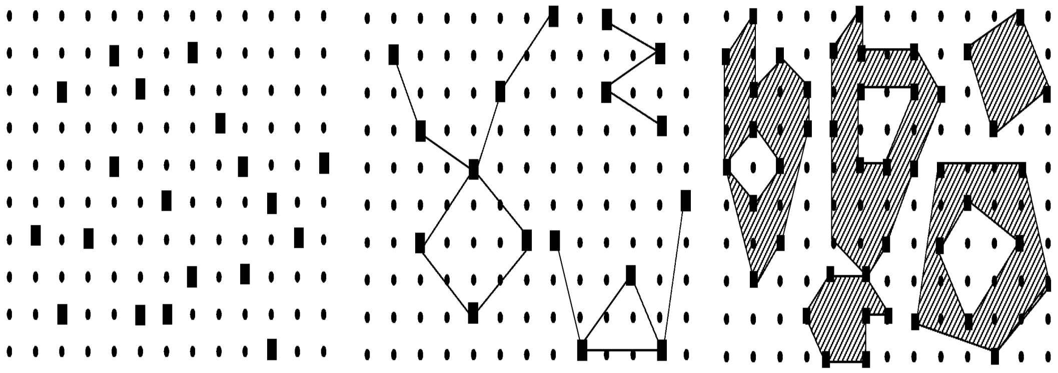


Figure A.1: (a) **points** value (b) **lines** value (c) **regions** value adapted from [Sch97]

A.2 Spatial Operations

This section describes the two types of operation in the ROSE algebra that are of greatest relevance in this dissertation, viz., spatial predicates (i.e., Boolean-valued operations that test topological relationships) and spatial-valued operations.

A.2.1 Spatial Predicates expressing Topological Relationships

Spatial predicates test whether two spatial values stand in the topological relationship denoted by the predicate. The result is therefore a Boolean value. Topological relationships include `equals`, `not_equals`, `edge_disjoint`, `vertex_disjoint`.

Schneider [Sch97] has provided definitions for topological relationships between combinations of spatial data types. This section discusses topological relationships between pairs of spatial data types such as *R_cycle*, *R_block*, *R_face*, and **regions**.

This section will first discuss the topological relationships between a pair of values of type *R_cycle*. Some of the relationships between two *R_cycles* are given in Table A.1. Informally, two *R_cycles* $c1$ and $c2$ are `area_disjoint` if they do not share an interior. The relationship is `edge_disjoint` if they do not share an interior or any boundary segment but their boundary may touch. Two *R_cycle* values are `vertex_disjoint` if they do not share any point. Similarly, $c1$ is `area_inside` $c2$ if $c1$ is a proper subset of $c2$ or $c1$ is equal to $c2$; $c1$ is `edge_inside` $c2$ if $c1$ is a proper subset of $c2$ and they do not share any boundary segment but their boundary may touch at any point on the boundary; $c1$ is `vertex_inside` $c2$ if $c1$ is a proper subset of $c2$ and they do

c1 `area_inside` c2	:⇔	P(c1) $\subseteq$ P(c2)
c1 `edge_inside` c2	:⇔	c1 `area_inside` c2 $\wedge$ S(c1)$\cap$ S(c2) $= \emptyset$
c1 `vertex_inside` c2	:⇔	c1 `edge_inside` c2 $\wedge$ P$_{on}$(c1)$\cap$ P$_{on}$(c2) $= \emptyset$
c1 `area_disjoint` c2	:⇔	P$_{in}$(c1) $\cap$ P(c2)$= \emptyset \wedge$ P$_{in}$(c2)$\cap$ P(c1) $= \emptyset$
c1 `edge_disjoint` c2	:⇔	c1 and c2 are `area_disjoint` $\wedge$ S(c1) $\cap$ S(c2) $= \emptyset$
c1 `vertex_disjoint` c2	:⇔	c1 and c2 are `edge_disjoint` $\wedge$ P$_{on}$(c1)$\cap$ P$_{on}$(c2) $= \emptyset$
c1 `adjacent` c2	:⇔	c1 and c2 are `area_disjoint` $\wedge$ S(c1)$\cap$ S(c2) $\neq \emptyset$
c1 `meets` c2	:⇔	c1 and c2 are `edge_disjoint` $\wedge$ P$_{on}$(c1) $\cap$ P$_{on}$(c2) $\neq \emptyset$

Table A.1: Predicates formally defined over two *R_cycles* c1 and c2 adapted from [Sch97]

$b1$ `meets` $b2$	:⇔	$\exists s \in S(b1)\, \exists t \in S(b2)$: s and t `meets` in a meeting point $\wedge$ $(\forall s \in S(b1)\, \forall t \in S(b2) : s \neq t) \wedge$ (s and t `meets` in p $\Rightarrow$ p is a meeting point)
$b1$ `intersects` $b2$	:⇔	$\forall s \in S(b1)\, \forall t \in S(b2) : s \neq t \wedge$ $(\exists s \in S(b1)\, \exists t \in S(b2) :) \wedge$ (s and t `meets` in p $\wedge$ p is a meeting point)

Table A.2: Formal definitions for spatial predicates on values b1 and b2 of type *R_blocks* [Sch97]

not share boundary point; $c1$ `meets` $c2$, if their boundaries share points but not segments and if their interiors are disjoint; $c1$ is `adjacent` to $c2$ if they share one or more boundary segment and their interiors are `disjoint`. In the ROSE algebra, `area_inside` is the default interpretation of the relationship `inside`, and `vertex_disjoint` is the default for the `disjoint` relationship.

As described in Section A.1, a value of type **lines** can be understood as a set of *R_segments* or as a set of pairwise disjoint *R_blocks*. Table A.2 describes some primitives describing two relationship between *R_blocks* $b1$ and $b2$.

Let p be a *R_point* where the two blocks $b1$ and $b2$ touch. For determining whether p is a meeting point, an angular sorted cyclic list L_p of *R_segments* belonging to any *R_block* value meeting at p is constructed. p is called a meeting point if L_p is the concatenation of two sublists, so that all elements of one of the sublists belong to $S(b1)$ and all elements of the other sublist belong to $S(b2)$. Figure A.2 illustrates that p is a meeting point but p' is not a meeting point.

The definitions of `area_inside` and `area_disjoint` given in Table A.1 are extended in Table A.3 for the computation of relationships between two *R_faces* $f = (f_0, \overline{F})$ and $g = (g_0, \overline{G})$. f is `area_inside` g if f_0 is `area_inside` g_0 and f_0 is `area_disjoint` from all holes in $\overline{G}$, or there exist holes in $\overline{F}$ to which the holes in $\overline{G}$ lie `area_inside` in it. f is `area_disjoint` g if f_0 and g_0 are `area_disjoint`, or if there exists any hole in $\overline{G}$ such that f_0 lies `area_inside` in it, or if there exists any hole in $\overline{F}$ such that g_0 lies `area_inside` in it.

As described in Section A.1, a **regions** value may consist of a set of *R_units* or a set of pairwise `edge_disjoint` *R_faces*. Let F and G be two values of type **regions**. Examples of the

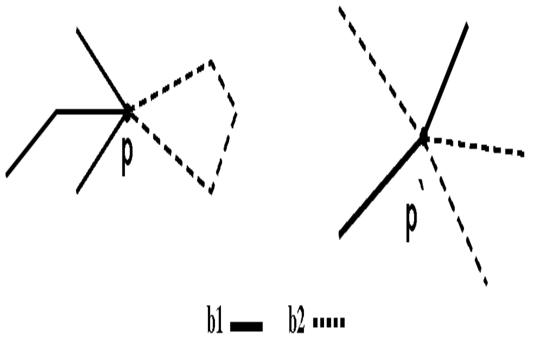


Figure A.2: Between two values of type *R_block* (i) p represents a meeting point (ii) $p\prime$ represents a non-meeting point [Sch97]

f `area_inside` g	$:\Leftrightarrow$	f_0 `area_inside` $g_0 \wedge$ $\forall\ \overline{g} \in \overline{G}$: $\overline{g}$ `area_disjoint` $f_0 \vee \exists\ \overline{f} \in \overline{F}$: $\overline{g}$ `area_inside` $\overline{f}$
f `area_disjoint` g	$:\Leftrightarrow$	f_0 `area_disjoint` $g_0 \vee$ $\exists\ \overline{g} \in \overline{G}$: f_0 `area_inside` $\overline{g} \vee \exists\ \overline{f} \in \overline{F}$: g_0 `area_inside` $\overline{f}$

Table A.3: Formal definitions for spatial predicates on values f and g of type *R_faces*

formal definitions for some of the operations when the arguments are of type **regions** are given in Table A.4 [Sch97]. F is `vertex_disjoint` G if all the elements of F have a `vertex_disjoint` relationship with the elements of G. For membership in an `area_inside` relationship, all elements of F must lie in one or more elements of G. F is `adjacent` to G, if the elements of both values are `area_disjoint` and there exists an element of G that is `adjacent` to an element of H.

The set of topological operators that are supported by, ROSE algebra over the spatial data types **points**, **lines**, and **regions** is shown in Table A.5. The ROSE algebra supports the combination of values of different spatial data types but maintains control over the operations that can be applied between values of spatial data types. The ROSE algebra type system allows polymorphic operations. To describe the operations in the algebra, second-order signatures [Sch97] have been introduced. A second-order signature is interpreted as defining a set of signatures formed by taking values from the set of types used to define it. Two such sets are EXT = {**lines**, **regions**} and GEO = {**points**, **lines**, **regions**}. This convention, makes it possible to define the `area_inside` predicate as operating on one value of type GEO and other

F `area_disjoint` G	$:\Leftrightarrow$	$\forall f \in F\, \forall g \in G$: f and g are `area_disjoint`
F `adjacent` G	$:\Leftrightarrow$	F `area_disjoint` $G \wedge\ \exists f \in F\, \exists g \in G$:f and g are `adjacent`
F `area_inside` G	$:\Leftrightarrow$	$\forall f \in F\, \exists g \in G$: f `area_inside` g
F `intersects` G	$:\Leftrightarrow$	$units(F) \cap units(G) \neq \emptyset$

Table A.4: Formal definitions for spatial predicates on values F and G of type **regions** [Sch97]

`equals`	: GEO $\times$ GEO	$\rightarrow$ Boolean
`not_equals`	: GEO $\times$ GEO	$\rightarrow$ Boolean
`intersects`	: $\text{EXT}_1 \times \text{EXT}_2$	$\rightarrow$ Boolean
`vertex_disjoint`	: GEO $\times$ GEO	$\rightarrow$ Boolean
`area_disjoint`	: **regions** $\times$ **regions**	$\rightarrow$ Boolean
`edge_disjoint`	: **regions** $\times$ **regions**	$\rightarrow$ Boolean
`adjacent`	: area $\times$ area	$\rightarrow$ Boolean
`meets`	: $\text{EXT}_1 \times \text{EXT}_2$	$\rightarrow$ Boolean
`area_inside`	: GEO $\times$ $\mathbb{R}$	$\rightarrow$ Boolean
`edge_inside`	: **regions** $\times$ **regions**	$\rightarrow$ Boolean
`vertex_inside`	: **regions** $\times$ **regions**	$\rightarrow$ Boolean
`border_in_common`	: $\text{EXT}_1 \times \text{EXT}_2$	$\rightarrow$ Boolean
`on_border_of`	: **points** $\times$ EXT	$\rightarrow$ Boolean

Table A.5: Spatial predicates [Sch97]

`intersection`	: **points** $\times$ **points**	$\rightarrow$	**points**
`intersection`	: **lines** $\times$ **lines**	$\rightarrow$	**points**
`intersection`	: **regions** $\times$ **regions**	$\rightarrow$	**regions**
`intersection`	: **regions** $\times$ **lines**	$\rightarrow$	**lines**
`contour`	: **regions**	$\rightarrow$	**lines**
`plus`	: GEO $\times$ GEO	$\rightarrow$	GEO
`minus`	: GEO $\times$ GEO	$\rightarrow$	GEO
`vertices`	: EXT	$\rightarrow$	**points**
`common_border`	: $\text{EXT}_1 \times \text{EXT}_2$	$\rightarrow$	**lines**
`interior`	: **lines**	$\rightarrow$	**regions**

Table A.6: Spatial-valued operations [Sch97]

value of type **regions**. If the operation is defined with a second-order signature of the form $T \times T$, then the implication is that the same element (e.g., from GEO or EXT) must be selected for each of the arguments. If the operation is defined by a second-order signature of the form $T_1 \times T_2$, i.e., one where the arguments are subscripted, then the implication is that different selections can be made for each of the arguments.

In the type system, *area* can be bound to any type in $\textbf{regions}^{\texttt{area\_disjoint}}$. The notation $\textbf{regions}^{\texttt{area\_disjoint}}$ is used to represent partitions (i.e. `area_disjoint` subdivisions of the plane into `area_disjoint` regions based on non-spatial attribute values). The type variable *area* guarantees that the operation `adjacent` will only be applicable to two **regions** value from the same partition.

A.2.2 Spatial Operations Returning Spatial Data Type Values

Operations such as *plus*, *intersection*, *minus*, *interior*, *contour*, *vertices*, *common_border* return a spatial data type value as result. Table A.6 lists the ROSE ***spatial-valued operations***.

Operations `plus` (union) and `minus` (difference) are more restricted in being applicable to values of the same data type. Operation `plus` returns the union of two values. In the case of operands of type **points** and **lines**, the operator `plus` merges each element of the operands into a new value having the same type as that of its operands.

For the computation of `plus`, `minus` and `intersection` with arguments of type **regions**, the

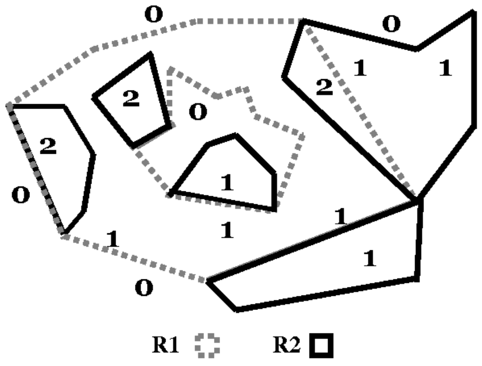


Figure A.3: Two intersecting **regions** Values [Sch97]

concepts of *overlap numbers* and *segment classification* have been introduced. The overlap number k assigned to *realm* grid points depends on the number of **regions** values it is part of. Figure A.3 illustrates the overlap number assigned to the areas belonging to two **regions** $R1$ and $R2$ that intersect one another. As each segment of a **regions** value divides the space into an interior and exterior part, each segment is identified by a pair (m/n) of overlap numbers, where m represents the lower (or the right) and n the upper (or the left) segment. Possible (m/n)-segments between two **regions** values are (0/1)-, (0/2)-, (1/0)-,(1/1)-, (1/2)-, (2/0)-, and (2/1)-segments. The computation of the `plus` of **regions** values $R1$ and $R2$ requires segments to be $(0/1)-, (1/0)-, (0/2)-$, and $(2/0)$.

Operation `minus` returns the difference between two values. In the case where both operands are of type **lines** or **points**, the `minus` operation determines all the elements that are present in the first operand and not in the second operand. The computation of the difference of two **regions** values $R1$ and $R2$ requires all (0/1) and (1/0)-segments of $R1$, all $(1/2)-$ and (2/1)-segments of $R2$, and all (1/1)-segments common to $R1$ and $R2$.

The `intersection` operation comes in several forms. The `intersection` between two values of type **lines** determines all their common points that are not meeting points and returns a value of type **points** as a result. The `intersection` of a **regions** value with a **lines** value returns a **lines** value as a result. The `intersection` of two values of type **regions** must be always a **regions** value. In the case of one operand of type **lines** and an other of type **regions**, the `intersection` operator produces a new **lines** value which contains all segments lying within the **regions** value. In the case of two values of type **regions**, the `intersection` operator considers all segments with segment classification $(0/2), (1/2), (2/0)$, and $(2/1)$.

The `common_border` operator creates a new spatial value of type **lines** containing the common segments of their operands of type **lines** or **regions**.

The `contour` operator computes the contour of a value of type **regions** and returns a value of type **lines**. It computes a **lines** value from the segments of only the outer cycles of the faces of a **regions** value (i.e. holes are not considered).

The `interior` operator returns a value of type **regions**, comprising the areas that are enclosed by segments of type **lines**.

