A Model Independent and User-Friendly Querying System for Indoor Spaces

Amrutha H.
Dept. of Computer Science and Engineering,
Amrita School of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham (University)
amrutha.hari12@gmail.com

Vidhya Balasubramanian
Dept. of Computer Science and Engineering,
Amrita School of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham (University)
b_vidhya@cb.amrita.edu

ABSTRACT

Querying indoor information has become important with increasing demand for indoor pervasive applications in vogue. A number of applications have been developed like indoor navigation, localization etc., which work on the modeled indoor data. Different models like geometric, spatial and topological models exist for the indoor space. Existing query languages are model specific, and not user friendly. We propose a querying system which will work irrespective of the underlying model by hiding the complex details of the indoor model from the user. A querying framework is developed which abstracts out basic entities and primitive operators from multiple models. A text-based query language for the indoor space is built on this framework. A visual querying interface is developed which further simplifies the task of querying.
Index Terms- indoor information modeling, querying framework, visual querying

1. INTRODUCTION

Indoor information modeling and management has gained significance, with a large number of applications like indoor navigation, localization, asset management etc operating on the indoor space. To support these applications an effective querying framework over indoor space is necessary. Existing querying systems over indoor space have been developed based on the underlying indoor models like geometric, spatial and topology based models [7]. Models constructed for the indoor space, represent its entities like rooms, doors etc, relations between the entities and a set of constraints. Each model deals with different aspects of an indoor space. Spatial models represent the spatial attributes of entities and relations, topology based models represent the space as a set of entities connected by a set of relations[11] etc. Each model is stored in suitable databases and querying is done using the general purpose query languages supported by them.

The current query languages which support querying over indoor models (e.g. SQL which supports spatial models [16], Cypher Query Language that supports topology based model [1] and BIMQL for Building Information Models that supports a semantic model [12]) have a syntax that is difficult for use by non-professional users. These languages use complex terminologies, and are tightly coupled with the underlying modeling framework. The user needs to be familiar with the specific terminologies associated with a framework, and the way in which the space is modeled, each time he queries the data model stored. In current systems, to query an indoor space, a naive user has to either directly query the underlying database using the associated general purpose query language or use an existing model specific language making the querying complicated. This necessitates development of a generalized query language which can work above multiple indoor data models.

The next challenge is that existing query languages over indoor space are complex i.e, though they use SQL like syntax, the queries are long and complicated. For instance for finding a path between two points in the indoor space, a function has to be written in the underlying language. There are no simple and direct constructs that can help users specify such queries easily. While such constructs have been developed for outdoor spatial applications, it has not been developed in the indoor domain to the best of our knowledge. Also, to ease the querying process further, effective visual querying systems are needed, as there are no known visual query interfaces for indoor spaces. Compared to text-based querying, visual querying mechanisms simplify the task of querying and provide an increased level of comprehension [13]. The user friendliness of querying can hence be improved by adopting a visual querying interface.

In this paper, we address the above issues by developing a model independent querying framework for the indoor space. This querying system can be used in different application scenarios irrespective of the underlying data models. Along with providing a model independent querying system, the work aims to enhance the user's querying experience, by defining an indoor query language that can help construct indoor queries easily both using SQL like syntax and a visual query interface.

To achieve these goals, we develop a querying framework which abstracts out the basic entities and operators which are common to multiple models. Based on this querying framework, SQL type text-based query operators are developed. An SQL type query language is developed as SQL

[0]This work has been funded in part by DST(India) grant DyNo. 100/IFD/2764/2012-2013

The 20th International Conference on Management of Data (COMAD), 17th-19st Dec, 2014 at Hyderabad, India.
Copyright ©2014 Computer Society of India (CSI).

syntax shares similarities with most of the existing query languages. A visual querying component is added above this language to help the user construct queries with much ease and improved comprehension. For using the querying system above multiple data models, translation modules are designed to translate the input queries to the general purpose languages supported by the models.

The rest of the paper is organized as follows : Section II and III present the related work and illustrate the architecture of the proposed indoor querying system. Section IV deals with design of the model independent querying framework along with its evaluation. The query translation and its evaluation are discussed in Section V. Conclusion and future directions are given in Section VI.

2. RELATED WORK

Our goal in this paper is to design a querying framework that is model independent and is user friendly. In this section, we detail the existing spatial querying approaches for both indoor and outdoor space, and motivate the need for our work.

One of the primary problems in querying spatial data is the complex syntax of the spatial functions. To address this, one of the earlier approaches to make querying over spatial data more easier is to use Structured Query Language (SQL) extensions. Works based on this approach, add functionalities to SQL for supporting spatial queries like shortest path and nearest neighbor queries. One such query language developed for spatial databases is Spatial SQL [8]. This provides support for spatial data types like lines and polygon, operators like intersects, disjoint etc., and predicates over SQL. Some systems, additionally use an interface that allows for spatial objects used in the queries to be picked from the screen. Another work with a similar approach is GEOQL (GEOgraphic Query Language) [14] that defines a similar set of spatial predicates for geographical data. In [3], a spatial query language for building information models is designed by adding extensions to SQL. It defines a set of geometric operators between objects in a 3D space by designing a 9IM (9 Intersection model). The operators defined are '*contain*', '*disjoint*', '*equal*', '*overlap*', '*touches*' and '*within*' between the geometries. In this language however, the specific terminologies in terms of IFC (Industry Foundation Classes) standard like IfcSpace, IfcDoors, etc. are used in the queries, making it difficult to use.

Another approach is to define a new language for a particular domain. A domain specific query language captures the semantics of the domain better than a general purpose query language. BIMQL (Building Information Model Query Language) [12] is an open source spatial query language developed for the spatial analysis of building information models. This is an improvement to the previously mentioned work that extended SQL for building information models. Building Information Modelling(BIM) is the standardization of IFC(Industry Foundation Classes) based models of buildings. The IFC specific terminologies like IfcDoor, IfcStandardWallCase etc., are replaced by natural language terms like 'doors', 'walls' etc. The language hides the complex terminologies involved in the IFC based modeling but does not reduce the complexities of query syntax. In addition the language is still tightly bound to the underlying model.

An indexing for the trajectories and a query language for finding the indoor objects is proposed in [10]. It uses two R-Tree based structures to represent the user trajectories. The queries defined are of the format, $Q(E_s, E_t, P)$ where E_s is an indoor space partition, E_t is the temporal extent and P is the topological predicate. This primarily is designed to support trajectory based querying and is not extensible to general indoor querying.

In order to support the heterogeneity in GIS data, a VirGIS mediation system is proposed in [4] for the outdoor space. There exist different data sources for GIS data (e.g. topographic maps, satellite images etc.). The system proposed in this work provides a unified model for supporting data from different data sources. A global schema is developed which represents a set of abstract features like roads, bridges etc. in the outdoor space. Mappings from the global schema to the underlying local data sources are done using one to one mappings. The queries issued to the global schema are converted using the corresponding local schema.

