Simplifying the Analysis of Building Information Models Using ^{t}QL4BIM and ^{v}QL4BIM

Simon Daum, simon.daum@tum.de
Chair of Computational Modeling and Simulation, Technische Universität München, Germany

André Borrmann, andre.borrmann@tum.de
Chair of Computational Modeling and Simulation, Technische Universität München, Germany

Abstract
This paper presents a new approach to analyse Building Information Models. Firstly, a domain-specific language in its textual notation is introduced. It is tailored to domain experts with little programming experience. Based on an uncomplicated pattern, the language offers a formal and general method of BIM-analysis. Additionally, its imperative nature and its sequential layout makes a stepwise execution and inspection of intermediate results possible. Thus, complex queries may consist of comprehensible and verifiable statements. As an alternative to textual definitions, we introduce a visual notation for BIM queries. Both the textual and the visual notation are based on the identical abstract syntax and offer the same range of functionalities.

Keywords: BIM, IFC, CityGML, Query Language, VPL

1 Introduction

The paradigms of Building Information Modeling (BIM) promote the use of comprehensive digital building representations. Instead of previously adopted approaches, which focus on geometric data, a BIM comprises not only the 3D shapes of elements and enclosed spaces, but also type information and the relationships between elements (Eastman et al., 2011). Therefore, a BIM has the potential to serve as an interface and data pool for exchanging information across the disciplines involved in the design, construction and operation of a building. The research presented in this paper is based on the data model *Industry Foundation Classes* (IFC), an open, evolved and universal schema for representing building models (Liebich, 2013).

However, various issues can arise when BIM processes are introduced in construction projects. In the context of data handling and the transfer of interdisciplinary information, the following issues are of central importance for domain experts such as architects and engineers:

- Building models have to be analysable and their quality must be verifiable.
- The extraction of submodels from an overall model must be supported.
- The effort involved in performing these analyses and transactions must be reduced.

The currently used BIM tools do not support a formal analysis of building models. Instead, manual work has to be dedicated to extracting the necessary information. Due to the diverging characteristics of analyses, these tasks have to be continuously repeated using changed parameters. This manual work is time-consuming and error-prone, especially in data intensive models. The same holds true for the extraction of submodels and post-processing.

Due to these weak points in a BIM-based workflow, new methods have to be invented. They should be developed with domain experts rather than programmers in mind and their application should be intuitive without requiring an extensive learning effort. At the same time, a general approach capable of supporting different types of projects with different boundary conditions should be adopted. Finally, the structure of data and its scope must be taken into account if new strategies for analysis, quality checking and data processing are to be introduced to BIM.

This paper is based on the hypothesis that a formal, domain-specific query language can solve the aforementioned deficits. Accordingly, the novel BIM query language *QL4BIM* and its runtime, the *QL4BIM System*, are presented. The language offers advanced operators and data structures for a comprehensive analysis of building information models. It comprises semantic operators for type/attribute extractions and relational operators for analysing the links between entities. Additionally, temporal operators for examining the installation times of building elements are introduced (Daum and Borrmann, 2013). Spatial operators for evaluating topological, directional and metric predicates are also included in the language (Daum and Borrmann, 2014).

The focus of previous research in the *QL4BIM* project was on spatial operators and a query interface which is based on Microsoft LINQ and .NET languages such as *C#* and *Visual Basic*. The comprehensiveness of this approach increases complexity in query definition, making the interface primarily suitable for users with a background in programming.

To realize a simplified approach to BIM analysis and processing, two new interfaces for the query system are introduced. The first one is the textual query notation *[t]QL4BIM*. In addition, with *[v]QL4BIM*, a visual query notation is integrated in the system, primarily dedicated to users who want to steer analysis using a graphical user interface rather than code. Both notations are based on a single abstract language syntax provided by the *QL4BIM* language. Despite the simplicity of the language, it allows comprehensive BIM queries.

This paper is structured as follows: Firstly, general and domain-specific query languages are discussed in brief. Then, the novel, textual query notation *[t]QL4BIM* is presented and the incorporated type system used by the runtime is addressed. Section 5 then describes the possibility of operator overloading in the system. Subsequently, an overview of the runtime components is provided. Section 7 shows the deduction of the visual notation *[v]QL4BIM*. The paper finishes with a conclusion on the research presented.