Appendix B

Algorithms

The Appendix is structured as follows. Section B.1 describes the algorithms for the task dissemination and evaluator components that are common to tasks irrespective of their types. Section B.2 provides the algorithms for each of the Boolean-valued operators. Section B.3 presents the algorithms for each of the spatial-valued operators. Section B.4 presents the algorithm for the evaluation of geometry induction.

B.1 Distributed Algorithms for Spatial Operations

B.1.1 Task Dissemination

This step involves forwarding the task message towards the target region (region inside the WSN) and the dissemination of a task in the target region where evaluation is to take place. The spatial task containing spatial operators is parsed at the base station and disseminated in compact, internal form into the target region.

The task message conveys the interpretable form of the task (represented in postfix notation). It has following payload fields: the spatial task *task[]*, the source entity ID (*srcID*), the destination entity ID (*destID*), the number of hops i.e., the cost of reaching the source (*hopCount*), the geometry ID (*gmtryID*), the coordinates of the pair of points (*x1Coord*, *y1Coord*, *x2Coord*, *y2Coord*) that determine the task MBR, the amount of time between two successive evaluations of task (*reEvalPeriod*) and the number of times the task is to be run for (*duration*). In addition to these, there are two additional fields, viz., *timeAtSink* denoting the time at the sink when the task is transmitted towards network, and *taskDissemTimeOut*, denoting the amount of time after which the processing of the task is to start. After receiving the message each MBR entity makes use of these two fields to compute the time at which to start processing the task.

The protocol for task dissemination is given in Figure B.1. Initially, the gateway computes whether it is part of MBR. If it is not, it selects among its neighbours the one that is closer to the left top-most coordinate of the MBR and transmits the task message *m_TDGF* towards it. The *GreedyForwardNgbr* protocol is responsible for greedy selection of the neighbour that is closer to the MBR. Upon receiving the *m_TDGF* message neighbour entity repeats the same

```
Protocol TaskDissemination()
// $P_{INIT}$ ≡ "Only sink has the message at time $t_0$ and will act as initiator" ≡
//            {$\exists$ n $\in$ N: n has message at $t_0 \Rightarrow$ n is the sink }
// $P_{FINAL}$ ≡ "All entities that are part of path towards MBR or in the task MBR
//            will receive task message (i.e., m_TDGF or m_TDBD) at time $t_f$" ≡
//            {$\forall$ n $\in$ N: n $\in$ path(sink,MBR) $\vee$ n $\in$ MBR $\Rightarrow$ n has received task message at time $t_f$}

Status Values: S= {INITIATING, AVAILABLE, DISSEMINATING, TASK_INTERPRETING}
$S_{INIT}$= {INITIATING, AVAILABLE}
$S_{START}$= {INITIATING}
$S_{INTERMEDIATE}$= {DISSEMINATING }
$S_{TERM}$ = {AVAILABLE, TASK_INTERPRETING}

INITIATING
    Spontaneously
        isFirstLevelLeader = false
        // n.ID denotes entity ID
        srcID = n.ID
        destID = n.ID
        hopCount = 0
        // Sink to confirm whether is it part of MBR
        isPartOfMBR = PartOfMBR(x1Coord, y1Coord, x2Coord, y2Coord)
        if (isPartOfMBR = true):
            isFirstLevelLeader = true
            Become(DISSEMINATING)
        else :
            ngbr = GreedyForwardNgbr()
            Send(m_TDGF, task[], srcID, destID, hopCount + 1, x1Coord, y1Coord, x2Coord, y2Coord,
            reEvalPeriod, duration, timeAtSink, taskDissemTimeOut) to ngbr

AVAILABLE
    Spontaneously
        isFirstLevelLeader = false

    Receive(m_TDGF)
        UnPack(m_TDGF)
        isPartOfMBR = PartOfMBR(x1Coord, y1Coord, x2Coord, y2Coord)

        if (isPartOfMBR = true):
            isFirstLevelLeader =true
            Become(DISSEMINATING)
        else :
            ngbr = GreedyForwardNgbr(N(n))
            Send(m_TDGF, task[], n.ID, destID, hopCount + 1, x1Coord, y1Coord, x2Coord, y2Coord,
            reEvalPeriod, gmtryID, duration, timeAtSink, taskDissemTimeOut) to ngbr

    Receive(m_TDBD)
        isPartOfMBR = PartOfMBR(x1Coord, y1Coord, x2Coord, y2Coord)
        if (isPartOfMBR):
            UnPack(m_TDBD)
            Become(DISSEMINATING)

DISSEMINATING
    Spontaneously
        txTime = ComputeTxTime()
        Set_Alarm(a_t2, txTime)

        taskEvalTime = ComputeTimeToStartTaskEval(taskDissemTimeOut, n.CurrentTime(), timeAtSink)
        Set_Alarm(a_t3, taskEvalTime)

    Receive(m_TDBD)
        if ((isFirstLevelLeader = false) and (isPartOfMBR)):
            if ((hopCount > m_TDBD.hopCount) or (destID > m_TDBD.destID):
                UnPack(m_TDBD)
                txTime = ComputeTxTime()
                Set_Alarm(a_t2, txTime)

// Time to transmit a m_TDBD message
    When(a_t2)
        if (isFirstLevelLeader)
            // First-level leader transmits the m_TDBD message by setting the srcID, destID
            // attributes in the message to its own ID and hopCount to 0
            Send(m_TDBD, task[], n.ID, n.ID, 0, x1Coord, y1Coord, x2Coord, y2Coord, reEvalPeriod,
            gmtryID, duration, timeAtSink, taskDissemTimeOut) to N(n)
        else
            // An MBR node other that first-level leader transmits the m_TDBD message by setting the srcID
            // attribute in the message to its own ID and hopCount incremented by 1
            Send(m_TDBD, task[], n.ID, destID, hopCount + 1, x1Coord, y1Coord, x2Coord, y2Coord,
            reEvalPeriod, gmtryID, duration, timeAtSink, taskDissemTimeOut) to N(n)
        Reset_Alarm(a_t1, TD_TIMEOUT)

```

Figure B.1: Protocol *Task Dissemination*

```
// Check that **m_TDBD** message is forwarded by sufficient number of neighbours
    *When*(**a_t1**)
        **if** ((*rcvdMsgs* < *MINNUMMSGS*) **and** (*isCollisionChkDone* = **false**)):
            *isCollisionChkDone* = **true**

            **if** (*isFirstLevelLeader*)
                **Send**(**m_TDBD**, *task*[], *n.ID*, *n.ID*, *0*, *x1Coord*, *y1Coord*, *x2Coord*, *y2Coord*,
                *reEvalPeriod*, *gmtryID*, *duration*, *timeAtSink*, *taskDissemTimeOut*) **to N**(*n*)
            **else**
                **Send**(**m_TDBD**, *task*[], *n.ID*, *destID*, *hopCount + 1*, *x1Coord*, *y1Coord*, *x2Coord*, *y2Coord*,
                *reEvalPeriod*, *gmtryID*, *duration*, *timeAtSink*, *taskDissemTimeOut*) **to N**(*n*)

            **Set_Alarm**(**a_t2**, *TD_TIMEOUT/3*)

        **elseif** (*isCollisionChkDone* = **false**):
            *isCollisionChkDone* = **true**

    *When*(**a_t3**)
    // Time to evaluate a task in the message
        **Become**(TASK_INTERPRETING)
        Evaluate(*task*, *reEvalPeriod*, *duration*, *gmtryID*)

```

Figure B.1 (continued)

process of greedy selection of the closest neighbour, if it finds itself not part of MBR. Otherwise, it takes on the role of *first-level leader*, records the information, updates the information in the packet related to the cost of reaching the source (i.e., the *hopCount* attribute is set to zero), sets its own ID as the *srcID* and *destID* in the task message and changes the message ID to *m_TDBD* before broadcasting it.

Each entity that receives the message *m_TDBD*, and is part of the task MBR takes on the role of DISSEMINATING. Upon receiving the *m_TDBD* message for the first time, it records the information, and increments the *hopCount* by one, sets its own ID as *srcID* and broadcasts the message. If an MBR entity has already received a task message, and receives it again, it checks whether the *hopCount* or *destID* is less than the one it has recorded earlier. If it is, it records the *srcID* as the parent entity ID, updates the attributes *hopCount* and *destID*, and broadcasts the task message. If it is not, it ignores the message.

Packet collisions are a significant challenge in WSNs. To avoid the chances of an MBR entity not receiving a task message because of collision, each MBR entity computes the time to transmit before broadcasting the task message to its neighbours. The *ComputeTxTime* procedure is responsible for computing the transmit time. Each entity randomly computes a time to transmit within the range (0 - *TD_DELAY_VAR*). *TD_DELAY_VAR* denotes the maximum expected variance in task dissemination. To minimize the effect of small-scale clock drift, a minimum fixed delay (denoted by *MIN_DELAY*) is added to the computed random time for transmission of the request message to ensure that in all neighbouring entities the radio is ON. In addition, before completing the task dissemination phase, each entity checks whether it has received the task message from a specific number of neighbours to make sure that the task dissemination is taken care by enough neighbouring entities. If not, it retransmits the task message again. The *UnPack* procedure is responsible for assigning the message attributes to the local variables. After the completion of task dissemination, all MBR entities call the *Evaluate* protocol in Figure B.3.

```
Procedure ComputeTxTime()
txTimeOut = RandomValue.rand()
if (n.ID ≠ 0):
    txTimeOut = (txTimeOut % TD_DELAY_VAR)
    txTimeOut = ((txTimeOut* n.ID) % TD_DELAY_VAR)) + MIN_DELAY
else :
    txTimeOut = (txTimeOut % TD_DELAY_VAR)
return txTimeOut
```

Figure B.2: Procedures used by *Task Dissemination* protocol

B.1.2 Task Evaluation

Every task MBR entity evaluates the interpretable structure conveyed by the task message, i.e., every entity runs an interpreter for the geometry induction and algebraic evaluation steps required in our approach to spatial analysis. As discussed in Section 5.6.2, the *Evaluate* component is responsible for validating the task. In addition, it is responsible for computing the type of valid tasks by inference on the operator occurring in it and for forwarding it to a specific component. The evaluation logic depends on the type of the task. This section, only discuss the *Evaluate* protocol (Figure B.3). The evaluation logic for Boolean-valued tasks is discussed in Section 5.6.3. Section B.3 discusses the evaluation logic for spatial-valued tasks and the evaluation logic for inducing geometry is discussed in Section B.4.

```
Protocol Evaluate(task, reEvalPeriod, duration, gmtryID)
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n can perform message interpretation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has performed message validation and interpretation at time $t_f$}

Status Values: S= {TASK_INTERPRETING, TASK_EVALUATING, AVAILABLE}
$S_{START}$= {TASK_INTERPRETING}
$S_{TERM}$ = {AVAILABLE, TASK_PREPROCESSING}

TASK_INTERPRETING
    Spontaneously
        isTaskValid = ValidityCheck(task)
        if (isTaskValid) :
            taskType = ExtractTaskType(task)
            if (taskType = BOOLEAN_VALUED):
                Become(TASK_PREPROCESSING)
                EvalBooleanValuedTask(task, reEvalPeriod, duration)
            elseif (taskType =SPATIAL_VALUED):
                Become(TASK_PREPROCESSING)
                EvalSpatialValuedTask(task, reEvalPeriod, gmtryID, duration)
            elseif (taskType = INDUCED):
                Become(TASK_PREPROCESSING)
                InduceGeometry(task, reEvalPeriod, duration, gmtryID)
        else :
            Become(AVAILABLE)

```

Figure B.3: Protocol *Evaluate*

B.2 Boolean-valued Task

The section describes the protocols for the Boolean-valued operators, aggregation and result processing.

B.2.1 Boolean-valued Operators

This section will now describe the protocol for each of the Boolean-valued operators.

B.2.1.1 AreaInside

Recall, from Section 4.2.1, that `area_inside` computes whether an operand of type **points**, **lines** or **regions** lies `area_inside` an operand of type **regions**. For the computation of this operation, the information required by an entity is available in its own GIT. The protocol is shown in Figure B.4.

```
Protocol AreaInside(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the area_inside operation at time $t_0\}$
// $P_{FINAL} \equiv\{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, unknown or false at time $t_f\}$

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        stateOP = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckAIDataTypeValidity(tupleG1, tupleG2)

            if (isOperandsDTValid):
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = false) and (tupleG2->bndryNode = true)):
                        stateOP = false
                    else
                        stateOP = true
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    stateOP = unknown
                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    stateOP = false
            else :
                stateOP = OP_NOT_SUPPORTED

        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure B.4: Protocol *AreaInside*

B.2.1.2 OnBorderOf

The procedures for the `on_border_of` protocol includes *CheckOBODataTypeValidity*, and *LocalGITLookup_Search*. Procedure *CheckOBODataTypeValidity* is responsible for computing whether the operands are valid i.e., it returns **true** if an entity satisfies both of the operands data type or if it it is part of only one of the operand and satisfies its data type. *LocalGIT-Lookup_Search* is responsible for local GIT-lookup and to return the address of the tuple in GIT.

```
Protocol OnBorderOf($gmtry1ID$, $gmtry2ID$, $stateOP$)
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the on_border_of operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, unknown, true, or false at time $t_f$}

Status Values: S= {TASK_EVALUATING,OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}
OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        stateOP = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckOBODataTypeValidity(tupleG1,tupleG2)
            if (isOperandsDTValid)
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if (tupleG2->bndryNode = true):
                        stateOP = true
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    if (tupleG2->bndryNode = true):
                        stateOP = unknown
            else :
                stateOP = OP_NOT_SUPPORTED
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure B.5: Protocol *OnBorderOf*

B.2.1.3 Equals

Recall, from Section 4.2.1, that `equals` computes whether two operands of the same type are equal. For the computation of this operation, the information required by an entity is available in its own GIT and EIT. The procedures for the `equals` protocol (Figure B.6) include *Check-EqualsDTValidity*, *EIT_IsEdgeInfEqual* and *LocalGITLookup_Search*. *CheckEqualsDTValidity* is responsible for computing whether the operands are valid i.e., it returns **true** if an entity satisfies both of the operands data type or if it it is part of only one of the operand and satisfies its data type. *LocalGITLookup_Search* is responsible for local GIT look and and returning the address of the tuple upon success, and *EIT_IsEdgeInfEqual* responsible for checking entries for neighbours in its EIT related to both operands. It returns **false** upon finding that it has an entry in EIT for one of the operands but not for both and **true** otherwise.

B.2.1.4 NotEquals

Recall, from Section 4.2.1, that `not_equals` computes whether two operands of the same type are unequal. For the computation of this operation, the information required by an entity is available in its own GIT and EIT. The procedures for the `not_equals` protocol in Figure B.7 include *CheckEqualsDTValidity*, *EIT_IsEdgeInfEqual* and *LocalGITLookup_Search.*

B.2.1.5 VertexInside

Recall, from Section 4.2.1, that `vertex_inside` computes whether an operand of type **regions** lies `vertex_inside` an operand of type **regions**. The `vertex_inside` protocol is shown in Figure

```
Protocol Equals(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the equals operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, or false at time $t_f\}$

Status Values: $S$= {TASK_EVALUATING,OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}
OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        stateOP = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckEqualsDTValidity(tupleG1,tupleG2)
            if (isOperandsDTValid):
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if (tupleG1->bndryNode = tupleG2->bndryNode):
                        stateOP = true
                    else :
                        stateOP = false
            else :
                stateOP = OP_NOT_SUPPORTED
        stateOP = EIT_IsEdgeInfEqual(gmtry1ID, gmtry2ID)
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.6: Protocol *Equals*

B.8.

B.2.1.6 BorderInCommon

The `border_in_common` operation computes whether both operands have at least one but possibly more CBSs. For the computation of the operation state, a CBN requires information from its neighbours. Recall, from Section 4.2.1, that `border_in_common` takes operands that are any combination of **lines**, and **regions** values. The *CheckBICDataTypeValidity* procedure is responsible for computing whether the operands are valid, i.e., it returns **true** if an entity satisfies the operand's data type, which it is a member of.

For the computation of the CBS condition, the `border_in_common` protocol invokes the *NeighbourLookUp* protocol with a request to compute CBS (line 35 in Figure B.9). This protocol is called by all MBR entities, but only entities that belong to one or both operands take part in communication. All other MBR entities wait for the protocol to finish (line 52 in Figure B.10). The reason is to maintain synchronization in an intra-network task evaluation phase comprising many operators. For the timely coordination among the MBR entities, participating entities must start and finish the processing of each spatial operator part of the complex task at an appropriate time.

Each CBN entity requests information from its neighbouring entities by transmitting the *m_GITREQ* message, which conveys its own ID (srcID) and the GIDs of the first and second operands. To avoid collisions, before transmitting a message, an entity compute a transmit time that is a random time between 0 and the maximum delay variance *GITRQ_DELAY_VAR*. To avoid small clock drifts, a minimum fixed delay (i.e., *MIN_DELAY*) is added to the computed

```
Protocol NotEquals(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the not_equals operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, or false at time $t_f\}$

Status Values: S= {TASK_EVALUATING,OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}
OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        stateOP = true
        isEdgeInfEqual = true

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckNotEqualsDTValidity(tupleG1,tupleG2)
            if (isOperandsDTValid):
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if (tupleG1->bndryNode = tupleG2->bndryNode):
                        stateOP = unknown
                    else :
                        stateOP = true
            else :
                stateOP = OP_NOT_SUPPORTED

        isEdgeInfEqual = EIT_IsEdgeInfEqual(gmtry1ID, gmtry2ID)
        if (isEdgeInfEqual=false):
            stateOP = true
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure B.7: Protocol *NotEquals*

random time for the transmission of a request message by a CBN (to ensure that all neighbouring entities radio is ON). The procedure *ComputeGITTxTime* for the *NeighbourLookUp* protocol is responsible for computing the random time for transmitting *m_GITREPLY*, *m_GITREQ*, *m_InvalidCLUTInf* and *m_InvalidEITInf* messages, taking into account the maximum transmission delay variance for request and reply messages.