To improve user friendliness of queries several approaches have been proposed, one of which is to use natural language. One such system [17] adopts a controlled natural language interface for GIS(Geographic Information Systems). Since the introduction of natural language interfaces can lead to vague inputs from the user, the work proposes a controlled language interface. A semantic representation of the GIS queries called Lambda SQL is defined which serves as an intermediate representation to the interface. The natural language query is converted to the intermediate format which is then converted to the SQL query with spatial support. This language works only for outdoor queries and high level queries describing a building and is not generalizable to any model.

Another approach to increase ease of querying, is to use a menu based natural language interface as is proposed in (MBNLI) [18]. It uses a completion based menu interface where each word selected by the user is parsed and another set of words are suggested to construct the query. This helps overcome issues in natural language queries and prevents the user from writing vague queries. An extension to MBNLI is introduced in [5] to support geospatial queries. Here support for spatial operators such as intersects, contains, touches, covers, disjoint etc. are added as defined in Oracle. The MBNLI query, termed as LingoLogic query is converted to the equivalent spatial query. The output is converted to KML(Keyhole Markup Language) and displayed in Google Earth. However such approaches are yet to be tested in a 3d space.

Visual querying is another suitable approach, which helps the user construct queries through visual interactions. Users need not learn the query syntax as in the text-based query languages. Visual querying on spatial databases is presented in [13], where a diagrammatic technique is used based on a data flow metaphor. The flow of data between the input and output elements through one or more filters visually represents a query. Spatial entities and spatial relations (e.g. disjoint, touches, crosses, in etc.) are defined, which interact in constructing spatial queries. Another work [2] presents a prototype implementation of Spatial-Query-By-Sketch which is a sketch based user interface to query GIS data. While the previous work involves using a set of icons for querying, this approach processes the sketch drawn by the user to convert it into a canonical form called digital sketch. This format identifies the entities, their topological and directional relations.

While several querying approaches are available as mentioned above, they are developed to suit a particular modelling framework. Also, visual querying which makes the task of querying the most simpler is not implemented for indoor information. We develop a generic query language to support the various(spatial, geometric and topology based) models of the indoor space. Additionally a visual querying interface that helps enhance the user's experience of building the queries is introduced in the system.

3. ARCHITECTURE OF THE INDOOR QUERYING SYSTEM

We now explain the architecture of the proposed querying system which will work on multiple models of indoor space. To achieve this, the system works on a framework that abstracts out the details that are common to most used indoor models inorder to construct a generic representation. Text-based and visual querying languages are designed based on this framework. The working of the system starts with the user constructing a visual query, which is converted to the text-based query defined specifically for indoor spaces. This query is then converted to the corresponding query languages like SQL or cypher query language associated with the underlying database. Figure 1 presents the architecture diagram of the indoor querying system proposed in this work. The main modules of this system are explained as follows.

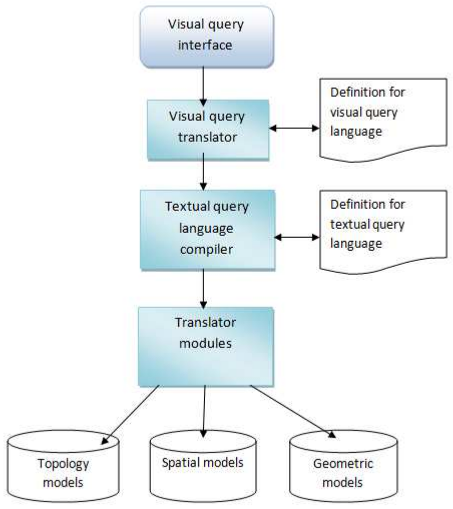


Figure 1: Architecture of the indoor querying system

- Visual query interface
  This provides a 3D visualization of the indoor space through which the user interacts to construct the visual queries. For each query, a set of visual interactions are defined like selecting the query type and giving the query parameters visually.

- Query compilers and translators
  These enable the translation of the input visual queries to a format which can be issued to the stored indoor data models. There are two query compiler modules defined in the system.

  – Visual query compiler
    The compiler processes the visual query input by the user to generate the query in corresponding text-based query language defined in the system. The relevant details like the query type and query parameters are extracted and substituted in the text query syntax.

  – Text-based query compiler
    This component parses the text-based query to validate the query syntax and aid its translation. The compiler on parsing each textual query generates an abstract syntax tree.

- Translator modules
  The parsed text query from the compiler module is fed to the translator to generate the queries in languages supported by each databases. Separate translation modules exist to generate queries in these general purpose query languages.(e.g. To SQL for PostGIS[16], to cypher queries for Neo4j [1] etc.)

- Databases
  Indoor information models are of different types like geometric, spatial and topological. Based on the data models, different databases are adopted (e.g. Topology based model best represented in a graph database like Neo4j, spatial models represented in PostGIS etc.).

The proposed querying framework works irrespective of the underlying models. The framework is formulated using the abstractions from different models. Based on this framework, a text-based and visual query languages are defined. The translation to the existing general purpose languages are done by the translator modules defined in the system. The next section will delve into the conceptual modeling of our querying framework.

4. MODEL INDEPENDENT QUERY FRAMEWORK

The primary purpose of this work as mentioned in previous sections is to generate an indoor querying system that is generic enough to support any indoor modeling framework. To achieve this, we propose the underlying framework that defines the basic indoor entities and primitive operators that operate in the indoor space. We identify the basic entities and operators in the main models of indoor space namely spatial, topological and geometric, and define a minimum common set that can map to entities and operators of these models in constant time. The identified entities and operators in the indoor space are as given below. While these entities are similar to the definitions of IndoorGML, they have been defined keeping in mind specifications in the most common indoor models.

- Space : This represents all the entities which semantically represent a space in a building's interior. These include rooms, corridors, sub-spaces of corridors/rooms etc. Space is created by a set of boundaries that determine its dimensions.

- Boundary: This corresponds to all the boundary structures which enclose the space entities in the indoor space. Boundaries can be classified as navigable boundaries and non-navigable boundaries. The non-navigable boundaries are the boundary components which block navigation, like the walls. The navigable boundaries are those which form the boundaries of the space entities and allow navigation, like the doors.
- Transition: Transitions are the entities in the indoor space which enable the movement from one space to another. They are the navigable connections which exist within any indoor environment. Examples of entities which belong to this class are exits, stairs and elevators.

To demonstrate the completeness of the chosen entities and operators, we show the correspondence between them and those in the existing modelling frameworks. The equivalence of entities in different models are shown in Table 1.