2 Related Work

The analysis of extensive data sets is covered by databases and their query languages. The *Structured Query Language (SQL)* is the de-facto standard in this domain (ISO, 2011). The sound theoretical foundation for this language is provided by Relational Algebra (Codd, 1990). However, the IFC data models are defined by the EXPRESS modelling language and there is no standardized mapping of this kind of schema to a relational structure (Schenck and Wilson, 1994). The propagation of XML-based modelling has led to the development of the XQuery language (Robie et al., 2013). ifcXML represents an XML-based variant of IFC which can be analysed without pre-processing by an XQuery implementation (Liebich, 2001).

In order to examine the application of general query language such as SQL and XQuery to the analysis of building models, a test bed has been implemented. In short, the experiments have shown that an analysis of a building model based on general query languages can result in tedious coding. For example, a query stated in XQuery that extracts walls and their material from an ifcXML file comprises 18 lines. A direct mapping of EXPRESS to SQL also produces a complicated structure of tables, which hampers this approach to data analysis (Borrmann and Rank, 2009).

A domain-specific language for BIM can provide more direct access to the required data, as it is tailored to the domain model and hides the details of the internal data representation. The Building Information Model Query Language (BIMQL) achieves such a domain-specific approach (Mazairac and Beetz, 2013). The language is inspired, inter alia, by SPARQL which is a query language for semantic web content (Prud'Hommeaux et al., 2008). In **Listing 1** a BIMQL query is presented that extracts the floor area of two spaces. This type of query is easily understood by individuals with knowledge of SQL and SPARQL but may confuse

Listing 1 BIMQL query returning the floor area of spaces

```
1 Select ?mySpace
2 Where ?mySpace.Attribute.GlobalID = 3Dn6BYWjfErxE1JocogMGQ Or
3    ?mySpaceAttribute.GlobalID = 0tLttARJ1F1esyGJSeaTmd
4 Select ?netFloorArea
5 Where ?netFloorArea := ?mySpace.property.NetFloorArea
```

domain experts. Additionally, the connection between each space and its floor area is lost as no relational results are produced. For a review of additional query languages please refer to (Daum and Borrmann, 2014).

A complex query can be composed of one or more statements. Each statement is built up by an assignment which binds a variable to the result of an operation. In addition, an operation can be subdivided into its operand and its parameters. Constants such as strings, numbers and floats as well as assigned variables can be used as concrete parameters. Furthermore, predicates evaluating attributes of entities are used to steer operators.

Listing 2 A query example in tQL4BIM

```
1 model = GetModel("C:\Institute.ifc")
2 allWalls = TypeFilter(model , "IfcWall")
3 someWalls = AttributeFilter(allWalls.Name = "Wall-002")
4 allWindows = TypeFilter(model, "IfcWindow")
5 rel[wall, window] = Touch(someWalls, allWindows)
```

3 tQL4BIM: The Textual Query Notation of the QL4BIM System

In previous research, a query interface based on LINQ was provided (Daum and Borrmann, 2013). It enables the end user do define queries in Microsoft .NET languages. While this is appropriate for programmers, domain experts such as architects and civil engineers are unable to use the query system intuitively. To resolve this restriction, a new streamlined language has been developed in accordance with the following paradigms:

- The language introduces a general and formal approach to BIM analysis.
- At the same time, only a few formal patterns are used in the language.
- An imperative query definition is intended, defining the ordering of statements and driving a sequential, comprehensible evaluation of subqueries.
- Type handling and low-level control flows are hidden from the end user.

Taking these requirements into account, an appropriate grammar for the textual notation *tQL4BIM* is developed. Parts of it are shown in **Figure 1**.