Initially, all MBR entities that belong to one or both operands wait for the reception of the *m_GITREQ* message. Once the wait period for the reception of the *m_GITREQ* message finishes, all entities that have received the *m_GITREQ* message and belong to one or both operands transmit the *m_GITREPLY* reply message which conveys, its own ID (*srcID*), and the membership status for the first and second operands, and the boundary state for the first and second operands.

Upon reception of the *m_GITREPLY* message (line 74 in Figure B.10), each *CBN* entity computes the sector of the neighbour based on its location, assigns the entity to the proper sector (using the *NIT_ComputeSector* procedure) and finds the minimum distance neighbour in each sector. A CBN records the entity ID and the status of its minimum distance neighbour in each of its sectors. A neighbour state *NBRCBN* denotes that the neighbour is a CBN, *NBRINTERSECT* denotes that the neighbour belongs to both operands but is not a CBN, *NBRG1* denotes that the neighbour belongs to the first operand only, and, finally, *NBRG2* denotes that the neighbour belongs to the second operand only.

After the completion of the wait period for the reception of replies from neighbours, the

```
Protocol VertexInside(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the vertex_inside operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, unknown or false at time $t_f$}

Status Values: S= {TASK_EVALUATING,OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}
OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        // LocalGITLookup_search method returns the address of the tuple in GIT
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        stateOP = false
        isOperandsDTValid = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckVIDataTypeValidity(tupleG1,tupleG2)
            if (isOperandsDTValid):
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if (tupleG2→bndryNode = false):
                        stateOP = true
                    else :
                        stateOP = false
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    stateOP = unknown
                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    stateOP = false
            else :
                stateOP = OP_NOT_SUPPORTED
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure B.8: Protocol *VertexInside*

CBN computes whether it is part of a valid CBS (line 7 in *ComputeCBSExistence* in Figure B.12). For this purpose, each CBN confirms that the segments formed by itself and its neighbouring CBNs are valid CBS of the first and the second operand. For this purpose the *EIT_ConfirmSegment* procedure computes whether the segment is in the EIT table. If it is not, then the *isCBSNotCommonToAdjacentLUTs* procedure computes whether the boundary segment is a common segment of two adjacent LUT formed by entities in the first operand (for this purpose, it uses the neighbour state information *sectorNbrState[]*). If the segment is a valid boundary segment for one of the operands, the entity checks the status of the segment for the other operand. The *NeighbourLookUp* protocol (called by entities with a request to compute existence of CBS), returns **segment_node** if the segment exists, or **commonBoundaryNode** if the node is a CBN.

This section, only describe the working of *NeighbourLookUp* protocol to detect the existence of a CBS. The *NeighbourLookUp* protocol also allows a CBN to compute whether it is part of a common area, part of an `area_disjoint` **regions** value and part of a CBS, part of an `area_disjoint` **regions** value and not part of a CBS, whether it is a **meetingPoint** or an **intersectingPoint** between two values of type **lines** or between one of type **lines** and another of type **regions**. The procedures involved by the *NeighbourLookUp* protocol responsible for computing the required information are shown in Figures B.18, B.16, B.21, B.19, B.29, and

```
Protocol BorderInCommon(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the border_in_common operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, or unknown at time $t_f\}$

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        CBN = false
        stateOP = unknown
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS

        if ((tupleG1!= null) or (tupleG2!= null)):
            isOperandsDTValid = CheckBICDataTypeValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode= true)):
                        CBN = true
            else :
                stateOP = OP_NOT_SUPPORTED

            if (isReqdToPartInNgbrLookup = true)
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, CBS, &CBNState, border_in_common, NULL)
            else :
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(NULL, NULL, CBS, &CBNState, border_in_common, NULL)


POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true):
            if (CBNState = segment_node):
                stateOP = true
        stateOP = unknown

        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.9: Protocol *BorderInCommon*

```
Protocol NeighbourLookUp(GIT* tupleG1, GIT* tupleG2, spatialOPChk, *operationState, OperationID,
drvdGmtryID)
// $P_{INIT} \equiv$ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the NeighbourLookUp protocol with a request to compute
//            whether CBN is part of CBS, Common Area, CBS and Common area both, derived line,
//            or derived region at time $t_0$}
// $P_{FINAL} \equiv$ {∀ n ∈ N: n ∈ MBR ⇒ n ∈ CBN computed a operation state as false, meetingPoint,
//            intersectingPoint, segment_node, true, or commonBoundaryNode }

Status Values: S= {OPERATION_EVALUATING, IDLE, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {POST_NEIGHBOURLOOKUP}

OPERATION_EVALUATING
    Spontaneously
        isGITReqPhase = true
        reqMsgcount = 0
        isLocalComputationDone = false
        isCLUTComputationDone = false
        opState = false
        CBN = false
        IsCLUTPossible = false
        isEITComputationDone = false
        Struct CommonLocalizedUnitTriangle CLUT[]
        InvalidNgbrInf[]

        if ((tupleG1 != NULL) and (tupleG2 != NULL)):
            if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode= true)):
                // CBN's to transmit GIT-lookup request to neighbours
                CBN = true
                txTime = ComputeGITTxTime(m_GITREQ)
                Set_Alarm(a_t1, txTime)

            if (spatialOPChk = RRINTERSECTION)
                // If common area existence request is made by Intersection operator non-CBN's
                // that belongs to both operands and to the boundary of one of the operands
                // also needs to to transmit GIT-lookup request to neighbours along with CBNs
                if ((tupleG1→bndryNode = true) or (tupleG2→bndryNode= true)):
                    txTime = ComputeGITTxTime(m_GITREQ)
                    Set_Alarm(a_t1, txTime)

            if (spatialOPChk = RRMINUS):
                // If part of derived region request is made by Minus operator non-CBN's that
                // belongs to both operands and to the boundary of second operand also needs
                // to transmit GIT-lookup request to neighbours along with CBNs
                if ((tupleG1→bndryNode = false) and (tupleG2→bndryNode= true)):
                    txTime = ComputeGITTxTime(m_GITREQ)
                    Set_Alarm(a_t1, txTime)

            Set_Alarm(a_t2, WAITGITREQ)

        elseif ((tupleG1 != null) or (tupleG2!= null)):
            Set_Alarm(a_t2, WAITGITREQ)
        else
            Become(IDLE):

    Receive(m_GITREQ)
        // To count number of GIT-lookup requests received from neighbours
        if ((tupleG1→GID = m_GITREQ.GID1)or (tupleG2→GID= m_GITREQ.GID2)):
            reqMsgCount = reqMsgCount + 1

// Idle wait period Expires
    When(a_t5)
        isGITReqPhase = false
        Set_Alarm(a_t2, FIRET2)

// Time to transmit m_GITREPLY message
    When(a_t3)
        Send(m_GITREPLY, n.ID, IsPartOfGmtry(tupleG1), tupleG1→bndryNode,
        IsPartOfGmtry(tupleG2), tupleG2→bndryNode) to N(n)

        if (reqMsgcount < MAXREPLY):
            SendGITReply()

// Time to transmit m_GITREQ message
    When(a_t1)
        Send(m_GITREQ, n.ID, tupleG1→GID, tupleG2→GID) to N(n)
```

Figure B.10: Protocol *NeighbourLookUp*

```
    Receive(m_GITREPLY)
        if (CBN = true):
            if ((m_GITREPLY.GIDG1=true) and (m_GITREPLY.GIDG2=true)):
                if ((m_GITREPLY.BNG1=true) and (m_GITREPLY.BNG2=true)):
                    nbrState = NBRCBN
                else :
                    nbrState = NBRINTERSECT

            elseif ((m_GITREPLY.GIDG1=true) and (m_GITREPLY.GIDG2=false)):
                nbrState = NBRG1

            elseif ((m_GITREPLY.GIDG1=false) and (m_GITREPLY.GIDG2=true)):
                nbrState = NBRG2

            sector = NIT_ComputeSector(m_GITREPLY.sourceID)

            if (sectorNbrID[sector]= EMPTY):
                sectorNbrState[sector] = nbrState
                sectorNbrID[sector] = m_GITREPLY.sourceID
            else :
                distance = NIT_getDistanceToMe(m_GITREPLY.sourceID)
                prvNgbrDistance = NIT_getDistanceToMe(sectorNbrID[sector])
                if (distance < prvNgbrDistance):
                    sectorNbrState[sector] = nbrState
                    sectorNbrID[sector] = m_GITREPLY.sourceID

// Time to transmit GIT Reply message and to perform local computation based on type of
// computation request and to return the control back
    When(a_t2)
        if (isGITReqPhase = true):   // GIT-lookup Reply phase
            isGITReqPhase = false
            if (reqMsgcount > MAXREPLY):
                reqMsgcount= MAXREPLY
            Set_Alarm(a_t2, WAITGITREPLY)
            if (reqMsgcount > 0):
                SendGITReply()

            // Local Computation after completion of GIT-lookup Reply phase
        elseif ((isGITReqPhase = false) and (isLocalComputationDone = false)):
            if (spatialOPChk = CBS):
                ComputeCBSExistence()
            elseif ((spatialOPChk = RRINTERSECTS) or (spatialOPChk = RRINTERSECTION)):
                ComputeCommonAreaExistence()

            elseif ((spatialOPChk = LRINTERSECT)or (spatialOPChk = LRMEETS)):
                ComputeSegmentAreaInsideExistence()

            elseif ((spatialOPChk = LLMEETS)or (spatialOPChk = LLINTERSECT)):
                ComputeMeetingPointExistence()

            elseif ((spatialOPChk = SHAREINFO)or (spatialOPChk = WAITIDLE)):
                Set_Alarm(a_t2,LOCALCOMPUTATIONTIME+CLUTCOMPUTETXRXTIME)
                isLocalComputationDone = true

            elseif ((spatialOPChk = SHAREINFO-DG)or (spatialOPChk = WAITIDLE-DG)):
                Set_Alarm(a_t2,EITCOMPUTETXRXTIME+CLUTCOMPUTETXRXTIME)
                isLocalComputationDone = true
            elseif (spatialOPChk = CBSANDCMNAREA):
                ComputeCommonAreaAndCBSExistence()

            elseif (spatialOPChk = LLMINUS):
                ComputePartOfDerivedline()

            elseif (spatialOPChk = RRMINUS):
                ComputePartOfDerivedRegion()
            // Finish evaluation and return control
        elseif ((isGITReqPhase = false) and (isLocalComputationDone = true)):
            *operationState = opState
            Become(POST_NEIGHBOURLOOKUP)
            // Method DirectControlToOperation is responsible for directing the control to the required operation
            DirectControlToOperation(OperationID)

// Time to transmit m_InvalidCLUTInf message
    When(a_t4)
        if (isCLUTComputationDone = true):
            InvalidNgbrInf[] = AssignNgbrInfEachInvalidClut()
            Send(m_InvalidCLUTInf, n.ID, InvalidNgbrInf[]) to N(n)

// Time to transmit m_InvalidEITInf message
    When(a_t6)
        if (isEITComputationDone = true):
            Send(m_InvalidEITInf, n.ID, drvdGmtryID, InvalidNgbrInf[]) to N(n)

```

Figure B.10 (continued)

```
    Receive(m_InvalidCLUTInf)
        if (IsCLUTPossible=true):
            UpdateCLUTSInformation(m_InvalidCLUTInf)

    Receive(m_InvalidEITInf)
        if (m_InvalidEITInf.drvdGmtryID = drvdGmtryID):
            EIT_RemoveEdgesMinusOP(m_InvalidEITInf)

IDLE
    Spontaneously
        Set_Alarm(a_t5, WAITGITREQ + WAITGITREPLY)

```

Figure B.10 (continued)

B.30. The descriptions of some of these procedures are provided in Appendix B and for others later in this chapter.

```
Procedure ComputeGITTxTime(txType)
delayVariance = 0
txTimeOut = RandomValue.rand()

if ((txType = m_GITREQ) or (txType = m_InvalidCLUTInf) or (txType =m_InvalidEITInf)):
    delayVariance= GITRQ_DELAY_VAR
else :
    delayVariance= GITRP_DELAY_VAR

if (n.ID ≠ 0):
    txTimeOut=(txTimeOut % delayVariance)
    txTimeOut=(((txTimeOut* n.ID) % delayVariance)))
else :
    txTimeOut= (txTimeOut % delayVariance)

if (txType = m_GITREQ):
    txTimeOut = txTimeOut + MIN_DELAY
return txTimeOut
```

```
Procedure SendGITReply()
txTime = ComputeGITTxTime(m_GITREPLY)
Set_Alarm(a_t3, txTime)
```

Figure B.11: Procedures used by *NeighbourLookUp* protocol

B.2.1.7 EdgeInside

In this section, we describe the protocol for the computation of the `edge_inside` operation. Recall, from Section 4.2.1, that *EdgeInside* takes operands of type **regions**. The `edge_inside` computes whether an operand of type **regions** lies `edge_inside` another operand of type **regions**. For the computation of the operation state for this operation, CBN requires information from its neighbours to compute whether it is part of CBS. Other entities can compute its state based on the information in its own GIT.

For the computation of the CBS `edge_inside` protocol makes use of the *NeighbourLookUp* protocol with a request to compute common segment. For the computation of the CBS, `edge_inside` in Figure B.13 makes use of the *NeighbourLookUp* protocol shown in Figure B.10. The procedures of *NeighbourLookUp* protocol for computing CBS are shown in Figure B.12. The *NeighbourLookUp* protocol was described in Section B.2.1.6. Upon completion of *neighbourLookUp* protocol with a request to compute CBS, all CBNs set their state to **false** if they form a CBS (i.e., receive a state of **segment_node** from *NeighbourLookUp* protocol).

```
Procedure ComputeCBSExistence()
isValidCBSExists = false
if ((isGITReqPhase=false)and (isLocalComputationDone = false)):
    Set_Alarm(a_t2, LOCALCOMPUTATIONTIME)
    reqMsgcount = 0

    if (CBN =true):
        isValidCBSExists = ConfirmCBS()
        if (isValidCBSExists =true):
            opState = segment_node
        else :
            opState = commonBoundaryNode
        isLocalComputationDone = true
```

```
Procedure ConfirmCBS()
validBSegG1 = false
validBSegG2 = false
isValidSeg = false

for sector = 1 to 12:
    if (sectorNbrState[sector] = CBN):
        validBSegG1 = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
        if (validBSegG1 = true):
            validBSegG1 = isCBSNotCommonToAdjacentLUTs(sector, tupleG1)
            if (validBSegG1= true):
                validBSegG2 = isCBSNotCommonToAdjacentLUTs(sector, tupleG2)
        if ((validBSegG1= true) and (validBSegG2= true)):
            isValidSeg = true
            Break
return isValidSeg
```

Figure B.12: Procedures used by *NeighbourLookUp* protocol to confirm CBS

B.2.1.8 EdgeDisjoint

This section, describes the protocol for the computation of the `edge_disjoint` operation, which computes whether two **regions** are `area_disjoint` and have no CBS. For the computation of the operation state, a CBN requires information from its neighbours to compute whether it is part of a CBS, other entities only use information in their own GIT.