Basic Entities	PostGIS (spatial)	BIM (geometric)	IndoorGML (topology based)
Space	Polygons	IfcSpace	Abstract Space
Boundary	Polylines	IfcWall, IfcDoor, IfcWindow	Abstract Space Boundary
Transition	MultilineString	IfcStair	Transfer Space

Table 1: Entity equivalence

In the spatial model, space is characterised by its geometry which defines its extent and position in the space under consideration[11]. This model defines a set of geometry types like point, polygon, linestring, multilinestring etc, which is common to spatial database extensions like PostGIS. The indoor entities are represented as polygons, polylines or multilinestrings, providing a direct mapping to the spatial model. In the IndoorGML standard specification, which deals with the topology based modeling of indoor space, a set of classes are defined for entities and a set of relations between these entities to form the topologies. The classes defined for the entities are AbstractSpace to represent an indoor space(e.g. rooms), AbstractSpaceBoundary to represent boundaries of an indoor space(e.g. walls), TransferSpace to represent passages from one space to another(e.g.stairs) etc. [11]. Such mappings can also be provided to other specifications of the topological model. Semantic representation is used in the Building Information Modeling(BIM), which represents a building's design as a collection of objects. The underlying modeling of space is based on geometric modeling. The objects carry their geometry, attributes and relations [6]. Industry Foundation Classes(IFC) is the standard which allows representation and exchange of BIM data. Different classes exist which defines the indoor entity types like IfcSpace, IfcDoor, IfcWalls, IfcWIndows, IfcStairs etc. The space entity defined in our framework is an abstraction of IfcSpace, boundary is an abstraction of IfcWalls, IfcDoors etc., and transition is an abstraction of IfcStairs in BIM.

Next we define the set of primitive operators through which the basic entities interact with each other. These operators form a maximum subset such that they can be transformed to the operators in any model in constant time. Table 2 shows the primitive operators which are specified between each of the above defined entities.

	SPACE	BOUNDARY	TRANSITION
SPACE	Adjacent, Connected, Overlaps, Within, Intersects.	Boundedby, Linked	Linked, Connected
BOUNDARY	Bounds, Linked	Intersects, Touches	Intersects, Touches
TRANSITION	Linked, Connected	Intersects, Touches	Connected

Table 2: Primitive operators in proposed Indoor Query Framework

We define these operators based on relationships between entities in the indoor space in various contexts. A space is *'adjacent'* to another space when there is a common boundary between them. Two spaces are *'connected to'* each other when there is a navigable boundary like a door or a transition between them. A space is *'linked to'* a navigable boundary and a space is *'bounded by'* a non-navigable boundary. Other standard spatial relations [16] like *'intersects'*, *'touches'*, *'overlaps'* and *'within'*, existing between the entities are also given in Table 2. These primitive operators specified in this work and their definitions are given in Table 3. The notations are defined as follows: E refers to an entity, G_{Entity} to the geometry of an entity, S refers to space, NB to navigable boundary, NNB to non-navigable boundary and T refers to transition.

To demonstrate the model independence of these operators and for translating the proposed language to the existing models, there needs to be a correspondence between the operators defined in our framework and that of the existing frameworks. We show the corresponding operators in the existing modelling frameworks like PostGIS and BIM in Table 4. The correspondences, either direct or two-step correspondences are as shown in Table 4.

These operators have either direct or two level correspondence with the operators in existing query languages. The former include operators which have a direct correspondence with PostGIS and BIM operators like Intersect, Within, Touches and Contains. The latter represents the operators that are equivalent to a combination of operations in PostGIS and BIM. For instance the '*Connected*' operator is defined for spaces S_1 and S_2/ transition T_2 if both intersect the same navigable boundary in PostGIS. In BIM, IfcSpace_1 is connected to the IfcSpace_2/ IfcTransition_2 if both intersect a navigable boundary such as IfcDoor.

4.1 Proposed query language

Using the specification of entities and operators over these entities as given above, we now define the proposed indoor query language. The goal of this language is to cover the indoor domain specific queries irrespective of the underlying

Primitive operator	Format	Definition
Intersects	E_1 Intersects E_2	Returns true when the geometry of two entities intersect.
Touches	B_1/T_1 Touches B_2/T_2	Two entities(B/T) touch when their geometries have at least one common point but their interiors do not intersect.
Overlaps	S_1 Overlaps S_2	Two spaces overlap when their geometries have a common part but are not completely contained by each other.
Within	S_1 Within S_2	A space lies within another space when the geometry of the former lies completely inside that of the latter.
Boundedby	S Boundedby NNB	A space S is bounded by a non-navigable boundary NNB when their geometries intersect
Bounds	NNB Bounds S	A non navigable boundary bounds a space when they have intersecting geometries.
Linked	S Linked NB/T	A space has a linked relation to a navigable boundary or a transition T with which it intersects.
Adjacent	S_i Adjacent S_j	Two spaces are adjacent when they are linked or bounded by a common boundary.
Connected	S_i Connected S_j/ T_j	A space is connected to another space or transition when they intersect the same navigable boundary.

Table 3: Primitive operator definition

Operator	PostGIS	BIM
One level correspondence		
Intersects	ST_Intersect (G_{E_1}, G_{E_2})	$IfcElement_1$ not *disjoint* $IfcElement_2$
Touches	ST_Touches (G_{B_1}/G_{T_1}, G_{B_2}/G_{T_2})	$IfcBuildingElement_1$ *touch* $IfcBuildingElement_2$
Overlaps	ST_Overlaps (G_{S_1}, G_{S_2})	$IfcSpace_1$ *overlap* $IfcSpace_2$
Within	ST_Within (G_{S_1}, G_{S_2})	$IfcSpace_1$ within $IfcSpace_2$
Two level correspondence		
Boundedby	ST_Intersect (G_S, G_{NNB})	IfcSpace *intersect* IfcWall/ IfcWindow
Bounds	ST_Intersect (G_{NNB}, G_S)	IfcWall/ IfcWindow *intersect* IfcSpace
Linked	ST_Intersect (G_S, G_{NB}) or ST_Intersect (G_S, G_T) and vice versa	IfcSpace *intersect* IfcDoor/ IfcStair and vice versa
Adjacent	ST_Intersect (G_{S_1}, G_{NNB}) && ST_Intersect(G_{NNB}, G_{S_2})	$IfcSpace_1$ intersect IfcStandardWallCase and IfcStandardWallCase intersect $IfcSpace_2$
Connected	ST_Intersect (G_{S_1}, G_{NB_1}) && ST_Intersect (G_{NB_1}, G_{S_2}/ G_{T_2})	$IfcSpace_1$ intersect IfcDoor intersect $IfcSpace_2$/ $IfcStair_2$

Table 4: Equivalence of primitive operators

models, and to provide user friendly querying experience. It is designed so that a single query in the language can replace a set of multiple queries, in a model specific query language. For instance, to find all the entities which fall within a specific range around a given entity, the user has to write a function which consists of multiple sub-queries in existing query languages. In addition, the logic for range querying in 3d space has to be implemented by the user. This task can be simplified by a single range query syntax defined in the new language. This section explains the proposed query language and demonstrates how it achieves the above objectives.