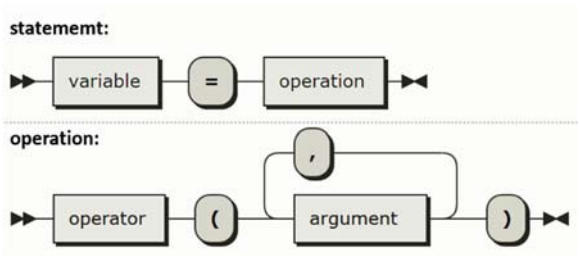

Figure 1 A section of the tQL4BIM grammar

Listing 2 illustrates an example stated in *tQL4BIM*. In the query, an IFC instance file is loaded and all walls are selected (Line 2). The result is then restricted by applying a predicate to each entity. Only walls with an attribute entitled *Name* with the value "Wall-002" are retained in the process. To access an attribute of an entity, the point operator is facilitated in the predicate definition (Line 3). After receiving all windows with a second *TypeFilter* (Line 4), the sets of walls and windows are topologically analysed by the *Touch* operator. The result is saved in a relational variable which comprises pairs of a wall and its touching window.

This demonstrates the main pattern of *tQL4BIM*: Repeated variable assignment combined with the calling of an operation. In this way, subsequent operations process variables assigned by previous calls. Despite its simplicity, this pattern enables a flexible and detailed analysis of building models by chaining manageable operations. The complexity of information retrieval is hidden in the interior processing of each operator. As a result, the domain expert can concentrate on the semantics of each operator instead of on low-level data handling.

4 Typing in the QL4BIM System

The type system of this novel QL4BIM version comprises four native types. These are *Constant*, *Entity*, *Set* and *Relation*. Denotation as native types highlights the presence of other types in the system. These additional types are introduced by the data model under examination. Currently, the *QL4BIM System*

Listing 3 A ᵗQL4BIM query which selects partial rooms with gross floor areas greater than 40 m²

```
1 model = GetModel("C:\Institute.ifc")
2 spaces = TypeFilter(model, "IfcSpace")
3 pSpaces = AttributeFilter(spaces.CompositionType = "partial")
4 rel1[space, prop] = PropertyResolver(pSpaces, model, "SpaceCommon")
5 rel2[space, prop] = AttributeFilter(rel1[prop].GrossPlannedArea > 40)
6 largeSpaces = Projection(rel2[space])
```

supports IFC files as data pool but the developed methods are not closely coupled to this data model. To enable an analysis of buildings using additional data on their surroundings, an extension to the CityGML model is in progress.

As mentioned above, a variable introduced in a ᵗQL4BIM query can be typed as *Entity*, *Set* or *Relation*. All of them act as wrappers for externally defined types stored in the current data pool. The following three examples demonstrate the interaction of the native and external types. (1) A variable can represent one entity, e.g. a wall. This is realized by a *QL4BIM Entity* which itself references an IfcWall. (2) A variable can refer to a set of entities. In this case a *QL4BIM Set* is used. This kind of variable acts as a container for (semantically-related) entities, e.g. windows which are located in the second storey. Internally, the set variable references *Entity*-instances each incorporating an IfcWindow. (3) A *QL4BIM Relation* is used to combine Entity instances and to express relationships between them. In contrast to a set variable, the relationship occurs between *Entity*-instances combined in a single tuple whereas many tuples are stored in one relation. A tuple refers to an ordered list of entities. A single *Entity* in the tuple can be accessed by a user-defined literal. To concretize the relational concept in use, **Listing 2** is considered once again. In Line 5, the variable rel[wall,window] is introduced, storing pairs of an IfcWall and its touching IfcWindow. The notation of literals for index access corresponds to the lower branch of the syntax diagram shown in **Figure 2**.

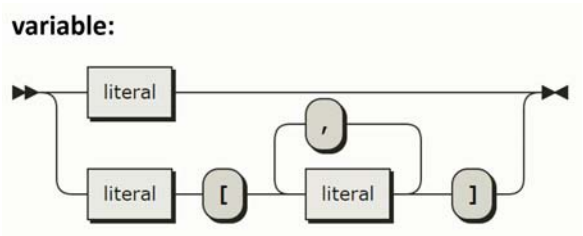

Figure 2 Syntax diagram of variables in QL4BIM

In the further processing, e.g. during predicate definition and parameter transfer, the window in each tuple can be accessed by rel[window].

A *QL4BIM Constant* is the base type for immutable and simple data structures. Like variables, constants are used as input for query operators. They are not fetched from the data pool but directly stated by the end user during query definition. Therefore, the values of these structures are independent from the data pool. *Constant* acts as a base type for *cstring*, *cnumber*, *cfloat* and *cid*. The first stands for strings and enumeration values. A *cnumber* represents an integer and *cfloat* a floating-point number. The last constant type *cid* is used to express a symbolic identifier. For the IFC model, a *cid* is an integer prefixed with a # token.