For the computation of the CBS condition, the `edge_disjoint` protocol in Figure B.14 makes use of the *NeighbourLookUp* protocol. The procedures of *NeighbourLookUp* protocol for computing CBS are shown in Figure B.12. The *NeighbourLookUp* protocol was described in the preceding section. Upon completion of the *NeighbourLookUp* protocol, a CBN sets its state to **false** if it forms a CBS (i.e., receives a **segment_node** state from the *NeighbourLookUp* protocol).

B.2.1.9 AreaDisjoint

This section, describes the protocol for the computation of the `area_disjoint` operation informally described in Section 4.2.1.6. To compute this operation, CBNs require information from their neighbours to compute whether they belong to a common area, while other entities only need information available in their own GIT. For the computation of the common areas, the `area_disjoint` protocol request *NeighbourLookUp* protocol to confirm the existence of common area (line 37 in Figure B.15). The working of the *NeighbourLookUp* protocol was explained in Section B.2.1.6. The procedures used by the protocol for the computation of a common area are shown in Figure B.16.

After the completion of the wait period for the reception of replies from neighbours (line

```
Protocol EdgeInside(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the edge_inside operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, true, unknown or false at time $t_f\}$

Status Values: S= {TASK_EVALUATING,OPERATION_EVALUATING,POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}
OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        CBN = false
        stateOP = false
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckEIDataTypeValidity(tupleG1,tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode = true)):
                        CBN = true
                    elseif ((tupleG1->bndryNode = false) and (tupleG2->bndryNode = true)):
                        stateOP = false
                    else :
                        stateOP = true
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    stateOP = unknown
                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    stateOP = false
            else :
                stateOP = OP_NOT_SUPPORTED

        if (isReqdToPartInNgbrLookup = true):
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(tupleG1,tupleG2,CBS,&CBNState,edge_inside,NULL)
        else :
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL,NULL,CBS,&CBNState,edge_inside,NULL)

POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true)
            if (CBNState = segment_node):
                stateOP = false
            else :
                stateOP = true
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.13: Protocol *EdgeInside*

number 112 in Figure B.10) the *ComputeCommonAreaExistence* procedure (in Figure B.16) is called by the entities. The *ChkForCommonNeighbours* procedure in Figure B.16 computes whether an entity has a neighbour with *NBRINTERSECT* state in any of the sectors and more than one neighbour with a *NBRCBN* state. If it finds any neighbour with a *NBRINTERSECT* state, the CBN sets its *opState* to **true**. If it has less than two neighbours with *NBRCBN* state, it does nothing and waits for the completion of the protocol. Otherwise, it repeats the following process for each distinct pair of neighbours with *NBRCBN* state (i.e., CBN neighbours). It first computes whether it forms a triangle with its CBN neighbours. If it does, it computes whether it forms a CLUT with its CBN neighbours. The *ComputePossibleCLUTs* procedure in Figure B.16 is invoked to test whether an entity forms a CLUT. If it does, it adds this information to

Protocol EdgeDisjoint(*gmtry1ID*, *gmtry2ID*, *stateOP*)

```
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the edge_disjoint operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed a operation state as
//            opNotSupported, NotPartOfOperands, true, false at time $t_f$}

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        CBN = false
        stateOP = false
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckEDDataTypeValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode = true)):
                        CBN = true
                    else :
                        stateOP = false
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    stateOP = true
                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    stateOP = true
            else :
                stateOP = OP_NOT_SUPPORTED
        if (isReqdToPartInNgbrLookup = true)
            NeighbourLookUp(tupleG1, tupleG2, CBS, &CBNState, edge_disjoint, NULL)
        else :
            NeighbourLookUp(NULL, NULL, CBS, &CBNState, edge_disjoint, NULL)


POST_NEIGHBOURLOOKUP
    Spontaneously

        if (CBN = true)
            if (CBNState = segment_node):
                stateOP = false
            else :
                stateOP = true
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.14: Protocol *EdgeDisjoint*

the list of CLUT records.

```
Protocol AreaDisjoint(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the area_disjoint operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed a operation state as
//            opNotSupported, NotPartOfOperands, true, false at time $t_f$}

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        stateOP = false
        CBN = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckADDataTypeValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 = null) or (tupleG2 = null)):
                    stateOP = true
                elseif ((tupleG1 != null)and (tupleG2 != null)):
                    if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode = true)):
                        CBN = true
                    else :
                        stateOP = false
            else :
                stateOP = OP_NOT_SUPPORTED
        if (isReqdToPartInNgbrLookup = true)
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(tupleG1, tupleG2, RRINTERSECTS, &CBNState, area_disjoint, NULL)
        else :
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, RRINTERSECTS, &CBNState, area_disjoint, NULL)


POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true)
            if (CBNState = true)
                stateOP = false
            else :
                stateOP = true
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.15: Protocol *AreaDisjoint*

Once *ComputePossibleCLUTs* has finished, the CBN checks whether the CLUT is valid based on the information in its *EIT* table. Each *CLUT* record has an attribute denoting its EIT validity (apart from the IDs of neighbouring entities). A CBN can compute such information using the *EIT_ConfirmCLUTS* procedure, which checks CLUT validity for the elements in the list on the basis of a local EIT lookup. If a CBN finds that it is part of an invalid CLUT, it sends information about the latter in a *m_InvalidCLUTInf* message during the invalid CLUT reply phase.

The *m_InvalidCLUTInf* message comprises the *source ID* and the *InvalidCLUTInf* list, in which each tuple includes the pair of neighbour IDs in the invalid CLUT. A CBN with a **true** value for *isCLUTPossible* updates its CLUT information upon receiving the *m_InvalidCLUTInf*

```
Procedure ComputeCommonAreaExistence()
if ((isGITReqPhase= false) and (isCLUTComputationDone = true)):
    Set_Alarm(a_t2, LOCALCOMPUTATIONTIME)
    isLocalComputationDone = true
    if ((opState=false) and (CBN= true) and (IsCLUTPossible=true))
        opState = PartOfValidCLUT()

    // If the request for computing common area existence is made my
    // Intersection operator then it is required to update the EIT table
    if (spatialOPChk=RRINTERSECTION)
        EIT_UpdateInfIntersectionOP(drvdGmtryID, gmtry1ID, gmtry2ID, sectorNbrID, sectorNbrState)

elseif ((isGITReqPhase=false)and (isCLUTComputationDone = false)):
    // Time to receive and transmit m_InvalidCLUTInf message
    Set_Alarm(a_t2, CLUTCOMPUTETXRXTIME)
    reqMsgcount = 0
    ChkForCommonNeighbours()

    if (IsCLUTPossible = true)
        ComputePossibleCLUTs()
        CLUT[] = EIT_ConfirmCLUTS(CLUT[])
        isCLUTComputationDone = true
        if (IsInvalidEdgeInfoInCLUTS())
            SendInvalidCLUTInfo()

Procedure ChkForCommonNeighbours()
stateOP = false
IsCLUTPossible = false
for sectori = 1 to 12
    if ((sectorNbrState[sectori] = NBRINTERSECT) and (CBN =true))
        stateOP = true
    elseif (sectorNbrState[sectori] = NBRCBN):
        cntNgbrCBN = cntNgbrCBN + 1

if ((cntNgbrCBN >= 2) and (CBN =true))
    IsCLUTPossible = true

Procedure ComputePossibleCLUTs()
isCreatingTriangle = false
isCreatingCLUT = false
CLUTExists = false

if ((IsCLUTPossible = true) and (CBN = true))
    for sectori = 1 to 12
        for sectorj = 2 to 12
            if ((sectorNbrState[sectori] = CBN) and (sectorNbrState[sectorj] = CBN)
            and (sectori != sectorj))
                // n.ID denotes entity ID
                isCreatingTriangle = NIT_CreatingTriangle(n.ID, sectorNbrID[sectori], sectorNbrID[sectorj])

                if (isCreatingTriangle = true)
                    isCreatingCLUT = CreatingCLUT(n.ID, sectorNbrID[sectori], sectorNbrID[sectorj])

                if (isCreatingCLUT = true)
                    CLUTExists = IsCLUTAlreadyExists(CLUT[], n.ID, sectorNbrID[sectori],
                    sectorNbrID[sectorj])
                    if (CLUTExists = false)
                        InsertCLUTInfo(n.ID, sectorNbrID[sectori], sectorNbrID[sectorj])

Procedure SendInvalidCLUTInfo()
txTime = ComputeGITTxTime(m_InvalidCLUTInf)
Set_Alarm(a_t4, txTime)
```

Figure B.16: Procedures used by *Neighbour-Lookup* protocol to compute common area

message. After the completion of the wait period for the *m_InvalidCLUTInf* message, a CBN with **false** *opState* computes whether it is still a part of at least one valid CLUT. If it is, it set its *opState* to **true**.

Upon completion of the *NeighbourLookUp* protocol, a CBN sets its operation state for the `area_disjoint` operation to **false** if it receives a **true** state from the invocation of the *NeighbourLookUp* protocol with a request to compute the existence of common area between operands.

B.2.1.10 Intersects

This section, describes the protocol for the computation of the `intersects` operation. Recall from Section 4.2.1, that `intersects` computes whether two operands, in any combination of type **lines** and **regions**, intersect.

If both operands are of type **lines**, the MBR entities use the *NeighbourLookUp* protocol shown in Figure B.10 with a request to compute the existence of an `intersects` relationship between values of type **lines** (line 39 in Figure B.17). If one of the operands is of type **lines** and other of type **regions** the MBR entities use the *NeighbourLookUp* protocol with a request to compute whether the **lines** value `intersects` the **regions** value (line 44 in Figure B.17). Finally, if both operands of type **regions**, the entities use the *NeighbourLookUp* protocol with a request to compute the existence of common area (line 48 in Figure B.17).

The procedures used by *NeighbourLookUp* protocol for computing the existence of an `intersects` relationship between values of type **lines** are given in Figure B.18. The procedures used for computing the existence of an `intersects` relationship between values of type **lines** and **regions** are shown in Figure B.19. Finally, the procedures for computing the existence of `intersects` relationship between values of type **regions** value are shown in Figure B.16. To compute whether one **lines** value `intersects` another, after the completion of GIT reply phase the *NeighbourLookUp* protocol calls the *ComputeMeetingPointExistence* procedure to compute the existence of a CBS (line 8 in Figure B.18). If a CBN is not part of a valid CBS, it checks whether its minimum distance neighbours in sectors form a cyclic concatenation of two sublists, each belonging to one of the operand using the *ConfirmCyclicConcatSubLists* procedure. The protocol returns **false** if the CBN is part of a valid CBS, **meetingPoint** if the neighbours list is the cyclic concatenation of two sublists, and **intersectingPoint** if the neighbours list is not a cyclic concatenation of two sublists.

To compute the `intersects` relationship between **lines** and **regions** value, after the completion of GIT reply phase, the *ComputeSegmentAreaInsideExistence* procedure in Figure B.19 calls the *ComputeSegIntersectRegion* procedure to check whether one or more segments of an operand of type **lines** is `area_inside` the operand of type **regions**.

The *NeighbourLookUp* protocol returns **intersectingPoint**, if at least one neighbour of CBN has a state *NBRINTERSECT* (i.e., the neighbour belongs to interior of **regions** value and to **lines** value), or has a state *NBRCBN* (i.e., the neighbour belongs to boundary of **regions** value and to **lines** value) and the segment created is a valid segment. If a CBN is not part of a valid segment, then it it checks whether its minimum distance neighbours in sectors form a cyclic concatenation of two sublists, each belonging to one of the operand

Protocol Intersects(*gmtry1ID*, *gmtry2ID*, *stateOP*)

```
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the intersects operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//            opNotSupported, NotPartOfOperands, true, false or unknown at time $t_f\}$

Status Values: $S$= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        stateOP = unknown
        CBN = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckIntersectsDTValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 = null) or (tupleG2 = null)):
                    stateOP = unknown
                elseif ((tupleG1 != null)and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode = true)):
                        CBN = true
                    else :
                        stateOP = true
            else :
                stateOP = OP_NOT_SUPPORTED

        if (isReqdToPartInNgbrLookup = true)
            if ((tupleG1->dataType = lines) and (tupleG2->dataType = lines)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LLINTERSECT, &CBNState, intersects, NULL)

            elseif ((tupleG1->dataType = lines) and (tupleG2->dataType = regions)) or
            ((tupleG1->dataType = regions) and (tupleG2->dataType = lines)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LRINTERSECT, &CBNState, intersects, NULL)

            elseif ((tupleG1->dataType = regions) and (tupleG2->dataType = regions)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, RRINTERSECT, &CBNState, intersects, NULL)
            else :
                // Entity that is part of only one operand and of valid data type
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, SHAREINFO, &CBNState, intersects, NULL)
        else :
            // Entity that is not part of any operand
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, WAITIDLE, &CBNState, intersects, NULL)


POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true)
            if ((tupleG1->dataType = lines) and (tupleG2->dataType = lines)):
                if (CBNState = intersectingPoint):
                    stateOP = true
                else :
                    stateOP = false
            elseif ((tupleG1->dataType = regions) and (tupleG2->dataType = regions)):
                if (CBNState = true):
                    stateOP = true
            elseif ((tupleG1->dataType = lines) and (tupleG2->dataType = regions)) or
            ((tupleG1->dataType = regions) and (tupleG2->dataType = lines)):
                if (CBNState = intersectingPoint):
                    stateOP = true
                else :
                    stateOP = false
    Become(TASK_EVALUATING)
    EvaluateBooleanValuedTask()
```

Figure B.17: Protocol *Intersects*

using the *ConfirmCyclicConcatSubLists* procedure. The protocol returns **meetingPoint**, if the neighbours list is the cyclic concatenation of two sublists.