The indoor query language, is defined to have an SQL like syntax, with clauses, predicates and expressions, since SQL like statements are easier to express.

Any query in this proposed query language has the format

```
'Find indoor entity where conditions'.
```

The *'Find'* clause contains the indoor domain specific entities or items to be selected. This is followed by an optional *'Where'* clause in which one or more conditions can be specified similar to an SQL query. The indoor queries can be broadly classified into the following types: attribute queries, spatial (e.g. adjacent, k-NN etc.) and geometric (e.g. finding area, volume).

Attribute queries select the indoor entities based on some operations on their attributes. An "attribute query" finds all entities E$_i$'s of the specified type whose attributes satisfy the conditions given in the query. These queries can be specified as 'Find entity where *conditions*'. For example 'Find room where type='classroom" is an attribute query.

Spatial queries deal with the spatial characteristics of the entities in the indoor space. Since indoor space is three dimensional, the spatial queries have to be modified to suit this space. They involve the use of spatial relations defined in the querying framework. For example, 'Find adjacent(SPACE)' , finds all the spaces which have a common boundary with this space. The common boundary can also be in the 'z' axis i.e., across floors. Similarly a range

Query category	Syntax	Definition	Description
Attribute query	Find E_i where E_i.Attr$_1$= value$_1$ && E_i.Attr$_2$ =value$_2$..	For every E_i , return E_i : { E_i.Attr$_1$= value$_1$ && E_i.Attr$_2$= value$_2$..}	Returns all entities of the specified type which satisfy the conditions specified on its attributes.
Adjacency query	Adjacent (S_i) where *conditions*	For every space S_j, return S_j, : { S_i adjacent S_j is true && *conditions* met}	Two SPACE entities are adjacent if they share a common boundary (navigable or non navigable) and all the conditions met.
Path query	Path (S_{start}, S_{end}/ T_{end}) [not] through E_i where *conditions*	Returns P ={ S_{start}, S_2, .. , S_{end}} such that for every S_i, S_i linked NB_i linked S_{i+1}/ T_{end} is true && E_i[not] in P && *conditions* met.	Returns the sequence of connected spaces or transitions between the start and end entities, with the conditions met.
Range query	Range (Type of entity, S_{origin}, *range value*) where *conditions*	For every E_i , return E_i :{ p= path(S_{origin} to E_i) exists && *conditions* met && length(p) $\leq$ *range value* }	An entity of the type specified selected if there is an accessible path which falls within the specified range meeting the conditions specified.
K-nearest neighbor query	Knn (Type of entity, S_{origin}, *k value*) where *conditions*	For every E_i , return E_i :{ p= path(S_{origin} to E_i) exists and E_i is not kth entity && *conditions* met}	Finds the first k entities of a given type at closest navigable distance from the queried entity and meet the conditions specified.
Volume	Find Volume(E_i)	Returns Volume(E_i)	Returns the volume of the specified entity calculated based on the geometry.

Table 5: Indoor query definitions

query aims to find all entities of a specified basic type (space, boundary or transition) that fall within a specific distance around a target entity. Here the range is based on navigable distance, rather than Euclidean distance due to obstacles in indoor space. The query format is '*Find range(entity type, entity, range value)*'. Similarly k-nearest neighbour queries return k entities, which are at the closest navigable distances from the origin entity.

Geometric queries deal with the geometric attributes of the indoor space entities, and they work based on the relationships between geometries. A query to find the volume of an entity also belongs to this category. The syntax for this query is defined as 'Find volume(entity)'.

Navigation queries help find the path between two points in the indoor space. It can be specified by using the *'Path'* keyword. The '*path query*' finds a sequence of spaces { S_{start}, S_2, .. , S_{end}} where each consecutive pair of spaces in the sequence are connected to each other. A sequence of connected spaces from the origin space which lead to the end space are identified. In order to find the shortest path among these, the navigable distance between the spaces are given to A* or Dijkstra's based path finding modules [19]. These queries can include basic shortest path queries or constrained path queries, which specify constraints like paths without stairs etc. In addition, the query language can also specify queries which are a combination of above queries. The *'Where'* clause can be used to combine different types of queries.

The query language syntax is defined based on the BNF grammar definitions of SQL(BNF Grammar for ISO/IEC 9075-2:2003- Database Language SQL(SQL-2003))[9]. The grammar defined for the query language is given in the following part. ANTLR parser generator[15] is used for constructing the compiler.

⟨*statement*⟩ ::='`Find`' qstatement [wherestatement]

⟨*qstatement*⟩ ::= attributestmnt | adjstmnt | pathstmnt
| knnstmnt | rangestmnt | geomstmnt

⟨*attributestmnt*⟩ ::= space | boundary | transition

⟨*adjstmnt*⟩ ::= '`Adjacent`' '('space')'

⟨*pathstmnt*⟩ ::= '`Path`' '('entity'`,`'entity')' [passconstraint]

⟨*knnstmnt*⟩ ::= '`Knn`' '('etype'`,`'entity'`,`'kval')'

⟨*rangestmnt*⟩ ::= '`Range`' '('etype'`,`'entity'`,`'range')'

⟨*geostmnt*⟩ ::= '`Volume`' '(' entity ')'
| '`Area`' '(' entity ')'

⟨*passconstraint*⟩ ::= ['`not`'] '`through`' entity

⟨*wherestatemnt*⟩ ::= '`Where`' ⟨*conditions*⟩

⟨*conditions*⟩ ::= orExpr

⟨*orExpr*⟩ ::= andexpr ('`Or`' andexpr)*

⟨*andexpr*⟩ ::= compexpr ('`And`' compexpr)*

⟨*compexpr*⟩ ::= atom ('`Less`' | '`Equal`' | '`Grtr`' atom)?

⟨*entity*⟩ ::= SPACE | BOUNDARY | TRANSITION

⟨*SPACE*⟩ ::= room | corridor

⟨*TRANSITION*⟩ ::= stair | elevator

$\langle BOUNDARY \rangle$	::=	walls | windows | doors
$\langle etype \rangle$	::=	‘`space`’ | ‘`boundary`’ | ‘`transition`’
$\langle kval \rangle$	::=	Num
$\langle range \rangle$	::=	Num
$\langle atom \rangle$	::=	(‘`a`’..‘`z`’ | ‘`A`’..‘`Z`’) | ‘`0`’..‘`9`’)+
$\langle Num \rangle$	:	‘`0`’..‘`9`’+

The grammar defined for the language presents the syntax definitions for the specific query types which belong to the different indoor query categories. The syntax for each query has at least one basic entity in the indoor space among its parameters. The definition of each query made in terms of the basic entities and primitive operators that belong to the query framework developed in this work are as shown in Table 5.