The query shown in **Listing 3** demonstrates the use of constants in *ᵗQL4BIM*. The query identifies all rooms attributed as partial which have a gross floor area greater than 40 m². In Line 3, the use of a *cstring* as an enumeration value is depicted. The attribute filter in Line 5 evaluates a *cnumber* in its predicate parameter.

One paradigm of *QL4BIM* is hiding its type handling. Therefore, the language is designed to cope without the need to explicitly type keywords and variables are simply introduced by initial use of a literal. The appropriate type of variable is inferred by the operator called in the assignment. So the distinction between *Entity* and *Set* variables is completely implicit. This is evidenced in **Listing 4** on the next page which presents a fragment from **Listing 2** together with a modification. In the original version (a), two sets are passed to the *Touch* operator. In modification (b), one *Set* and one *Entity* are handed over. In both examples, the textual syntax stays identical. Instead, the user can concentrate on the different semantics of the overloaded *Touch* operator. Section 5 depicts the concept of overloading operators in detail. Whereas there is no syntactical difference between entities and sets, the necessary definition of index literals for relational results and parameter handovers makes the use of relations more explicit. The presented approach

relieves the user in that the user is not required to handle native types in the language.

Listing 4 The implicity between entities and sets in the QL4BIM System

```
1a allWindows = TypeFilter(model, "IfcWindow")
2a rel[wall, window] = Touch(someWalls, allWindows)

1b theWindow = GetInstance(model, #2456)
2b rel[wall, window] = Touch(someWalls, theWindow)
```

This should also hold true if external types, e.g. from the IFC data model, are processed. Externally typed attributes are examined if predicates are passed to operators. Here the runtime has to convert external types to native types. To facilitate the definition of predicates, these conversions are performed automatically. This is done by recursively traversing the IFC schema until simple types of the *EXPRESS* language are reached. **Listing 5** containing fragments from the IFC4 schema, clarifies the used recursive processing. As shown here, the definition of IfcNonNegativeLengthMeasure is based on IfcLengthMeasure and finally ends up at the simple REAL type. As the query runtime incorporates a mapping for simple *EXPRESS* types to native *QL4BIM* types, the user is not restricted to explicitly instantiating an IfcNonNegativeLengthMeasure. If an entity attribute of this type is used in a predicate, a *cfloat* that maps to an *EXPRESS REAL* can be used during comparisons.

This is demonstrated by four applications of the *Touch* operator in **Listing 6**. In this case two sets (walls, windows) are extracted from the overall model. Additionally, the entity *wall2456* is fetched by the *GetInstance* operator. In the Lines 5-8, different overloads of the *Touch* operator are called. In the first variant the set of walls is used as a parameter. This results in a processing which finds all pairs of touching walls by evaluating the Cartesian product of set elements. In the second call of *Touch*, the instance *wall2456* is tested against each element of the window set. If *n* is the number of elements in the passed set, *n Touch* tests are performed. In Line 7, the operator is called with two sets. In this case only touching constellations between entities contained in different sets are examined. If *n* walls and *m* windows are passed, n^*m test are evaluated. Finally, a one-to-one test should also be supported by the system. In this case a relation containing pairs of entities is passed to *Touch* (Line 8). In the following topological

Listing 5 Fragments of the IFC4 EXPRESS schema used for type conversions

```
1 TYPE IfcNonNegativeLengthMeasure = IfcLengthMeasure
2 TYPE IfcLengthMeasure = REAL
```

5 Operator Overloading in tQL4BIM

As mentioned in the language paradigms, the user should be able to state queries at a high level of abstraction. No patterns for repeated execution of subqueries such as *for*-statements therefore exists in the language. Different query processing, each tuple will be examined individually. In this way *element1* is tested for touching *element2*. If *n* is the number of tuples in the relation, *n* tests are executed.