```
Procedure ComputeMeetingPointExistence()
isCmnSegExists = false
isMeetingPoint = false
if ((isGITReqPhase=false) and (isLocalComputationDone = false)):
    Set_Alarm(a_t2, LOCALCOMPUTATIONTIME+CLUTCOMPUTETXRXTIME)
    reqMsgcount = 0

    if (CBN =true):
        for sector = 1 to 12
            if (sectorNbrState[sector] = NBRCBN):
                validBSeg = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
                if (validBSeg= true):
                    isCmnSegExists = true
                    Break


        if (isCmnSegExists = true):
            opState = false
        elseif (isCmnSegExists = false):
            isMeetingPoint = ConfirmCyclicConcatSubLists(sectorNbrID[], sectorNbrState[])
            if (isMeetingPoint = true):
                opState = meetingPoint
            else :
                opState = intersectingPoint
    isLocalComputationDone = true
```

Figure B.18: Procedures used by *NeighbourLookUp* protocol to compute whether two **lines** values `meets` or `intersects`

B.2.1.11 Adjacent

This section, describes the protocol for the computation of the `adjacent` operation. The protocol for `adjacent` is shown in Figure B.20. Recall, from Section 4.2.1, that *Adjacent* takes operands of type **regions**. The operation computes whether operands are `area_disjoint` and have at least one CBS.

To test the existence of a common area or of a CBS, the `adjacent` protocol makes use of the *NeighbourLookUp* protocol. The procedures of the *NeighbourLookUp* protocol for computing common area and CBS are shown in Figure B.21, B.12, and B.16. The *NeighbourLookUp* protocol was described in Section B.2.1.6. After the completion of the GIT reply phase, the *NeighbourLookUp* protocol uses the *ComputeCommonAreaAndCBSExistence* procedure to compute whether the two **regions** values are `area_disjoint` and have a boundary segment in common. The *ChkForCommonNeighbours* procedure is called to confirm whether a CBN has any neighbours which belong to both operands. The description of the computation to test whether a CBN belongs to a common area is given in Section B.2.1.9. If a CBN is not part of a common area, it uses the *ConfirmCBS* procedure of the *NeighbourLookUp* protocol (as explained in Section B.2.1.6) to check whether it is part of a valid CBS. The *NeighbourLookUp* protocol returns state **commonBoundaryNode** if a CBN is not part of common area and not of CBS, **segment_node** if a CBN is not part of common area and part of CBS, and **false** otherwise.

Upon completion of the *NeighbourLookUp* protocol, a CBN sets its state as follows (line 45 in Figure B.20). The **segment_node** state denotes that the CBN is not part of a common area between two operands and is part of a valid CBS. The **false** state denotes that the CBN is part of a common area. The **disjoint** state denotes that the CBN is not part of a CBS nor of any

```
Procedure ComputeSegmentAreaInsideExistence()
isSegAI = false
isMeetingPoint = false
if ((isGITReqPhase=false) and (isLocalComputationDone = false)):
    Set_Alarm(a_t2, LOCALCOMPUTATIONTIME+CLUTCOMPUTETXRXTIME)
    reqMsgcount = 0
    if (CBN =true):
        isSegAI = ComputeSegIntersectRegion()
        if (isSegAI =true):
            opState = intersectingPoint
        else :
            isMeetingPoint = ConfirmCyclicConcatSubLists(sectorNbrID[], sectorNbrState[])
            if (isMeetingPoint = true):
                opState = meetingPoint
            else
                opState = false
    isLocalComputationDone = true
```

```
Procedure ComputeSegIntersectRegion()
validBSeg = false
isSegAI = false
for sector = 1 to 12
    if (sectorNbrState[sector] = NBRINTERSECT):
        validBSeg = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
        if (validBSeg= true):
            isSegAI = true
            Break
    if (sectorNbrState[sector] = NBRCBN):
        validBSeg = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
        if (validBSeg= true):
            isSegAI = true
            Break

return isSegAI
```

Figure B.19: Procedures used by *NeighbourLookUp* protocol to compute whether **lines** value `meets` or `intersects` **regions** value

common area.

B.2.1.12 Meets

This section, describes the protocol for the computation of the `meets` operation in Figure B.22. The procedures used by *NeighbourLookUp* protocol for computing the existence of a `meets` relationship between values of type **lines** are given in Figure B.18, and was discussed in Section B.2.1.10. A CBN sets its state to **true** upon receiving the state **meetingPoint** from the *NeighbourLookUp* protocol, otherwise it sets its state to **false**. The procedures used for computing the existence of a `meets` relationship between a **lines** value and a **regions** value are shown in Figure B.19, and was discussed in Section B.2.1.10. A CBN sets its state to **true** upon receiving the state **meetingPoint** from the *NeighbourLookUp* protocol, otherwise it sets its state to **false**.

For computing the existence of `meets` relationship between **regions** values, the *NeighbourLookUp* protocol uses the *ComputeCommonAreaAndCBSExistence* procedure in Figure B.21. *ComputeCommonAreaAndCBSExistence* procedure, and was discussed in Section B.2.1.11. A CBN sets its state to **commonBoundaryNode** upon receiving the state **commonBoundaryNode** from the *NeighbourLookUp* protocol, otherwise it sets its state to **false**.

```
Protocol Adjacent(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the adjacent operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed a operation state as
//              opNotSupported, NotPartOfOperands, disjoint, false or segment_node at time $t_f\}$

Status Values: $S$= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        stateOP = false
        CBN = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckAdjacentDTValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 = null) or (tupleG2 = null)):
                    stateOP = disjoint
                elseif ((tupleG1 != null)and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode = true)):
                        CBN = true
                    else :
                        stateOP = false
                else :
                    stateOP = OP_NOT_SUPPORTED

            if (isReqdToPartInNgbrLookup):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, CBSANDCMNAREA, &CBNState, adjacent, NULL)
            else :
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(NULL, NULL, CBSANDCMNAREA, &CBNState, adjacent, NULL)

POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true):
            if (CBNState = segment_node):
                stateOP = segment_node
            elseif (CBNState = commonBoundaryNode):
                stateOP = disjoint
            else :
                stateOP = false
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()
```

Figure B.20: Protocol *Adjacent*

```
Procedure ComputeCommonAreaAndCBSExistence()
CBSExists = 0
if ((isGITReqPhase= false)and (isLocalComputationDone = false) and (isCLUTComputationDone = true)):
    Set_Alarm(a_t2, LOCALCOMPUTATIONTIME)
    isLocalComputationDone = true
    if ((opState=false) and (CBN= true)and (IsCLUTPossible=true))
        isPartOfCLUT = PartOfValidCLUT()

    // CBN is part of common area
    if (isPartOfCLUT=true)
        opState = false

    // To compute CBS, as CBN is not part of common area
    if ((isPartOfCLUT=false) and (CBN= true))
        CBSExists = ConfirmCBS()
        if (CBSExists=CBS):
            opState = segment_node
        else :
            opState = commonBoundaryNode

elseif ((isGITReqPhase=false) and (isCLUTComputationDone = false)):
    reqMsgcount = 0
    Set_Alarm(a_t2, CLUTCOMPUTETXRXTIME)
    ChkForCommonNeighbours() // in Figure B.16
    if (IsCLUTPossible = true):
        ComputePossibleCLUTs(CLUT[])
        CLUT[] = EIT_ConfirmCLUTS(CLUT[])
        if (IsInvalidEdgeInfoInCLUTS()):
            SendInvalidCLUTInfo()
    isCLUTComputationDone = true
```

Figure B.21: Procedures used by *NeighbourLookUp* protocol to compute existence of CBS and common area

B.2.2 Aggregation

This section provide more details on the gossip-based scheme that underpins the selection of geometry-element leaders and the aggregation of local task states. The protocol is shown in Figure B.23. The process involves a number of rounds, in each of which an entity can be in one of seven states, viz., `IDLE`, `LEAD_PENDING`, `GROUP_LEAD`, `MEMBERSHIP_PENDING`, `GROUP_MEMBER`, `TENTATIVELY_CONVERGE`, `CONVERGE`. In the first round, group members declare themselves SEG leaders but, gradually, after successive rounds, as a result of the convergence properties of the algorithm, a single, final SEG leader and aggregated task state at the geometry-element level emerges and each node part of the geometry is aware of it. Initially, each entity is in the `IDLE` state. Once this state expires, the entity generates a random value that denotes its bid to be group leader. The *LeaderElection* procedure is responsible for generating a bid. If this bid value is greater than a probability threshold the entity moves to the `IDLE` state. A maximum limit has been set to handle the problem of consecutive retries an entity could make to become a `GROUP_LEAD`, in case if each time the computed bid value is greater than threshold and not hearing any `GROUP_LEAD` request from its neighbours. The purpose of adding this condition is two-fold: firstly, to give preference to an entity that is not receiving group leader requests from neighbouring entities as they are busy in the current round and is unable to poll a valid bid in the maximum consecutive number of retries, and, secondly, to handle the problem of delayed convergence, if most of the neighbouring entities have already converged, and the entity still has to complete certain number of rounds to move to `CONVERGE` state.

If the bid value is less than a probability threshold, the entity moves to the `LEAD_PENDING` state and broadcasts the *m_GLRQ* message. The group leader request message *m_GLRQ*,

```
Protocol Meets(gmtry1ID, gmtry2ID, stateOP)
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the meets operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed a operation state as
//             opNotSupported, NotPartOfOperands, true or commonBoundaryNode, disjoint, false at time $t_f$}

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        isOperandsDTValid = false
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        CBN = false
        stateOP = false
        CBNState = false
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = CheckMeetsDTValidity(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode = true)):
                        CBN = true
                    else :
                        stateOP = false
                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    stateOP = disjoint
                elseif (((tupleG1 != null) and (tupleG2 = null)):
                    stateOP = disjoint
            else :
                stateOP = OP_NOT_SUPPORTED
        if (isReqdToPartInNgbrLookup):
            if ((tupleG1->dataType = lines) and (tupleG2->dataType = lines)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LLMEETS, &CBNState, meets, NULL)

            elseif (((tupleG1->dataType = lines) and (tupleG2->dataType = regions)) or
            ((tupleG1->dataType = regions) and (tupleG2->dataType = lines))):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LRMEETS, &CBNState, meets, NULL)

            elseif ((tupleG1->dataType = regions) and (tupleG2->dataType = regions)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, CBSANDCMNAREA, &CBNState, meets, NULL)
            else :
                // Entity that is part of only one operand and of valid data type
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, SHAREINFO, &CBNState, meets, NULL)

        else :
            // Entity that is not part of any operand
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, WAITIDLE, &CBNState, meets, NULL)

POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true)
            if ((tupleG1->dataType = regions) and (tupleG2->dataType = regions)):
                if (CBNState = commonBoundaryNode):
                    stateOP = commonBoundaryNode
                else :
                    stateOP = false
            if ((tupleG1->dataType = lines) or (tupleG2->dataType = lines)):
                if (CBNState = meetingPoint):
                    stateOP = true

                else :
                    stateOP = false
        Become(TASK_EVALUATING)
        EvaluateBooleanValuedTask()

```

Figure B.22: Protocol *Meets*

conveys the source ID (*srcID*), and the geometry ID (*gmtryID*). Upon transmission of the request, the entity moves to the GROUP_LEAD state and waits, for a specified duration, for membership information. Upon receiving *m_GLRQ* message an entity in the IDLE state, moves to the MEMBERSHIP_PENDING state and transmits group member reply *m_GMRP* message, to the candidate group leader and moves to the GROUP_MEMBER state (line 54 in Figure B.23). In the first round, group members declare themselves SEG leaders.

The *m_GMRP* message conveys its partially aggregated results (*taskStateSEG*), and SEG leader information (*leaderSEG*). In the GROUP_MEMBER state, the entity waits for a specified duration for the reply from its group leader. Upon receiving reports from group members, the entity that happens to be in the GROUP_LEAD state aggregates (i.e., using BitwiseAND) the reported results and identifies the minimum entity ID between its own *leaderSEG* and the proposed *leaderSEG* received in the group members *m_GMRP* message (line 62 in Figure B.23). Upon expiry of the wait period for members to report, the group leader entity broadcasts a reply *m_GLRP* message to its group members containing the computed aggregated result (*taskStateSEG*) and the updated SEG leader ID (*leaderSEG*). Upon receiving such a reply message from its group leader (line 83 in Figure B.23), an entity in the GROUP_MEMBER state aggregates its own outcome (*taskStateSEG*) with the outcome sent by the group leader and updates its (*leaderSEG*) to the minimum between its own (*leaderSEG*) value and the one received in the leader reply. As a stopping criterion for this iterative process, a technique is used that allows each entity to determine if convergence has occurred. For this purpose, upon receiving the required information and performing the required computation each entity in the GROUP_LEAD or the GROUP_MEMBER state monitors any change in its *taskStateSEG* and its *leaderSEG* at the end of every round. If these do not change at the end of round, an entity increments the parameter which keeps record of the number of consecutive rounds for which the parameters do not change, by one. If these do not change after a specified number of consecutive rounds, the entity moves to the TENTATIVELY_CONVERGE state. If, at the end of a round, any of the parameters (i.e., *taskStateSEG* or *leaderSEG*) changes, it initializes the round count to zero and moves to the IDLE state. Upon changing the state to TENTATIVELY_CONVERGE, it waits in this state for a specified duration. If it does not receive any *m_GLRQ* message from its neighbours during that time, the entity moves to the CONVERGE state when the waiting period expires, otherwise it moves to the MEMBERSHIP_PENDING state and participates in another round (line 106 in Figure B.23). As a result of the convergence properties of the algorithm, a single, final SEG leader and aggregated task state at the geometry element level emerges and each entity in the geometry becomes aware of it.

To address the risk of packet loss due to collision, wait period has been used. When the wait period expires, the entity changes state as described above even if it does not receive a packet for which it was waiting. The purpose is to differentiate the scenarios in which the GROUP_LEAD does not receive any membership message because of packet loss, or because neighbouring entities have converged, or because neighbouring entities are already in either the GROUP_LEAD or the GROUP_MEMBER states. In such a case, as there will be no change in GROUP_LEAD results, at the end of its wait for membership reports, incrementing the tentative convergence round count would impact on the successful convergence of the geometry and not incrementing could

```
Protocol Aggregation($gmtryID$, $*taskStateSEG$, $*leaderSEG$)
// $P_{INIT}$ ≡ {Entities in a geometry element with at least one operation state other than
//            **OP_not_applicable** participate in the aggregation of the task state at time $t_0$}
// $P_{FINAL}$ ≡ {Every entity in the geometry element knows who is the SEG leader and what
//            is the partial aggregated task state at time $t_f$}

Status Values: $S$= {IDLE, GROUP_LEAD, GROUP_MEMBER, MEMBERSHIP_PENDING, LEAD_PENDING,
TENTATIVELY_CONVERGE, CONVERGE }
$S_{START}$= {IDLE }
$S_{INTERMEDIATE}$= {IDLE, GROUP_LEAD, GROUP_MEMBER, MEMBERSHIP_PENDING, LEAD_PENDING,
TENTATIVELY_CONVERGE, CONVERGE }
$S_{TERM}$ = {TASK_EVALUATING}

IDLE
    Spontaneously
        Set_Alarm(a_t1, $IDLE\_TIMEOUT$)

// Idle period time out
    When(a_t1)
        $isGroupLeader$ = LeaderElection()
        $cntSuccessiveBidAttempts$ = $cntSuccessiveBidAttempts$ + 1
        if (($isGroupLeader$ = true) or ($cntSuccessiveBidAttempts > MAXBIDATTEMPT$)):
            $cntSuccessiveBidAttempts$ = 0
            Become(LEAD_PENDING)
        else :
            Become(IDLE)


    Receive(m_GLRQ)
        if ($gmtryID$ = m_GLRQ.$gmtryID$):
            Stop_Alarm(a_t1)
            $cntSuccessiveBidAttempts$ = 0
            Become(MEMBERSHIP_PENDING)

LEAD_PENDING
    Spontaneously
        $isMembersInfoRcvd$ = false
        // n.ID denotes entity ID
        $srcID$ = $n.ID$
        $txTime$ = ComputeAggMsgTxTime(LEAD_PENDING)
        Set_Alarm(a_t6, $txTime$)

    When(a_t6)
        Send(m_GLRQ, $srcID$, $gmtryID$) to N($n$)
        Become(GROUP_LEAD)

MEMBERSHIP_PENDING
    Spontaneously

        $txTime$ = ComputeAggMsgTxTime(MEMBERSHIP_PENDING)
        Set_Alarm(a_t5, $txTime$)

    When(a_t5)
        Send(m_GMRP, $*LeaderSEG$, $*taskStateSEG$) to $groupLeader$
        Become(GROUP_MEMBER)

GROUP_LEAD
    Spontaneously

        $txTime$ = ComputeAggMsgTxTime(GROUP_LEAD)
        Set_Alarm(a_t2,($LEADER\_TIMEOUT$+ $txTime$))

    Receive(m_GMRP)
        $*LeaderSEG$ = MinNodeID(m_GMRP.$leaderSEG$, $*LeaderSEG$)
        $*taskStateSEG$ = BitwiseAND($*taskStateSEG$, m_GMRP.$taskStateSEG$)
        $isMembersInfoRcvd$ = true

// Group Leader Time Out
    When(a_t2)
        $srcID$ = $n.ID$
        Send(m_GLRP, $srcID$, $*LeaderSEG$, $*taskStateSEG$) to N($n$)
        $isTentativeConverged$ = CheckConvergence($isMembersInfoRcvd$)
        if ($isTentativeConverged$):
            Become(TENTATIVELY_CONVERGE)
        else :
            Become(IDLE)

```

Figure B.23: Protocol *Aggregation*

```
GROUP_MEMBER
    Spontaneously

        $txTime$ = ComputeAggMsgTxTime(GROUP_MEMBER)
        Set_Alarm(a_t3,($MEMBER\_TIMEOUT +$ $txTime$))

    Receive(m_GLRP)
        if ($groupLeader$ = m_GLRP.$srcID$):
            Stop_Alarm(a_t3)
            $*LeaderSEG$ = MinNodeID(m_GLRP.$leaderSEG$, $*LeaderSEG$)
            $*taskStateSEG$ = BitwiseAND($*taskStateSEG$, m_GMRP.$taskStateSEG$)
            $isConverged$ = CheckConvergence($*LeaderSEG$, $*taskStateSEG$, true)
            if ($isConverged$):
                Become(TENTATIVELY_CONVERGE)
            else :
                Become(IDLE)