As shown in the table, the language also supports additional constraints to be specified on some queries. For example, in path query, the user can specify constraints i.e., whether a path must or must not pass through an entity. This is specified through the ‘*passconstraint*’ part of the path query syntax. Additional conditions can be specified in the ‘where’ clause of each query. This helps provide completeness to the indoor query language. The query language as shown helps provide the user with a simple to use language where most indoor queries can be specified using simple constructs. To improve the user experience further, a visual query interface is proposed which is described next. The user inputs queries visually and these are converted to the above defined query syntax. The following subsection presents the visual composition of queries and the corresponding text-based queries generated in the language defined in this work.

4.2 Visual Query Language for Indoor Spaces

A visual querying interface helps improve the querying experience of a user, specially in the indoor space. The user makes visual interactions to construct the queries. This helps any user query the indoor space without need for an understanding of the underlying system or model. Additionally it reduces the chances of syntax errors being made when writing a text-based query.

The visual querying module designed for the indoor querying framework is presented in Table 6. It constitutes 1) a 3D visualization of the indoor space, 2) visual primitives/ operators defined for a set of queries and 3) translation mechanisms to generate equivalent textual queries. The visual operators help allow the user to construct the queries in the indoor space. These operators include the basic operations like select, point etc, which are common to most visual query interfaces. In addition we define operators specific to the indoor space, and the Table 6 show these operators, their operation and their equivalent text command.

The visual operators are designed in such a way that they visually express their functionality in 3D indoor space. Most of them have been adapted from common visual operators used in current spatial systems. For instance, the operator for the k-nearest neighbor query is composed of a sphere surrounded by k-smaller spheres denoting the neighbors around

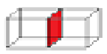
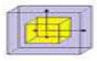
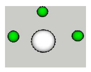
Visual operator	Querying mechanism	Equivalent text query
	Represents the adjacency of two spaces. This pointer is placed on the 3D entity for which adjacency is to be found.	Find adjacent (SPACE)
	This supports the range query. A resizable box is placed on the entity to be queried. Increasing/decreasing the pointer size sets the range.	Find range (Etype, SPACE, rangeval)
	This represents the path query. The user selects the start and end entities and a link is drawn between the entities for finding the path.	Find path (Estart,Eend)
	For the $k - NN$ query a sphere is first generated around the selected entity. Selecting the sphere generates one small sphere each time around the entity. The number of small spheres denotes the value of k.	Find knn(Etype, SPACE,k)

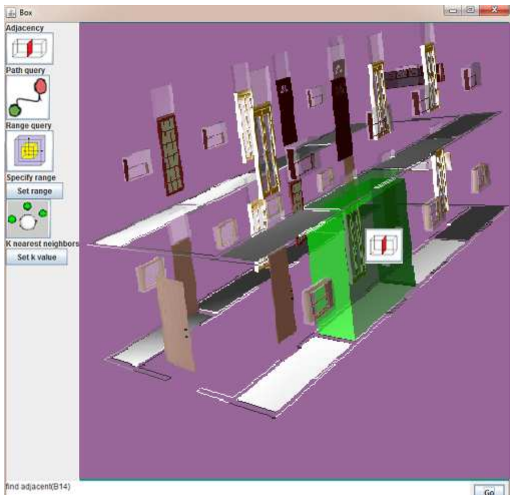
Table 6: Visual query primitives

an entity. Each operator visually explains its intended functionality. Queries like volume queries are defined using the simple select function and choosing options from the selected entity. For specifying constraints, menu buttons and text menus are provided.

Using these primitive visual operators a user can construct the desired indoor queries. To generate the equivalent text-based query, we need : 1) type of the query (e.g. adjacency, range query etc.), 2) entity on which the query is issued and 3) the associated parameters(e.g. value of range in range query, end entity in path query etc.). These are then substituted in the corresponding text query syntax. A visual query interface is designed, which allows the users to interact with the indoor space, and construct queries over it. Figure 2 shows a prototype visual query interface.

The interface consists of the operators panel, and the canvas showing the 3d visualization of the chosen building. The user selects the type of the query using the visual components (Table 6) presented in the top left side of the interface. Then an entity of interest is picked from the 3D visualization. The user now performs the interactions defined for each query (see Table 6).

The query type, entity chosen, and the constraints are then substituted in the textual syntax defined for each query. Since the visual query has a one-to-one correspondence with



```
find adjacent(B14)
```

Figure 2: Adjacency query constructed

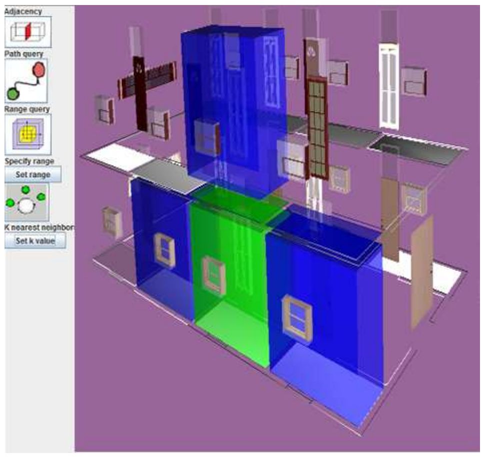


Figure 3: Adjacency query result visualized

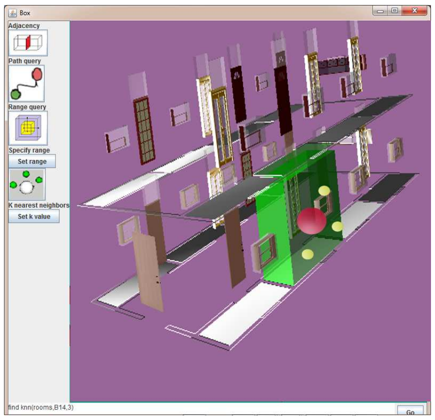
the textual query syntax, the conversion is done with simple substitutions of the details extracted from the visual interactions as mentioned previously.

Figure 2 shows the adjacency query constructed on a required indoor space. Selection on the visual primitives given on the left part of the interface is made to construct each query type. The corresponding text-based query is automatically generated and displayed in the text area at the bottom of the interface.

The resultant adjacent rooms of the given space computed from the indoor data models(spatial models stored in PostGIS, topology based models stored in Neo4j) are shown in the 3D visualization (Figure 3).

Figure 4 shows an example k-nearest neighbor query constructed. The user first selects the visual operator for the k nearest neighbor query. A sphere pointer is placed above the chosen entity, and as the user clicks the number of outer spheres increase, indicating the value of k. Figure 5 shows the rooms that are highlighted as a result of this operator.