This example shows that different processing can be realized by simply changing the types of parameters passed to an operator. Thus,

Listing 6 Overloads of the Touch operator

```
1 model = GetModel("C:\Institute.ifc")
2 walls = TypeFilter(model, "IfcWall")
3 wall2456 = GetInstance(model, #2456)
4 windows = TypeFilter (model, "IfcWindow")
5 rel1[wall1, wall2] = Touch(walls)
6 rel2[wall, window] = Touch(wall2456, windows)       //creation of
7 rel3[wall, window] = Touch(walls, windows)          //rel0 is ommited
8 rel4[element1, element2] = Touch(rel0[element1, element2])
```

overloaded variants are available to the majority of *QL4BIM* operators. For example, the *AttributeFilter* is present in three versions and can execute on a single entity, a set or a relational variable. The amount of semantical difference between the overloads varies for each operator. A detailed description of all available operators including information on each overloaded version will be provided in the *QL4BIM Manual* which is currently being drafted.

6 The QL4BIM System and its Components

This section describes the runtime for interpreting *tQL4BIM* queries known as *QL4BIM System*. **Figure 3** shows the internal components of the runtime and their interrelationships.

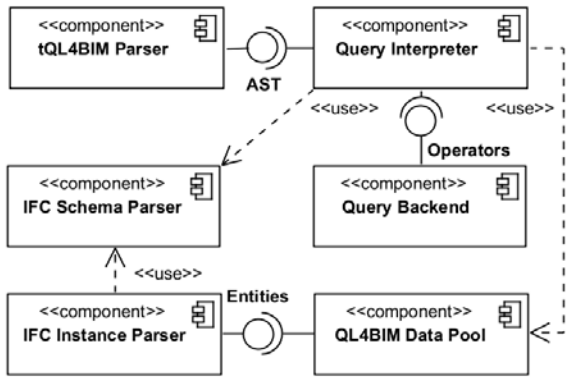

Figure 3 UML component diagram of the QL4BIM System

The system includes three parsers. They are used for the syntactic analysis of query input, for importing the underlying pool data and for examining the schema of the pool data. In general, a query is initiated by importing an IFC instance file to the current data pool via the *GetModel* operator.

The IFC instance parser is responsible for interpreting the referenced file and for creating an in-memory representation of IFC entities. The runtime uses a late binding approach to representing these external entities. Instead of specific classes for IFC types, entities are set up dynamically using generic parts. Fine-grained elements such as object identifiers, lists of symbolic references and values of simple types are stored in these parts. The late binding used has the advantage that all IFC versions can be supported ad-hoc. Additionally, the integration of further data models such as CityGML is facilitated.

The IFC schema parser supports query execution when external IFC types have to be compared with *QL4BIM* types (see **Listing 3**). Additionally, meta information such as attribute names of entities can be directly read out from the schema without the computationally expensive use of reflection mechanisms.

Once a query has passed syntactic analysis, the *tQL4BIM* parser translates it to an *abstract syntax tree* (AST) (Parr, 2010). This intermediate representation of a query is consumed by the query interpreter. It is a streamlined structure deduced from the textual query input without lexical helpers. Instead, a tree structure is set up that can be processed for query validation and query execution.

Figure 4 shows a *tQL4BIM* query and its representation as AST. The *S*-nodes correspond to the four subquery statements. As query execution is triggered by the user, the interpreter traverses the AST, collecting results and calling the appropriate function in the query backend with referenced variables. The imperative style of the language combined with the forced saving of interim results allows a step-wise query execution. Instead of processing all statements in the AST, the runtime interprets one statement after another, continuing by user request. In this

```
1 model = GetModel ("C:\Institute.ifc")
2 walls = TypeFilter (model, "IfcWall")
3 windows = TypeFilter (model, "IfcWindow")
4 r1[wall, window] = Touch(walls, windows)
```

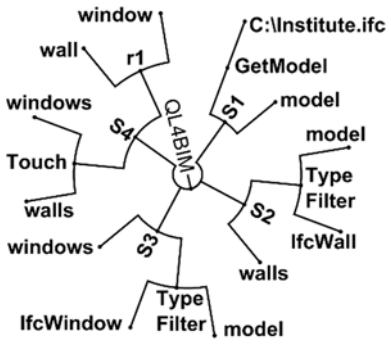

Figure 4 A query in tQL4BIM and its abstract syntax tree used by the runtime

step-wise execution, the content of the current intermediate variable can be shown in the *QL4BIM* user interface. This is done in a textual manner and extended by a 3D visualisation if the variable refers to geometrical information. The approach is similar to a debugging session in an IDE and a general programming language. It supports domain experts in defining complex queries as the influence of each single statement is testable.