// Group Member Time Out
    When(a_t3)
        Become(IDLE)

TENTATIVELY_CONVERGE
    Spontaneously
        Set_Alarm(a_t4, $CONVERGENCE\_TIMEOUT$)

    Receive(m_GLRQ)
        if ($gmtryID$ = m_GLRQ.$gmtryID$):
            Stop_Alarm(a_t4)
            $cntSuccessiveBidAttempts$ = 0
            $groupLeader$ = m_GLRQ.$srcID$
            Become(MEMBERSHIP_PENDING)

// Tentative Convergence Time Out
    When(a_t4)
        Become(CONVERGE)

CONVERGE
    Spontaneously
        Become(TASK_EVALUATING)

```

Figure B.23 (continued)

Procedure CheckConvergence($isMembersInfoRcvd$)

```
if (($*taskStateSEG$ = $previousTaskStateSEG$) and ($*LeaderSEG$ = $previousLeaderSEG$)):
    if ($isMembersInfoRcvd$ = true):
        $equivCount$ = $equivCount$ +1
    else :
        $equivCount$ = $equivCount$ +0.4
else :
    $equivCount$ = 0
    $previousLeaderSEG$ = $*LeaderSEG$
    $previousTaskStateSEG$ = $*taskStateSEG$
if ($equivCount$ >= EQUAL_THRESHOLD):
    return true
else :
    return false
```

Procedure ComputeAggMsgTxTime($state$)

```
$txTimeOut$ = RandomValue.rand()
if ($state$ = MEMBERSHIP_PENDING):
    $delay\_var$ = MBR_TX_RANDOMNESS
else :
    $delay\_var$ = LDR_TX_RANDOMNESS
if ($n.ID \neq 0$):
    $txTimeOut$ = ($txTimeOut$ % $delay\_var$)
    $txTimeOut$ = ((($txTimeOut$* $n.ID$) % $delay\_var$)))
else :
    $txTimeOut$ = ($txTimeOut$ % $delay\_var$)
return $txTimeOut$
```

Figure B.24: Procedures used by *Aggregation* protocol

prevent an entity from converging (i.e., keep on trying). In order to handle such a situation, if the group leader entity does not receive any group membership message, it increments the tentative convergence round count by less than one (i.e., 0.4), instead by one as explained above (line 5 in Figure B.23).

B.2.3 Result Processing

Once the distributed evaluation sub-task is concluded, results are routed towards the *first-level leader* by the SEG leaders along the tree that was established during task dissemination. The result processing protocol is shown in Figure B.25.

```
Protocol Result Processing(taskState, gmtryID)
 1  // $P_{INIT}$ ≡ "Only SEG leaders has the m_RoutingInf message at time $t_0$ and will act as initiator"
 2  // $P_{FINAL}$ ≡ "Sink will receive m_RoutingResult message containing the final outcome by time $t_f$
 3
 4  Status Values: $S$= {SEGLEADER, AVAILABLE, FIRSTLEVELLEADER }
 5  $S_{START}$= {SEGLEADER}
 6  $S_{INIT}$= {SEGLEADER, AVAILABLE}
 7  $S_{INTERMEDIATE}$= {FIRSTLEVELLEADER }
 8  $S_{TERM}$= {AVAILABLE}
 9
10  SEGLEADER
11      Spontaneously
12          // n.ID denotes entity ID
13          srcID = n.ID
14          destID = n.ID
15          msgRxCount = 0
16          retransmitAttempt = 0
17          isFirstLvlLeader = TaskDissemination_FirstLevelLdr()
18          if (isFirstLvlLeader = false):
19              txTime = ComputeRouteMsgTxTime()
20              Set_Alarm(a_t11, txTime)
21              cntRcvdDistinctGmtries = ComputeCntRcvdGmtries(m_RoutingInf.gmtryID)
22          else :
23              Become(FIRSTLEVELLEADER)
24
25      When(a_t11)
26          destID = TaskDissemination_getParentID()
27          Send(m_RoutingInf, srcID, taskState) to destID
28          retransmitAttempt = retransmitAttempt + 1
29          if (retransmitAttempt < RETRYATTEMPTS):
30              txTime = ComputeWaitAckTime()
31              Set_Alarm(a_t11, txTime)
32          Become(AVAILABLE)
33
34      Receive(m_RoutingAck)
35          Stop_Alarm(a_t11)
36
37  AVAILABLE
38      When(a_t11)
39          destID = TaskDissemination_getParentID()
40          if (msgID = m_RoutingInf):
41              Send(m_RoutingInf, srcID, taskState) to destID
42          elseif (msgID = m_RoutingResult):
43              Send(m_RoutingResult, srcID, taskState) to destID
44          retransmitAttempt = retransmitAttempt + 1
45          if (retransmitAttempt < RETRYATTEMPTS)
46              txTime = ComputeWaitAckTime()
47              Set_Alarm(a_t11, txTime)
48
```

Figure B.25: Protocol *Result Processing*

A SEG leader first computes whether it is a first-level leader (line 17 in Figure B.25). If it is not, it transmits a *m_RoutingInf* message towards its parent entity (which has been determined during task dissemination) and sets a wait timer for the reception of an acknowledgement. The *m_RoutingInf* message conveys source ID (*srcID*), geometry ID (*gmtryID*) and the partial

```
    When(a_t13)
        Send(m_RoutingAck, srcID) to childID
        if (isFirstLvlLeader = false):
            txTime = ComputeRouteMsgTxTime()
            Set_Alarm(a_t11, txTime)

    Receive(m_RoutingAck)
        Stop_Alarm(a_t11)

    Receive(m_RoutingInf)
        childID = m_RoutingInf.srcID
        isFirstLvlLeader = TaskDissemination_FirstLevelLdr()
        msgID = m_RoutingInf
        if (isFirstLvlLeader = true):
            Become(FIRSTLEVELLEADER)
            cntRcvdDistinctGmtries = ComputeCntRcvdGmtries(m_RoutingInf.gmtryID)
        txTime = ComputeRouteACKMsgTxTime()
        Set_Alarm(a_t13, txTime)

    Receive(m_RoutingResult)
        childID = m_RoutingResult.srcID
        msgID = m_RoutingResult
        txTime = ComputeRouteACKMsgTxTime()
        Set_Alarm(a_t13, txTime)

FIRSTLEVELLEADER
    Spontaneously
        srcID = n.ID
        msgRxCount = 1
        expDistinctGmtries = ComputeExpDistinctGmtries(postfixtask)
        timeToWait = ComputeWaitTimeForSEGs(expDistinctGmtries, cntRcvdDistinctGmtries)
        Set_Alarm(a_t12, timeToWait)

    When(a_t11)
        destID = TaskDissemination_getParentID()
        Send(m_RoutingResult, srcID, taskResult, gmtryID) to destID
        retransmitAttempt = retransmitAttempt + 1
        if (retransmitAttempt < RETRYATTEMPTS)
            txTime = ComputeWaitAckTime()
            Set_Alarm(a_t11, txTime)
        else :
            Become(AVAILABLE)

    When(a_t13)
        Send(m_RoutingAck, srcID) to childID

    When(a_t12)
        if (cntRcvdDistinctGmtries < expDistinctGmtries)
            Reset_Alarm(a_t12, DELAYWAIT)
            cntRcvdDistinctGmtries = expDistinctGmtries

        opStates[] = DecompressTaskState()
        taskResult = ApplyBooleanConnectives(opStates[])
        txTime = ComputeRouteMsgTxTime()
        Set_Alarm(a_t11, txTime)

    Receive(m_RoutingAck)
        Stop_Alarm(a_t11)
        Become(AVAILABLE)

    Receive(m_RoutingInf)
        childID = m_RoutingInf.srcID
        cntRcvdDistinctGmtries = ComputeCntRcvdGmtries(m_RoutingInf.gmtryID)
        msgRxCount = msgRxCount + 1
        timeToWait = ComputeWaitTimeForSEGs(expDistinctGmtries, cntRcvdDistinctGmtries)
        Reset_Alarm(a_t12, timeToWait)
        if ((msgRxCount > 1) and (isFirstLvlLeader)):
            taskState = BitwiseAND(taskState, m_RoutingInf.taskState)
        elseif ((msgRxCount <= 1) and (isFirstLvlLeader)):
            taskState = m_RoutingInf.taskState
        txTime = ComputeRouteACKMsgTxTime()
        Set_Alarm(a_t13, txTime)

```

Figure B.25 (continued)

aggregated task state (*taskState*). Upon receiving this information, the parent entity sends the acknowledgement and forwards the message towards its own parent unless it itself is the first-level leader. If a child sees the wait period expire without receiving an acknowledgement, it retransmits the message towards the parent.

Upon finding itself the first-level leader, an entity moves to the `FIRSTLEVELLEADER` state and waits for certain amount of time for the task states from other SEG leaders (more than one in case of MEGs). The first-level leader then aggregates the task states from the SEG leaders. Upon the expiry of the wait period, the first-level leader decompresses the finally aggregated task state and computes the operation states (line 99 in Figure B.25). These operation states represent the final result for each spatial predicate in the task. The next step is to apply the Boolean connectives (i.e., **not**, **and** and **or**) that occur in the task. For this purpose, each spatial predicate in the task is replaced by the corresponding operation state and the Boolean connectives are applied to compute the final outcome. The *ApplyBooleanConnectives* procedure is responsible for applying the Boolean connectives to the operation states. The final outcome, is then transmitted by the first-level leader towards the sink in the *m_RoutingResult* message. The *m_RoutingResult* message conveys the source ID (*srcID*) and final outcome (*taskResult*).

B.3 Spatial-Valued Tasks

Recall, that the spatial-valued task evaluation process comprises two phases, viz., distributed task evaluation, and result processing. This section, discuss the *EvaluateSpatialValuedTask* protocol. It also uses a stack-based approach. The protocol supports re-evaluation of tasks up to a fixed number of times. Each operation in a task is evaluated in turn. An operand is pushed onto the operand stack. A binary operator causes two operands to be popped from the stack; a unary operator, causes one operand to be popped from the stack. After the evaluation of each operation in a complex spatial-valued task, the resulting derived geometry is referred to as a *temporary intermediate derived geometry.* For each operation, the corresponding protocol is called by MBR entities. An entity computes the ID of the temporary intermediate derived geometry using the *SelectPartialDrvdGmtryID* procedure. This procedure checks which of the temporary geometry IDs (i.e., *TEMPGMTRY1* or *TEMPGMTRY2*) is already on the operand stack and selects accordingly. At the end of each operation evaluation, some of the entities will have set the event state to **false** and some to **true** indicating membership in the derived geometry. The entities that satisfy the success criteria of the operation set their event state to **true** and compute the other attributes of the geometry including its type, its boundary membership, and the *time-to-live* (TTL) of the derived geometry. This information is returned to the *EvaluateSpatialValuedTask* protocol. After each operation, each entity stores the temporary geometry ID on their operand stack and the entities with **true** state store the information about the temporary geometry in their GIT temporarily. The procedure *GIT_InsertTuple* is responsible for inserting the tuple in the GIT if it does not already exists. If the tuple already exists for that geometry, its attributes are reset.

Upon the evaluation of each operator in which one or both operands are temporary geometries, all entities calls the *ProcessPartialDrvdGmtryInfo* procedure to remove the relevant entry from GIT and EIT if their event state is **false**. Nodes with a **true** event state also update the attributes TTL, *spatialdatatype* and *boundarynode* about temporary geometry in their GIT.

Upon evaluation of the last operation in the task, the entities with a **true** event state update the GIT and associated entries in EIT for *intermediate derived geometry* with the finally derived geometry ID. This section describes the protocol for the operator, `minus`. The protocols for the remaining operators (e.g., `plus`, `intersection`, `common_border`, `vertices`) are provided in Appendix B.

B.3.1 Minus

The `minus` protocol is shown in Figure B.28. If the operands are two **points** values, the entities can compute the event state based on the information in the local GIT. If the operands are two **lines** values, the entities that belong to first operand only set their event state to **true** based on the local information in their GIT but the CBNs need to find out whether they are part of a segment in which one of the **points** value belongs to first operand only (as the `minus` operation returns a value of type **lines** in this case). For this purpose, the entities call the *NeighbourLookUp* protocol in Figure B.10 with a request to compute the difference between the **lines** values (line 51 in Figure B.28). After collecting information from its neighbours, if one of its valid segment neighbours belongs to the first operand only, the CBN sets its state to **true** (line 6 in Figure B.29).

In the case of **regions**, the entities that belong to the first operand only or the entities that are part of the interior of the first and the boundary of the second operand sets their operation state to **true**. A CBN needs information from its neighbours to compute its operation state as well as information about invalid edges. An entity that belongs to the interior of the first and to the boundary of the second operand also needs information from its neighbours to compute its EIT information. Therefore, in the *NeighbourLookUp* protocol, the condition is added (line 39 in Figure B.10) that along with the CBNs all entities that belong to the interior of the first and the boundary of the second operand must also request information from their neighbours.

The procedures used by the *NeighbourLookUp* protocol to compute the difference between two **regions** values are given in Figure B.30 and B.16. After collection of information from its neighbours, a CBN checks whether it is part of any CLUT formed with neighbour CBNs(line 38 in *ConfirmPartOfDerivedRegion* procedure in Figure B.30). If a CBN is not part of a CLUT with two other CBNs and if it has at least one neighbour that belongs to the first operand only amongst its closest neighbours, the CBN sets its state to **true** (line 6 in Figure B.30). If the CBN forms a CLUT with two other CBNs and has at least one neighbour that belongs to the first operand only such that the segment between the CBN and that neighbour does not intersect any CLUT that the CBN belongs to, it sets its state to **true**, otherwise it sets it to **false**. Entities with a **true** operation state, that are CBNs or belong to the interior of first operand and to the boundary of the second operand or belong to the first operand only and to its boundary set the boundary state information for the derived geometry to **true** (line 38 in Figure B.28). Entities that belong to the interior of the first operand set their boundary state information to **false**.

An entity that is part of the first operand only needs to perform a local EIT lookup to find

```
Protocol EvalSpatialValuedTask(task* postfixTask, reEvalPeriod, derivedGmtryID, duration)
// $P_{INIT}$ ≡ "All entities in the task MBR are ready to evaluate the spatial-valued task message at time $t_0$" ≡
//             {∀ n ∈ N: n ∈ MBR ⇒ n can evaluate the task message at time $t_0$}
// $P_{FINAL}$ ≡ "Nodes in the new derived geometry have stored the information about the geometry in their GIT by time $t_f$"

Status Values: S= {TASK_EVALUATING, TASK_PREPROCESSING, POST_SPATIALOPERATIONEVALUATION, AVAILABLE}
$S_{START}$= {TASK_PREPROCESSING}
$S_{INTERMEDIATE}$= {TASK_EVALUATING, POST_SPATIALOPERATIONEVALUATION}
$S_{TERM}$ = {AVAILABLE}

TASK_PREPROCESSING
    Spontaneously
        Stack OPStack, taskStack
        eventNode = false
        rightArg= InvalidGmtry
        leftArg= InvalidGmtry
        // tmpGmtry contains attributes gmtryID, stateOP, isGmtryMbr, isBndryNode, gmtryTTL, gmtryDataType
        Struct TempGmtryRec tmpGmtry

        InitializeTmpGmtry(tmpGmtry)
        PushTaskOnStack(postfixTask, taskStack)

        Set_Alarm(a_t19, reEvalPeriod)
        Become(TASK_EVALUATING)