We have described so far, a querying framework, a text-based query language and visual querying mechanism that



```
find knn(rooms,B14,3)
```

Figure 4: k-NN visual query constructed

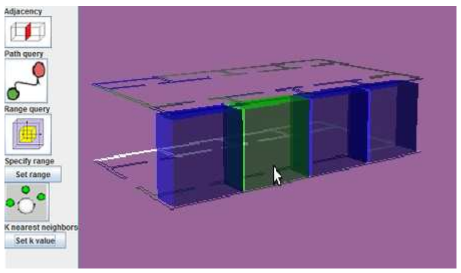


Figure 5: k-NN query result

have been developed based on this framework. The following sub-section deals with the evaluation of the proposed query framework.

4.3 Evaluating the query framework

The querying framework is evaluated based on a set of queries constructed in the proposed language and analysing the results obtained. This section explains the system environment and the use cases demonstrating the correctness and effectiveness of the proposed query framework.

4.3.1 System Environment

In order to evaluate the system, building's indoor models stored in PostGIS and Neo4j graph databases are used. The dataset consists of a simulated 20-storey building, where each floor has approximately 30 rooms.The database stores both the spatial and topological models of indoor space. The spatial model is represented as a set of tables storing the spatial and non spatial attributes of the indoor entities and is stored in the PostgreSQL database. The topological model consists of the following graphs namely adjacency, connectivity,and logical graphs with reference to the IndoorGML standard[11]. Each graph depicts the basic indoor entities as the nodes and a particular relation between them as the edges. The adjacency graph models the Adjacent_to relationships between the indoor entities, connectivity graph depicts the Connected_to relations and logical graphs model the access and temporal constraint information about the indoor entities. These graphs are generated as follows. Adjacency graph is generated by creating an Adjacent_to relation between the spaces that share a common boundary. Connectivity graph comprises of edges between nodes corresponding to the spaces or transitions which are linked to a common navigable boundary. Logical graphs are made by storing the user access types(normal or administrative) and temporal constraints(the open or closed status). These graphs are stored in the Neo4j graph database. The indoor querying system accesses both these databases.

4.3.2 Use cases

In order to evaluate the query language, we analyze the following a) model independence, b)correctness and c) completeness of the proposed query framework using use cases. Model independence is brought about when the queries can be specified irrespective of the underlying model, or language. Correctness is assured when the queries in the language yields results which match with the expected outcomes. Evaluation of completeness is to demonstrate that any query in the indoor domain can be issued using the language under consideration.

Now we present the set of use cases for performing evaluations in the above mentioned criteria. Table 7 presents use cases that evaluate the model independence of the querying framework. For each query in our language, equivalent queries exist in the SQL(for PostGIS) and cypher query language(Neo4j) to which it can be mapped. From the example queries in the tables we can see the following

- A query in PostGIS or Neo4j requires the user to understand the underlying database schema. In our language the user only has to enter the entity name, which is commonly used. The only constraint is that the name commonly used must be unique and an attribute in the database.

- Additionally a user should understand the spatial properties (property of r.geom) and how the geometrical operators like ST_Intersects work when querying the spatial model or the graph theoretical terminologies like nodes, and operators like "n-[:ADJACENT_TO]->m". In the proposed framework, the user just needs to understand the semantic aspects of indoor space and can be ignorant of the details of how it is represented.

These two aspects demonstrate the model independence of the query language.

In order to analyze the correctness of the query language, we have to assure that the structure of the query types is suitable for the underlying databases. This ensures that the query can retrieve the intended results by processing the indoor data which is available in the databases. The querying framework that we have defined in this work is based on the abstractions of different data models of indoor information. Each query is formulated based on these basic entities and operators which are abstracted out from the data models. Hence the queries will have a structure which is compatible to the data present in the databases. The queries also prevent the users from giving inputs that deviate from its defined structure or format. It does not allow the user to provide incorrect arguments to the queries and details which may not exist in the database. For instance, consider a query to find the adjacent rooms to a specific room and which are of type 'class room'. This query in textual form is represented as 'Find Adjacent(A19) where type='class room' '. The following sequence of interactions are accepted by our visual query interface to construct the query.

- The user picks the room on which the query is to be issued from the 3D visualization of the indoor space. This prevents the user from giving unavailable rooms or incorrect references to the rooms in the query.

- In order to state the condition that the adjacent rooms are to be of type 'class room', the user makes a selection from a drop down list which lists all the room types which are available in the database. This again prevents the user from giving a type which is unavailable.

Hence the query language prevents the user from constructing queries which are incorrect with respect to the indoor data stored in the databases.

The next set of use cases are provided to analyse the output of queries specified in the proposed Indoor query language. To analyze this, a set of queries (both simple and complex) in the indoor space are executed. The results obtained are compared with the ideal results. Tables 8 and 9 show the sample queries and their results, and corresponding expected results. Instances of attribute, adjacency, range, nearest neighbor and path queries are presented along with the results. These form basic set of queries which emerge in the indoor domain and hence are considered for the evaluation. For each query issued in the language, the expected and the obtained results are presented. The table shows that our queries retrieve the right results in various cases.

Finally we study the completeness of the proposed query framework. In this framework a set of basic queries have been defined. Any complex query can be formulated in terms of the basic query types defined. This ensures the completeness of the language. For instance, consider a query to find all rooms within 200 metres around a specific entity which are accessible to normal users and are open. This query can be constructed by enhancing the existing range query by adding conditions checking to the access types and the closed/ open status of the spaces. Consider another query for finding five nearest rooms which can accommodate at least 100 people, which may arise in the case of an educational institution. The above mentioned query can be written in our language as 'find knn (room, entity_id, 5) where capacity $\geq$ 100'. This indicates that new queries can be formulated and issued using the basic set of queries without changes to syntax.

5. QUERY TRANSLATIONS

We have proposed a model independent query framework and evaluated the query language in previous sections. One of the primary requirements in achieving this model independence is to be able to translate the query in the proposed language to any general purpose query language. Specifically, translations from visual to text-based and from text-based to the general purpose query languages supported by the modelling frameworks are required. This section details the translation mechanisms defined in the system.