7 The Visual Notion ᵛQL4BIM

A *Visual Programming Language* (VPL) uses a graphical notation to represent the intent of the user rather than typed code. The concept first emerged in the 1970s but is currently adopted increasingly in modern applications (Sutherland, 1966). In the domain of model-based planning, *Dynamo* and *Grashopper* are examples of VPL-equipped tools (Yazar, 2014). The general approach of a VPL is to place visuals on a canvas and to establish a connection between these items. A network structure is thereby set up. The semantic of this structure is specific to each VPL.

Nowadays, rich graphic user interfaces are the primary access points to applications for non-programmer. In contrast, users may face obstacles when it comes to textual programming languages. This holds true for domain experts such as architects and civil engineers and a language such as *ᵗQL4BIM* despite its minimal amount of patterns. Thus, a VPL which reuses the methodology of the presented textual query notation is introduced. It is called *ᵛQL4BIM* and the following characteristics and services are borrowed from its textual sibling:

- the general paradigms used in the design of *ᵗQL4BIM*,
- the native type system, the handling of external types and operator overloading,
- the intermediate representation of a query in the form of an AST.

For the graphical syntax, visuals are defined, representing constants, variables and operators (**Figure 5**). Constants are defined as rectangles, variables as rounded rectangles and operators as hexagons. The assignment of operational results and parameter handovers are both denoted by arrows. The connection between a relational variable and its index literals are defined by lines with diamonds at their endpoints.

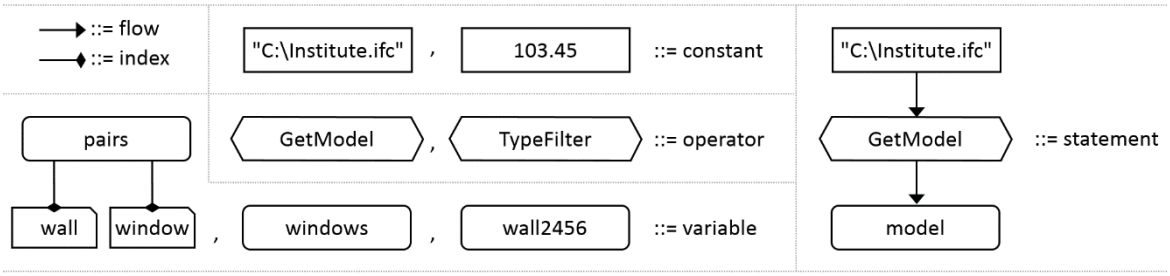

Figure 5 Visuals to express queries in ᵛQL4BIM

```
1 model = GetModel ("C:\Institute.ifc")
2 walls = TypeFilter (model, "IfcWall")
3 windows = TypeFilter (model, "IfcWindow")
4 r1[wall, window] = Touch(walls, windows)
```