TASK_EVALUATING
    Spontaneously

        while(not taskStack_Empty()):
            op = taskStack_Pop()
            if (op=plus):
                rightArg= OPStack_Pop()
                leftArg= OPStack_Pop()
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                PlusOperator(leftArg, rightArg, &tmpGmtry)
            elseif (op=intersection):
                rightArg = OPStack_Pop()
                leftArg = OPStack_Pop()
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                IntersectionOperator(leftArg, rightArg, &tmpGmtry)
            elseif (op= minus) :
                rightArg= OPStack_Pop()
                leftArg= OPStack_Pop()
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                MinusOperator(leftArg, rightArg, &tmpGmtry)
            elseif (op= vertices):
                leftArg = OPStack_Pop()
                rightArg = INVALIDGMTRY
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                VerticesOperator(leftArg, &tmpGmtry)
            elseif (op= common_border):
                rightArg= OPStack_Pop()
                leftArg = OPStack_Pop()
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                CommonBorderOperator(leftArg, rightArg, &tmpGmtry)
            elseif (op= contour):
                leftArg = OPStack_Pop()
                rightArg = INVALIDGMTRY
                tmpGmtry.gmtryID = SelectPartialDrvdGmtryID()
                Become(OPERATION_EVALUATING)
                Contour(leftArg, &tmpGmtry)
            else :
                    OPStack_Push(op)

        if (eventNode = true):
            GIT_RenameGID(tmpGmtry.gmtryID, derivedGmtryID)
            EIT_RenameGID(tmpGmtry.gmtryID, derivedGmtryID)
            Become(AVAILABLE)

```

Figure B.26: Protocol *EvalSpatialValueTask*

```
POST_SPATIALOPERATIONEVALUATION
    Spontaneously
        ProcessPartialDrvdGmtryInfo()
        Become(TASK_EVALUATING)

AVAILABLE
    When(a_t19)
        evalPeriod = ComputeReEvalPeriod(evalPeriod, reEvalPeriod)
        if (evalPeriod <= duration)
            Become(TASK_PREPROCESSING)
        else :
            Become(AVAILABLE)

```

Figure B.26 (continued)

```
Procedure ComputeOprndPartOfTmpGmtry()
tempGmtryToRmve = INVALIDGMTRY

if ((leftArg = TEMPGMTRY1) and (leftArg = TEMPGMTRY2)) or
((rightArg = TEMPGMTRY2) and (rightArg = TEMPGMTRY1))):
    tempGmtryToRmve = BOTHTEMPGMTRY

elseif ((leftArg = TEMPGMTRY1) or (rightArg = TEMPGMTRY1)):
    tempGmtryToRmve = TEMPGMTRY1

elseif ((leftArg = TEMPGMTRY2) or (rightArg = TEMPGMTRY2)):
    tempGmtryToRmve = TEMPGMTRY2

return tempGmtryToRmve
```

```
Procedure ProcessPartialDrvdGmtryInfo()
if (tmpGmtry.gmtryMbrshp = true):
    eventNode = true
else :
    eventNode = false
OPStack_Push(tmpGmtry.gmtryID)

if (eventNode = false)
    tmpGmtryToRmve = ComputeOprndPartOfTmpGmtry(leftArg, rightArg)

    if (tmpGmtryToRmve = BOTHTEMPGMTRY):
        GIT_RemoveTuples(TEMPGMTRY1, TEMPGMTRY2)
        EIT_RemoveTuples(TEMPGMTRY1, TEMPGMTRY2)

    elseif (tmpGmtryToRmve != INVALIDGMTRY):
        GIT_RemoveTuple(tmpGmtryToRmve)
        EIT_RemoveTuple(tmpGmtryToRmve)
if (eventNode = true):
    if (tmpGmtryToRmve = BOTHTEMPGMTRY)
        GIT_RemoveTuple(TEMPGMTRY2)
        EIT_RemoveTuple(TEMPGMTRY2)
        // ComputeGITTuple procedure is responsible for computing the GIT tuple from the tmpGmtry tuple.
        GIT_UpdateTuple(ComputeGITTuple(tmpGmtry))
    elseif (tmpGmtryToRmve != INVALIDGMTRY)
        GIT_RemoveTuple(tmpGmtryToRmve)
        GIT_InsertTuple(ComputeGITTuple(tmpGmtry))
    else
        GIT_InsertTuple(ComputeGITTuple(tmpGmtry))
    InitializeTmpGmtry(tmpGmtry)
```

Figure B.27: Procedures used by *EvalSpatialValuedTask* protocol

any tuple belonging to the first operand. If it finds one or more tuples, it appends the same information to the EIT for the derived geometry by updating the GID. The *EIT_UpdateInfMinusOP* procedure (line 22 of *ComputeNodePartOfDerivedRegion* procedure in Figure B.30) is responsible for such computation. Recall, from Section 5.4.2, that a tuple in EIT comprises two attributes, viz., *GID*, and *neighbourID*.

The *EIT_UpdateInfMinusOP* procedure is also responsible for the computation of information about invalid edges for a CBN or an entity that belongs to the interior of the first and to the boundary of the second operand and belongs to the derived geometry. In such cases, the entity, first checks whether it has neighbouring entities that lie in sectors 7-12 and are CBNs or belong to the interior of the first and to the boundary of the second operand. If it finds any, it computes the information about invalid edges that it needs to share with its neighbours.

Recall from Section 5.5.2.1, that only entities that form an invalid edge with neighbours that are CBNs or belongs to the interior of the first operand and to the boundary of the other and that lies in sectors 7 to 12 needs to transmit such information in an *m_InvalidEITInf* message. All entities that belong to the derived geometry update their EIT table upon reception of the *m_InvalidEITInf* message. The *EIT_RemoveEdgesMinusOP* procedure (line 162 in Figure B.10) is responsible for such computation.

B.3.2 Plus

The operator `plus` returns the union of two values of the same spatial type. If an MBR entity belongs to one or both operands, it declares itself part of the derived geometry (i.e., it is an event entity) by setting its operation state to **true**. All the entities that belong to only one operand and belong to a boundary set the boundary state their **true**. All CBNs need information from their neighbours in order to compute their boundary state. For that purpose, the MBR entities use the *DeriveGeometry* protocol, which implements the modified T-Fit algorithm discussed in Section 3.4.2.

After the boundary computation, the boundary entities compute the information to be added to their EIT for the derived geometry (using the *EIT_UpdateInfPlusOP* procedure). The entities need to perform a local-lookup of their EIT for that purpose. In the case of the `plus` operation, a boundary entity that belongs to only one of the operands checks that operand's entry in its EIT. If an entity finds any information about the operand, it copies this information to its EIT after updating the tuple with the GID of the derived geometry. If it is a CBN, it checks the tuples in its EIT that are common to both operands and copies those tuples to its EIT after updating the tuple with the GID of the derived geometry.

B.3.3 Intersection

The `intersection` protocol is shown in Figure B.34. In the case of `intersection`, all non-CBNs can compute their event state based on the information in the local GIT. The CBNs need information from their neighbours. For this purpose, if both operands are of type **lines**, the MBR entities use the *NeighbourLookUp* protocol with a request to compute whether the lines intersect (line 50 in Figure B.34). The procedures used by *NeighbourLookUp* protocol

```
Protocol Minus(gmtry1ID, gmtry2ID, Struct TempGmtryRec * tmpGmtry)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the minus operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed whether it is
//                part of new derived geometry and all of its associated attributes at time $t_f\}$

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isOperandsDTValid = false
        tmpGmtry→stateOP = OPDONE
        CBN = false
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            tmpGmtry→stateOP = NOT_PART_OF_OPERANDS
        else
            isOperandsDTValid = OperandsDTValidMinus(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                tmpGmtry→gmtryDataType = DrvdGmtryOprndTypeMinus(tupleG1, tupleG2)

                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode = true)):
                        if ((tupleG1→dataType != points) and (tupleG2→dataType != points)):
                            CBN = true
                    elseif ((tupleG1→bndryNode = false) and (tupleG2→bndryNode = true)):
                        tmpGmtry→isGmtryMbr = true
                        tmpGmtry→isBndryNode = true
                        tmpGmtry→gmtryTTL = tupleG1→TTL

                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    tmpGmtry→isGmtryMbr = true
                    if ((tupleG1→bndryNode = true)):
                        tmpGmtry→isBndryNode = true
                    tmpGmtry→gmtryTTL = tupleG1→TTL
            else :
                tmpGmtry→stateOP = OP_NOT_SUPPORTED

        if ((tupleG1→dataType = points) or (tupleG2→dataType = points)):
            isReqdToPartInNgbrLookup = false

        if (isReqdToPartInNgbrLookup):
            if ((tupleG1→dataType = lines) or (tupleG2→dataType = lines)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LLMINUS, &CBNState,minus, tmpGmtry→gmtryID)
            elseif ((tupleG1→dataType = regions) or (tupleG2→dataType = regions)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, RRMINUS, &CBNState,minus, tmpGmtry→gmtryID)
            else :
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, SHAREINFO-DG, &CBNState,minus, tmpGmtry→gmtryID)
        else :
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, WAITIDLE-DG, &CBNState,minus, tmpGmtry→gmtryID)

POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true):
            if (CBNState = true):
                tmpGmtry→isGmtryMbr = true
                tmpGmtry→isBndryNode = true
        Become(POST_SPATIALOPERATIONEVALUATION)
        EvaluateSpatialValuedTask()
```

Figure B.28: Protocol *Minus*

```
Procedure ComputePartOfDerivedline()
if (isGITReqPhase= false):
    validBSegG1 = false
    Set_Alarm(a_t2, EITCOMPUTETXRXTIME+CLUTCOMPUTETXRXTIME)

    for sector = 1 to 12
        if (sectorNbrState[sector] = GIDG1):
            validBSegG1 = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
            if (validBSegG1 = true):
                opState = true
            else :
                opState = false
        isLocalComputationDone = true
```

Figure B.29: Procedures used by *Minus* protocol to compute the difference between two **lines** values

```
Procedure ComputeNodePartOfDerivedRegion()
if ((isGITReqPhase= false) and (isCLUTComputationDone = true)):
    Set_Alarm(a_t2, EITCOMPUTETXRXTIME)
    isLocalComputationDone = true
    isEITComputationDone = true

    if ((IsCLUTPossible = false) and (ngbrInG1Exists = true)):
        stateOP = true
    elseif ((IsCLUTPossible = true) and (ngbrInG1Exists = true)):
        for sector = 1 to 12
            if (sectorNbrState[sector] = GIDG1):
                validBSeg = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
                if (validBSeg = true):
                    stateOP = ConfirmSegIntersectsAnyCLUT(sectorNbrID[sector])
                    if (stateOP = false):
                        stateOP = true
                        Break
                    else :
                        stateOP = false

    if ((stateOP = true) or ((tupleG1 !=NULL) and (tupleG2 =NULL)) or
    ((tupleG1->bndryNode = false) and (tupleG2->bndryNode = true)))
        InvalidNgbrInf[] = EIT_UpdateInfMinusOP(tmpGmtry, tupleG1, tupleG2, sectorNbrID, sectorNbrState)

    if (IsInvalidEdgeInfoExists(InvalidNgbrInf[]):
        SendInvalidEdgeInfo()

elseif (isGITReqPhase=false):
    Set_Alarm(a_t2, CLUTCOMPUTETXRXTIME)
    for sector = 1 to 12
        if ((sectorNbrState[sector] = GIDG1)) :
            validBSeg = EIT_ConfirmSegment(sectorNbrID[sector], tupleG1, tupleG2)
            if (validBSeg = true):
                ngbrInG1Exists = true

        if (sectorNbrState[sector] = NBRCBN):
            cntNgbrCBN = cntNgbrCBN + 1

    if (cntNgbrCBN >= 2):
        ComputePossibleCLUTs()
        CLUT[] = EIT_ConfirmCLUTS(CLUT[])
        if (IsInvalidEdgeInfoInCLUTS()):
            SendInvalidCLUTInfo()
    isCLUTComputationDone = true
```

```
Procedure SendInvalidEITInfo()
txTime = ComputeGITTxTime(m_InvalidEITInf)
Set_Alarm(a_t6, txTime)
```

Figure B.30: Procedures used by *Minus* protocol to compute the difference between two **regions** values

```
Protocol Plus(gmtry1ID, gmtry2ID, Struct TempGmtryRec * tmpGmtry)
// $P_{INIT} \equiv$ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the plus operation at time $t_0$}
// $P_{FINAL} \equiv$ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed whether it
//                 is part of new derived geometry and all of its associated attributes at time $t_f$}

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_GEOMETRYDERIVATION}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isOperandsDTValid = 0
        isReqdToPartInNgbrLookup = false
        tmpGmtry→stateOP = OPDONE
        CBN = false

        if ((tupleG1= null) and (tupleG2= null)):
            tmpGmtry→stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = OperandsDTValidPlus(tupleG1, tupleG2)

            if (isOperandsDTValid):
                tmpGmtry→gmtryDataType = DrvdGmtryOprndTypePlus(tupleG1, tupleG2)
                tmpGmtry→isGmtryMbr = true
                isReqdToPartInNgbrLookup = true

                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode = true)):
                        CBN = true
                    tmpGmtry→gmtryTTL = MinimumTTL(tupleG1→TTL, tupleG2→TTL)

                elseif ((tupleG1 = null) and (tupleG2 != null)):
                    if (tupleG2→bndryNode = true):
                        tmpGmtry→isBndryNode = true
                    tmpGmtry→gmtryTTL = tupleG2→TTL

                elseif ((tupleG1 != null) and (tupleG2 = null)):
                    if (tupleG1→bndryNode = true):
                        tmpGmtry→isBndryNode = true
                    tmpGmtry→gmtryTTL = tupleG1→TTL
            else :
                tmpGmtry→stateOP = OP_NOT_SUPPORTED
                tmpGmtry→isGmtryMbr = false

        if ((tupleG1→dataType = points) or (tupleG2→dataType = points)
        or (tupleG1→dataType = lines) or (tupleG2→dataType = lines)):
            isReqdToPartInNgbrLookup = false

        if (isReqdToPartInNgbrLookup)
            Become(OPERATION_EVALUATING)
            DeriveGeometry(tmpGmtry→gmtryID, tmpGmtry→isGmtryMbr, CBN, &bndryNode, PLUS)
        else
            Become(OPERATION_EVALUATING)
            DeriveGeometry(NULL, false, false, &bndryNode, PLUS)

POST_GEOMETRYDERIVATION
    Spontaneously
        if ((CBN = true) and (tmpGmtry→gmtryDataType =regions)):
            tmpGmtry→isBndryNode = bndryNode
        elseif ((CBN = true) and (tmpGmtry→gmtryDataType !=regions)):
            tmpGmtry→isBndryNode = true

        EIT_UpdateInfPlusOP(tmpGmtry, gmtry1ID, gmtry2ID)

        Become(POST_SPATIALOPERATIONEVALUATION)
        EvaluateSpatialValuedTask()

```

Figure B.31: Protocol *Plus*

```
Protocol DeriveGeometry(GIDG1, *eventNode, CBN, *operationState, operation)
Status Values: S= {OPERATION_EVALUATING, IDLE, POST_GEOMETRYDERIVATION}
S_START= {OPERATION_EVALUATING}
S_TERM= {POST_GEOMETRYDERIVATION}
OPERATION_EVALUATING
    Spontaneously
        DrvdGmtryRq = true
        reqMsgcount = 0
        isLocalComputationDone = false
        opState = false
        isEventNode = *eventNode

        if (operation = plus):
            if (CBN = true):
                txTime = ComputeDrvdGmtryTxTime(m_DRVDRQ)
                Set_Alarm(a_t1, txTime)
                Set_Alarm(a_t2, WAITDGREQ)
            elseif ((isEventNode = false) and (CBN = false)):
                Become(IDLE)
            elseif ((isEventNode = true) and (CBN = false)):
                Set_Alarm(a_t2, WAITDGREQ)
        if (operation = IG):
            reqMsgcount= 1
            Set_Alarm(a_t2, SENDMSG)

    Receive(m_DRVDRQ)
        if (GIDG1 = m_DRVDRQ.GIDG1):
            reqMsgCount = reqMsgCount + 1

// Time to transmit m_DRVDRQ message
    When(a_t1)
        Send(m_DRVDRQ, n.ID, GIDG1,isEventNode) to N(n)

// Time to transmit GIT Reply message and to perform local computation based on
// type of computation request and to return control to calling protocol
    When(a_t2)
        if (DrvdGmtryRq=true):  // GIT-lookup Reply phase
            DrvdGmtryRq = false
            Set_Alarm(a_t2, WAITDGREPLY)
            if (reqMsgcount > MAXREPLY):
                reqMsgcount= MAXREPLY
            if (reqMsgcount > 0):
                SendDrvdGmtryReply()
            // Local Computation after completion of GIT-lookup Reply phase
        elseif ((DrvdGmtryRq= false) and (isLocalComputationDone = true)):
            *operationState = opState
            *eventNode = isEventNode
            Become(POST_GEOMETRYDERIVATION)
            if (operation = plus)
                Plus()
            elseif (operation = IG)
                EvalInduced()
            // Finish evaluation and return control
        elseif ((DrvdGmtryRq=false)and (isLocalComputationDone = false)):
            reqMsgcount = 0
            Set_Alarm(a_t2, LOCALCOMPUTATIONTIME)
            if ((operation = plus) and (CBN = true)):
                opState = ConfirmBNState()
            elseif (operation = IG):
                opState = ConfirmInducedGmtryBN()
            isLocalComputationDone = true