The visual to text-based query translation involves simple substitutions. There exists a 1-1 correspondence between each text-based and visual query, as explained in the previous section, and hence the translation is as described

Query	PostGIS	Neo4j
Find Adjacent(A21)	SELECT r.roomname FROM bld1floor1rooms as s , bld1floor1walls as w, bld1floor1rooms as r, WHERE ST_Intersects (r.geom,w.geom) and ST_Intersects (w.geom,s.geom) and s.roomname='A21'	start n=node(*) match n-[:ADJACENT_TO]->m where n.roomname='A21' return n,m
Find range (rooms,B19, 200)	SELECT r.roomname, r.geom FROM bld1floor1rooms as s , bld1floor1doors as d, bld1floor1rooms as r, WHERE ST_Intersects (r.geom,d.geom) and ST_Intersects (d.geom,s.geom) and sum (ST_Distance (r.geom, d.geom), ST_Distance (d.geom, s.geom)) $\leq$ 200 and s.roomname= 'B19'	start n=node(*),m=node(*) match p= (n)-[r:CONNECTED_TO *..10]->(m) where n.roomname='B19' and sum (r.distance) $\leq$ 200 return m
Find rooms where type = 'conference halls' and floor=1	SELECT * from bld1floor1rooms where roomtype= 'conference halls'	Start n=node(*) match n.room n.type: 'conference hall', n.floor:'1' return n

Table 7: Evaluation of model independence

Query	Query in proposed language	Expected result	Obtained result
Find all restrooms in 2nd floor	Find rooms where type= 'restroom' and floor=2	(ID,name) (B5,Gents) (B8, Gents) (B15 , Ladies) (B29, Ladies)	(ID,name) (B5,Gents) (B8, Gents) (B15 , Ladies) (B29, Ladies)
To find all adjacent spaces of room P12	Find adjacent (P12)	(ID,name) (P6 , Room _6_6)	(ID,name) (P6 , Room _6_6)

Table 8: Evaluating querying framework's correctness

Query	Query in proposed language	Expected result	Obtained result
Find all adjacent class rooms to B4	Find adjacent (B19) where type='class room'	(ID,name) (B1 , IV IT A) (B3 , IV IT C)	(ID,name) (B1 , IV IT A) (B3 , IV IT C)
Finding path from B2 to B16	Find path (A21, B12)	{B2, CRB1, CRB2, CRB2, CRB2, B16 }	{B2, CRB1, CRB2, CRB2, CRB2, B16}
To find five nearest neighboring rooms of B4 of type 'classroom'	Find knn (rooms, B4, 2) where type='class room'	(ID,name) (B3 ,Room _3_3) (B5, Room _3_5) (B1, Room _3_1) (B12, Room _3_12) (B6, Room _3_6)	(ID,name) (B3 ,Room _3_3) (B5, Room _3_5) (B1, Room _3_1) (B12, Room _3_12) (B6, Room _3_6)
To find all rooms within 200m around A2	Find range (rooms, A2, 200)	(ID,name) (A5 ,CSE PI 2)	(ID,name) (A5 ,CSE PI 2)

Table 9: Evaluating querying framework's correctness (contd)

earlier. The translation of the text-based query to the existing general purpose languages is done by processing the abstract syntax tree (AST), which is generated while parsing the text-based indoor query. The nodes in the AST represent each construct in the input query. Parsing the input query involves a set of syntax rules matched from the syntax definition being invoked. Each rule invoked, triggers the generation of a subtree in the AST corresponding to the query. So the AST generation evolves through a sequence of syntax rule invocations.

In order to perform translation by processing the generated AST, the structure of the AST has to be defined and known. Each invoked syntax rule determines the structure of a part or a subtree of the query's AST. A syntax rule is made up of a set of constructs. In order to specify the structure of the AST's subtree generated by the rule, which construct forms the root node and which form the child nodes are defined. Consider a syntax rule with n constructs say $construct_1$, $construct_2$, .., $construct_n$. The structure definition format for the subtree to be generated is as given below.
$\wedge(construct_1\ construct_2\ ..\ construct_n)$
Here the first element after the $\wedge$ symbol indicates the root element. $construct_2$, .., $construct_n$ form its child nodes in the same order. For every syntax rule in the language's definition, a similar structure is defined. The evolution of an AST with the help of each syntax rule defined and the

associated structure definition is presented below. Every query is made of a 'qstatement'(or query statement), and the 'wherestmnt' which indicates the 'where' clause in the query. The AST therefore has the 'Find' as root of the AST and the 'qstatement' and 'wherestmnt' become the children. The subtree with 'qstatement' as the root node, consists of the type(e.g. adjacent, range, knn etc.) as its children. Based on the query type, the subtree is chosen. Consider an input query type to be a 'range' query. The syntax for range query statement is given by

```
<rangestmnt> ::= 'Range' ( etype, querypoint, range)
```

The structure of the corresponding sub tree in the AST is specified as

```
^(Range  etype querypoint  rval)
```

Here 'Range' forms the root of the subtree and the parameters form the children. Here 'etype' indicating the type of the entities, 'querypoint' the entity on which the query is issued and 'rval' the range value form the children.

The input query may have a set of conditions given in a 'where' clause. The syntax of the 'where' clause in the language is given as follows.

```
<wherestmnt> ::= 'Where' attrcomp
```

Here the construct 'attrcomp' indicates one or more attribute comparisons of the form *'attribute comparison-operator value'* combined using 'and' or 'or' operators. The structure of the corresponding subtree in the AST is defined as follows

```
^(Where attrcomp)
```

Similarly for each rule in the language's syntax definition, a structure definition is provided in terms of its constructs.

The AST generated is processed for performing the query translation. It is processed using a preorder depth first search traversal. We define an algorithm 1 which will specify the method of extracting the details from the abstract syntax tree. From any input query, the query type, the parameters and conditions given in the 'where' clause are extracted using this algorithm. Since the structure of the subtree corresponding to each syntax rule is defined, for each such subtree, we know which is the root node and in which order are the child nodes present. The algorithm presents, for each subtree of the AST defined, the method of extracting its details. For example, consider the method of processing the subtree corresponding to 'range' query type. Based on the structure defined, the first node is extracted as the type of entity, the second as the query origin and the third as the range value. All these details are then used to generate the general purpose queries namely SQL(for PostGIS) and cypher queries(for Neo4j). They are substituted to generate a single query as in attribute queries or used to invoke functions written in SQL or cypher query language in case of range, adjacent, k nearest neighbor and path queries.

An example of translating an attribute query from the proposed language to SQL is presented in figure 6. The entities and attributes specified are mapped to the entity and attribute names in the underlying schema to generate the SQL query. In some cases, converting these queries to the general purpose query languages involve invoking procedures corresponding to the query functions in the language.