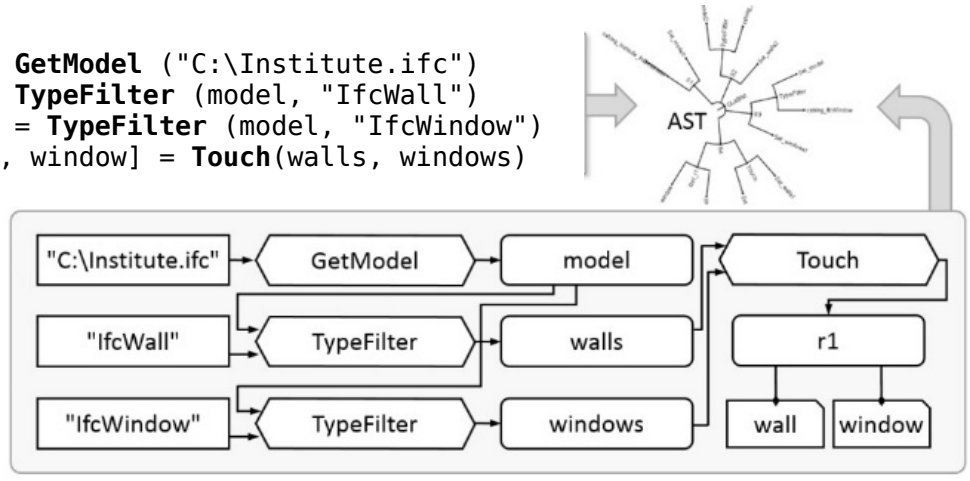

Figure 6 A textual and its equivalent visual query resulting in a common AST

The visual *ᵛQL4BIM* and the textual *ᵗQL4BIM* share the same intermediate representation in the form of an AST. The important fact is that a bijective relationship exists between a *ᵗQL4BIM* query and its AST. As a *ᵛQL4BIM* query and its resulting AST are also bijective connected, this tree structure can be used to synchronize both notations. Thereby an altering of the current query in both notation is possible at any time. If the query is changed in one notation, the obsolete representation is automatically renewed by traversing the AST and calling a visual, or respectively a code creation backend. **Figure 6** shows two equivalent queries: The first one is stated in *ᵗQL4BIM* with the second arranged by *ᵛQL4BIM* visuals. Both queries result in an identical AST.

8 Conclusion

BIM and its underlying data models, especially the IFC, have evolved over the past two decades. Today, they provide a well-grounded foundation for representing a wide range of buildings and the processes of their planning, construction and maintenance. On the other hand, there are gaps in analysis and data handling, particularly in relation to the formal exploration of models and the extraction of submodels.

This paper presents a novel query language in two notations that is designed to overcome these issues. The query language is focused on usage by domain experts and supports a straightforward, albeit formal and comprehensive examination of complex models. Together with the previously developed spatial and temporal operators of the system, a novel level of methodical support for analysing, verifying and processing of BIM data is achieved. This increases the efficiency of model-based processes and promotes a BIM-based methodology.

References

Borrmann, A. and Rank, E. (2009) 'Query support for BIMs using semantic and spatial conditions', Handbook of Research on Building Information Modeling and Construction Informatics: Concepts and Technologies: Concepts and Technologies, p. 405.

Codd, E. F. (1990) The relational model for database management: version 2, Addison-Wesley Longman Publishing Co., Inc.

Daum, S. and Borrmann, A. (2013) 'Definition and Implementation of Temporal Operators for a 4D Query Language', Proc. of the ASCE International Workshop on Computing in Civil Engineering.

Daum, S. and Borrmann, A. (2014) 'Processing of Topological BIM Queries using Boundary Representation Based Methods', Advanced Engineering Informatics, vol. 28, no. 4, pp. 272–286.

Eastman, C., Teicholz, P., Sacks, R. and Liston, K. (2011) BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors.

Liebich, T. (2001) 'XML schema language binding of EXPRESS for ifcXML', International Alliance for Interoperability.

Liebich, T. (2013) IFC4 specification [Online], no. 4. Available at http://www.buildingsmart-tech.org/ifc/IFC2x4/rc4/html/index.htm.

Mazairac, W. and Beetz, J. (2013) 'BIMQL – An open query language for building information models', Advanced Engineering Informatics, vol. 27, no. 4, pp. 444–456 [Online]. Available at http://www.sciencedirect.com/science/article/pii/S1474034613000657.

Parr, T. (2010) Language implementation patterns: Create your own domain-specific and general programming languages, Raleigh, N.C., Pragmatic Bookshelf.

Prud'Hommeaux, E., Seaborne, A. and others (2008) 'SPARQL query language for RDF', W3C recommendation, vol. 15.

Robie, J., Chamberlin, D., Dyck, M. and Snelson, J. (2013) 'XQuery 3.0: An XML Query Language', W3C Last Call Working, July.

Schenck, D. and Wilson, P. R. (1994) Information modeling: the EXPRESS way, Oxford University Press New York.

Sutherland, W. R. (1966) On-line Graphical Specification of Computer Procedures, DTIC Document.

Yazar, T. (2014) 'Design of Dataflow', Nexus Network Journal, pp. 1–15 [Online]. DOI: 10.1007/s00004-014-0222-8.