// Time to transmit m_DRVDGMRYREPLY message
    When(a_t3)
        Send(m_DRVDGMRYREPLY, n.ID, GIDG1,isEventNode) to N(n)
        if (reqMsgcount < MAXREPLY):
            SendDrvdGmtryReply()
```

Figure B.32: Protocol *DeriveGeometry*

```
    Receive(m_DRVDGMRYREPLY)
        if (CBN = true):
            if (GIDG1 = m_DRVDGMRYREPLY.GIDG1):
                sector = NIT_ComputeSector(m_DRVDGMRYREPLY.sourceID)
                if (sectorNbrID[sector]= EMPTY):
                    sectorNbrID[sector] = m_DRVDGMRYREPLY.sourceID
                    sectorNbrState[sector] = m_DRVDGMRYREPLY.isEventNode
                else :
                    distance = NIT_getDistanceToMe(m_DRVDGMRYREPLY.sourceID)
                    prvNgbrDistance = NIT_getDistanceToMe(sectorNbrID[sector])
                    if (distance < prvNgbrDistance):
                        sectorNbrID[sector] = m_DRVDGMRYREPLY.sourceID
                        sectorNbrState[sector] = m_DRVDGMRYREPLY.isEventNode

// Idle wait period Expires
    When(a_t5)
        isGITReqPhase = false
        Set_Alarm(a_t2, FIRET2)

IDLE
    Spontaneously
        Set_Alarm(a_t5, WAITDGREQ + WAITDGREPLY)

```

Figure B.32 (continued)

for computing `intersection` between values of type **lines** are given in Figure B.18. A CBN sets its event state to **true** if the protocol returns **intersectingPoint**. The procedures used for computing `intersection` between **lines** and **regions** values are shown in Figure B.19. The entity sets its event state to **true** if **intersectingPoint** results from execution of the protocol. If both the operands are of type **regions**, the MBR entities use the *NeighbourLookUp* protocol with a request to test the existence of common area. The entity sets its event state to **true** if the protocol returns **true**.

The entities belonging to the boundary of a derived geometry need to compute the information to be added to their EIT. For this purpose, the entities need information from the neighbours that belong to the boundary of a derived geometry. Therefore, in the *NeighbourLookUp* protocol the condition is added (line 31 in Figure B.10) that, along with the CBNs, all entities that belong to both operands and to the interior of one operand and boundary of other should also request information from their neighbours. The *EIT_UpdateInfIntersectionOP* procedure is responsible for computing and adding such information in EIT (line 9 called from *ComputeCommonAreaExistence* procedure in Figure B.16).

B.3.4 Vertices

The `vertices` operation returns a new spatial-valued geometry of type **points**. It is a unary operation and all entities that belong to the boundary of the **regions** value passed as input set their operation state to **true**, other entities sets the operation state to **false**.

B.3.5 CommonBorder

The `common_border` operation creates a new spatial value of type **lines** containing the common segments of their operands of type **lines** or **regions** or a combination of **lines** and **regions** value. For this purpose, the MBR entities uses the *NeighbourLookUp* protocol with a request

```
Procedure ComputeDrvdGmtryTxTime(txType)
delayVariance = 0
txTimeOut = RandomValue.rand()
if ((txType = m_DRVDRQ)):
    delayVariance= DRVDRQ_DELAY_VAR
else :
    delayVariance= DRVDRP_DELAY_VAR
if ($n.ID \neq 0$):
    txTimeOut=(txTimeOut % delayVariance)
    txTimeOut=(((txTimeOut* n.ID) % delayVariance)))
else :
    txTimeOut= (txTimeOut % delayVariance)
if (txType = m_DRVDRQ)
    txTimeOut = txTimeOut + MIN_DELAY
return txTimeOut
```

```
Procedure ConfirmBNState()
stateOP = false
quadrants[] = ComputeNgbrsInEachQuadrant(sectorNbrID[sector])
axes[] = ComputeNgbrsOnEachAxes(sectorNbrID[sector])
noOfQuadsWithNbrs = computeQuadsWithNgbrs(quadrants[])
noOfAxesWithNbrs = computeAxesWithNgbrs(axes[])

if (noOfQuadsWithNbrs= 4) :
    stateOP = false
elseif (noOfQuadsWithNbrs= 1) :
    stateOP = true
elseif (noOfQuadsWithNbrs= 2) :
    angle = angleBetweenNbgrs(quadrants[])
    if ($angle < 180$) :
        stateOP = true
    else :
        stateOP = false

elseif (noOfQuadsWithNbrs= 3) :
    angle = angleBetweenNgbrsInDiagonalQuads(quadrants[])
    if ($angle < 180$) :
        stateOP = true
    else :
        stateOP = false

elseif (noOfAxesWithNbrs= 4) :
    stateOP = false
elseif (($noOfAxesWithNbrs < 4$)and (noOfQuadsWithNbrs= 0)) :
    stateOP = true
return stateOP
```

```
Procedure SendDrvdGmtryReply()
if (operation = plus) :
    txTime = ComputeDrvdGmtryTxTime(m_DRVDGMRYREPLY)
    Set_Alarm(a_t3, txTime)
else
    txTime = ComputeInducedGmtryTxTime(m_DRVDREPLY)
    Set_Alarm(a_t3, txTime)

```

Figure B.33: Procedures used by *DeriveGeometry* protocol

Protocol Intersection(*gmtry1ID*, *gmtry2ID*, `Struct` *TempGmtryRec* * *tmpGmtry*)

```
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the intersection operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed whether it is
//                 part of new derived geometry and all of its associated attributes at time $t_f\}$

Status Values: $S$= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isOperandsDTValid = false
        tmpGmtry→stateOP = OPDONE
        CBN = false
        isReqdToPartInNgbrLookup = false

        if ((tupleG1= null) and (tupleG2= null)):
            tmpGmtry→stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = OperandsDTValidIntersection(tupleG1, tupleG2)
            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                tmpGmtry→gmtryDataType = DrvdGmtryOprndTypeIntersection(tupleG1, tupleG2)

                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1→bndryNode = true) and (tupleG2→bndryNode = true)):
                        if ((tupleG1→dataType = points) and (tupleG2→dataType = points)):
                            tmpGmtry→isGmtryMbr = true
                            tmpGmtry→isBndryNode = true
                        else :
                            CBN = true
                    else :
                        tmpGmtry→isGmtryMbr = true
                        tmpGmtry→isBndryNode = false
                        if ((tupleG1→bndryNode = true) or (tupleG2→bndryNode = true)):
                            tmpGmtry→isBndryNode = true
                    tmpGmtry→gmtryTTL = MinimumTTL(tupleG1→TTL, tupleG2→TTL)
            else :
                tmpGmtry→stateOP = OP_NOT_SUPPORTED

        if ((tupleG1→dataType = points) or (tupleG2→dataType = points)):
            isReqdToPartInNgbrLookup = false

        if (isReqdToPartInNgbrLookup)
            if ((tupleG1→dataType = lines) and (tupleG2→dataType = lines)):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LLINTERSECT, &CBNState, intersection, tmpGmtry→gmtryID)
            elseif (tmpGmtry→gmtryDataType = lines):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, LRINTERSECT, &CBNState,intersection, tmpGmtry→gmtryID)
            elseif (tmpGmtry→gmtryDataType = regions):
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, RRINTERSECTION, &CBNState,intersection, tmpGmtry→gmtryID)
            else
                // Entity that is part of only one operand and of valid data type
                Become(OPERATION_EVALUATING)
                NeighbourLookUp(tupleG1, tupleG2, SHAREINFO, &CBNState,intersection, tmpGmtry→gmtryID)
        else
            // Entity that is not part of any operand
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, WAITIDLE, &CBNState,intersection, tmpGmtry→gmtryID)
```

Figure B.34: Protocol *Intersection*

```
POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true):
            if (tmpGmtry→gmtryDataType = points):
                if (CBNState = intersectingPoint):
                    tmpGmtry→isGmtryMbr = true
                    tmpGmtry→isBndryNode = true
                else :
                    tmpGmtry→isGmtryMbr = false
            elseif (tmpGmtry→gmtryDataType = lines):
                if (CBNState = intersectingPoint):
                    tmpGmtry→isGmtryMbr = true
                    tmpGmtry→isBndryNode = true
                else :
                    tmpGmtry→isGmtryMbr = false
            elseif (tmpGmtry→gmtryDataType = regions):
                if (CBNState = true):
                    tmpGmtry→isGmtryMbr = true
                    tmpGmtry→isBndryNode = true
        Become(POST_SPATIALOPERATIONEVALUATION)
        EvaluateSpatialValuedTask()

```

Figure B.34 (continued)

Protocol Vertices(*gmtry1ID*, Struct *TempGmtryRec* * *tmpGmtry*)

```
// $P_{INIT}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n starts evaluating the vertices operation at time $t_0$}
// $P_{FINAL}$ ≡ {∀ n ∈ N: n ∈ MBR ⇒ n has evaluated the operation and computed whether it
//                 part of new derived geometry and all of its associated attributes at time $t_f$}

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        isOperandsDTValid = false
        tmpGmtry→stateOP = OPDONE

        if (tupleG1= null):
            tmpGmtry→stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = OperandsDTValidVertices(tupleG1)
            if (isOperandsDTValid):
                tmpGmtry→gmtryDataType = DrvdGmtryOprndTypeVertices(tupleG1)
                if (tupleG1 != null):
                    if (tupleG1→bndryNode = true):
                        tmpGmtry→isGmtryMbr = true
                        tmpGmtry→isBndryNode = true
                        tmpGmtry→gmtryTTL = tupleG1→TTL
            else :
                tmpGmtry→stateOP = OP_NOT_SUPPORTED

            Become(POST_SPATIALOPERATIONEVALUATION)
            EvaluateSpatialValuedTask()

```

Figure B.35: Protocol *Vertices*

to compute CBSs explained in Section B.2.1.6. An entity only declares itself an event entity (i.e., part of derived geometry) if it is a CBN and belongs to a valid CBS.

```
Protocol CommonBorder(gmtry1ID, gmtry2ID, Struct TempGmtryRec * tmpGmtry)
// $P_{INIT} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n starts evaluating the common_border operation at time $t_0\}$
// $P_{FINAL} \equiv \{\forall$ n $\in$ N: n $\in$ MBR $\Rightarrow$ n has evaluated the operation and computed whether it
//                is part of new derived geometry and all of its associated attributes at time $t_f\}$

Status Values: S= {TASK_EVALUATING, OPERATION_EVALUATING, POST_NEIGHBOURLOOKUP}
$S_{START}$= {OPERATION_EVALUATING}
$S_{TERM}$= {TASK_EVALUATING}

OPERATION_EVALUATING
    Spontaneously
        Struct GIT *tupleG1
        Struct GIT *tupleG2
        tupleG1 = LocalGITLookup_Search(gmtry1ID)
        tupleG2 = LocalGITLookup_Search(gmtry2ID)
        isOperandsDTValid = false
        tmpGmtry->stateOP = OPDONE
        isReqdToPartInNgbrLookup = false
        CBN = false

        if ((tupleG1= null) and (tupleG2= null)):
            tmpGmtry->stateOP = NOT_PART_OF_OPERANDS
        else :
            isOperandsDTValid = OperandsDTValidCmnBorder(tupleG1, tupleG2)

            if (isOperandsDTValid):
                isReqdToPartInNgbrLookup = true
                tmpGmtry->gmtryDataType = DrvdGmtryOprndTypeCmnBorder(tupleG1, tupleG2)
                if ((tupleG1 != null) and (tupleG2 != null)):
                    if ((tupleG1->bndryNode = true) and (tupleG2->bndryNode = true)):
                        CBN = true
            else
                tmpGmtry->stateOP = OP_NOT_SUPPORTED

        if (isReqdToPartInNgbrLookup):
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(tupleG1, tupleG2, CBS, &CBNState,border_in_common, NULL)
        else :
            Become(OPERATION_EVALUATING)
            NeighbourLookUp(NULL, NULL, CBS, &CBNState,border_in_common, NULL)


POST_NEIGHBOURLOOKUP
    Spontaneously
        if (CBN = true):
            if (CBNState = segment_node):
                tmpGmtry->isGmtryMbr = true
                tmpGmtry->isBndryNode = true
                tmpGmtry->gmtryTTL = MinimumTTL(tupleG1->TTL, tupleG2->TTL)

        Become(POST_SPATIALOPERATIONEVALUATION)
        EvaluateSpatialValuedTask()

```

Figure B.36: Protocol *CommonBorder*

B.4 Geometry Induction Tasks

This section, discuss the *InduceGeometry* protocol. The protocol supports re-evaluation of tasks up to a fixed number of times. The protocol for geometry induction is given in Figure B.37. This protocol first calls the *Sense_EventGmtry* protocol to sense the event geometry, then it calls the *DeriveGeometry* protocol with a request to compute the boundary of event geometry (i.e., line 19 in Figure B.37). For computing the boundary of the induce geometry CFEBD algorithm [RZL06] discussed in Section 3.4.1 is applied.

Upon completion of boundary detection, the entities with a **true** event state insert a tuple in the GIT for *induced geometry.* The procedure *GIT_InsertTuple* is responsible for inserting the tuple in the GIT if it does not already exists. If the tuple already exists for that induced geometry, its TTL is reset.

```
Protocol InduceGeometry(predicate, reEvalPeriod, inducedGmtryID, duration)
// $P_{INIT}$ ≡ "All entities in the network can evaluate geometry induction task at time $t_0$" ≡
//              {∀ n ∈ N ⇒ n can evaluate the task at time $t_0$}
// $P_{FINAL}$ ≡ "Nodes in the induced geometry have stored the information about the geometry in their GIT by time $t_f$"

Status Values: S= {TASK_PREPROCESSING, INDUCING, POST_GEOMETRYDERIVATION, AVAILABLE, POST_SENSINGGEOMETRY}
$S_{START}$= {TASK_PREPROCESSING,POST_GEOMETRYDERIVATION, POST_SENSINGGEOMETRY}
$S_{INTERMEDIATE}$={INDUCING }
$S_{TERM}$ = {AVAILABLE}
TASK_PREPROCESSING
    Spontaneously
        isEventNode = false
        isBndryNode = false
        Set_Alarm(a_t22, reEvalPeriod)
        Become(INDUCING)
INDUCING
    Spontaneously
        Sense_EventGmtry(predicate,&isEventNode)
        Become(OPERATION_EVALUATING)
        DeriveGeometry(inducedGmtryID, &isEventNode, true, &isBndryNode, IG)
POST_GEOMETRYDERIVATION
    Spontaneously
        if (isEventNode = true)
            GIT_InsertTuple(inducedGmtryID, isBndryNode,REGION, reEvalPeriod)
        Become(AVAILABLE)
AVAILABLE
    When(a_t22)
        evalPeriod = ComputeReEvalPeriod(evalPeriod, reEvalPeriod)
        if (evalPeriod < duration)
            Become(TASK_PREPROCESSING)
        else :
            Become(AVAILABLE)
```

Figure B.37: Protocol *EvaluateInduceValuedTask*

```
Procedure ConfirmInducedGmtryBN()
countEventNbrs = 0
countNonEventNbrs = 0

isEventNode = ComputeIsTrueEventNode(sectorNbrID[sector], sectorNbrState[sector])

if (isEventNode = true)
    for sector = 1 to 12
        if (sectorNbrState[sector] = true)
        else :
            countNonEventNbrs = countNonEventNbrs + 1

    computedValue = (1.00-((countEventNbrs-countNonEventNbrs)/(countEventNbrs+countNonEventNbrs)));
if (isEventNode = true)
    if (computedValue > CFEBDTHRESHOLD)
        opState = true
    else
        opState = false
```

Figure B.38: Procedures used by *InduceGeometry* protocol to compute the boundary of Induced geometry