Next we evaluate the designed translation mechanism by analysing the time incurred in a set of query translations.

```
Input: Node astRoot
Node N=astRoot;
if N.data='Root' then
    ProcessAST(N.getChild(0));
else if N.data='Find' then
    ProcessAST(N.getChild(0));
    ProcessAST(N.getChild(1));
else if N.data='where statement' then
    for Each child ch of N do
        $Condition_i$.operator=ch.data;
        $Condition_i$.attribute=ch.getChild(0).data;
        $Condition_i$.value=ch.getChild(1).data;
    end
    ConditionList.add(Condition_i);
else if N.data='Adjacent' then
    Entity= N.getChild(0).data;
else if N.data='Path' then
    startEntity= N.getChild(0).data;
    endEntity= N.getChild(1).data;
    if N.getChild(2) != null then
        passConstraint=N.getChild(2).data;
        passentity=N.getChild(2).getChild(0).data;
    end
else if N.data='Range' or N.data='knn' then
    EntityType= N.getChild(0).data;
    Entity= N.getChild(1).data;
    if N.data='Range' then
        rangevalue=N.getChild(2).data;
    else
        if N.data='Knn' then
            k=N.getChild(2).data;
        end
    end
else
    EntityType=N.getChild(0);
end
```

Algorithm 1: ProcessAST

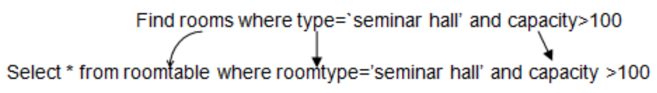

Find rooms where type='seminar hall' and capacity>100
Select * from roomtable where roomtype='seminar hall' and capacity >100

Figure 6: Attribute query generated for PostGIS

Table 10 presents the results of this analysis. Here t_1 represents translation time incurred in our language, t_2, the execution time of a query in our language, and t_3, the execution time of a query in PostGIS/Neo4j. It also shows for each query in the proposed language, the translated queries in the general purpose query languages. The translation proceeds as mentioned previously by traversing the abstract syntax tree constructed.

It can be observed that the translation times are considerably smaller and hence do not cause much overhead to the entire query execution. The extra translation part which exists in the system does not affect the total execution time of the query. To determine if this performance holds for a larger set of queries, we evaluated the translation time for 100 different queries. The average query translation time

Query	Translated query	Evaluation time (sec)
Find adjacent (A21)	SELECT r. roomname FROM bld1floor1rooms as s , bld1floor1walls as w, bld1floor1rooms as r, WHERE ST_Intersects (r.geom, w.geom) and ST_Intersects (w.geom, s.geom) and s.roomname =‘A21’	t_1= 0.02248, t_2= 0.09485, t_3= 0.07237 (PostGIS)
Find range (rooms, B19, 200)	start n=node(*), m=node(*) match p= (n)-[r: CONNECTED _TO *..10]-(m) where n. roomname=‘B19’ and sum (r. distance) $\leq$ 200 return m	t_1= 0.02276, t_2= 0.93542, t_3 0.91261 (Neo4j)
Find path (B14,B2) where length $<$ 400	start n=node(*), m =node(*) match p= (n)- [r: CONNECTED_TO * .. 10] -(m) where n. roomname =‘B14’ and n. roomname =‘B2’ and sum (r. distance) $<$ 400	t_1= 0.02851, t_2= 0.93557, t_3 0.90704 (Neo4j)

Table 10: Translations and evaluations

was found to be 0.0207 seconds, with a standard deviation of 0.0046 seconds. This demonstrates that the translation component is efficient and does not degrade the querying system's performance.

6. CONCLUSION AND FUTURE WORK

In this paper we propose a model independent querying system for indoor spaces. We have developed a querying framework which abstracts out and represents the common features of the underlying models. Based on this querying framework, a text-based and visual querying languages are developed. Visual querying enhances the ease of querying the indoor data models. Translation modules are defined for converting queries in the proposed query language to the general purpose query languages(SQL and Neo4j) supported by the models. This allows the system to be used above multiple models. Evaluations of the querying framework developed and the translation mechanisms demonstrate the completeness, correctness and model independence of this framework. The future work on this system include defining more queries in the visual querying system and adding support for other modelling frameworks like BIM and IndoorGML. Additionally user studies for evaluating the visual querying and improving it is part of ongoing work.

7. REFERENCES

[1] Neo4j graph database. http://www.neo4j.org/, November 2013.

[2] Blaser, A. D., and Egenhofer, M. J. A visual tool for querying geographic databases. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (2000), AVI ’00, ACM, pp. 211–216.

[3] Borrmann, A., and Rank, E. Topological analysis of 3d building models using a spatial query language. *Adv. Eng. Inform 23* (2009), 370–385.

[4] Boucelma, O., Essid, M., Lacroix, Z., Vinel, J., Garinet, J.-Y., and Betari, A. Virgis: mediation for geographical information systems. In *Data Engineering, 2004. Proceedings. 20th International Conference on* (March 2004), pp. 855–.

[5] Chintaphally, V., Neumeier, K., McFarlane, J., Cothren, J., and Thompson, C. Extending a natural language interface with geospatial queries. *Internet Computing, IEEE 11* (2007), 82–85.

[6] Chuck Eastman, Paul Teicholz, R. s. A guide to building information modelling for owners, managers, designers, engineers and contractors, 2011.

[7] Claus Nagel, Thomas Becker, R. K. Requirements and space-event modelling for indoor navigation, 2010.

[8] Egenhofer, M. Constraint qualifications in maximization problems. *Knowledge and Data Engineering, IEEE Transactions on 6* (1994), 86–95.

[9] IEC, I. Bnf grammar for iso/iec 9075-2:2003. http://savage.net.au/SQL/sql-2003-2.bnf, December 2013.

[10] Jensen, C. S., Lu, H., and Yang, B. Indexing the trajectories of moving objects in symbolic indoor space. In *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases* (Berlin, Heidelberg, 2009), SSTD ’09, Springer-Verlag, pp. 208–227.

[11] Jiyeong Lee, Ki-Joune Li, S. Z. Open geospatial consortium inc. indoorgml draft,reference no: Ogc 13-nnnrx, 2013.

[12] Mazairac, W., and Beetz, J. Bimql - an open query language for building information models. *Adv. Eng. Inform 27* (2013), 444–456.

[13] Morris, A. J., Abdelmoty, A. I., and El-Geresy, B. A. A visual query language for large spatial databases. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (New York, NY, USA, 2002), AVI ’02, ACM, pp. 359–360.

[14] Ooi, B.-C., Davis, R., and McDonnell, K. Extending a dbms for geographic applications. In *Data Engineering, 1989. Proceedings. Fifth International Conference on* (Feb 1989), pp. 590–597.

[15] Parr, T. Antlr. http://www.antlr.org/, December 2013.

[16] Research, R. Postgis. http://postgis.net/, September 2013.

[17] Sela Mador-Haim, Yoad Winter, A. B. Controlled language for geographical information system queries. In *Proc. of Inference in Computational Semantics*. 2006.

[18] Tennant, H. R., Ross, K. M., and Thompson, C. W. Usable natural language interfaces through menu-based natural language understanding. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1983), CHI ’83, ACM, pp. 154–160.

[19] Thomas H. Cormenn, Charles E. Leiserson, R. L. R. C. S. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.