Analysis of possible frameworks for a graphical query editor in a web application

Daniel Haferkorn[a], Christian Kerth[b], Roland Rodenbeck[c], and Andre Schöbel[d]

[a,b,c,d]Fraunhofer IOSB, Fraunhoferstraße 1, 76131 Karlsruhe, Deutschland

ABSTRACT

Today's sensors and analysis systems produce huge amounts of data and one of the main challenges is how to enable users to find relevant data in time. Software systems supporting queries must provide suitable filter mechanisms. Through this filtering, the full dataset can be reduced to a manageable amount of relevant data. Respective filter criteria can be chained together in the form of Boolean expressions using disjunctions and conjunctions, and the resulting hierarchical structures can be further simplified by parentheses.

The practical use case that we present in this publication consists of a web application to access potentially large data sets using a defined set of metadata based catalogue entries. This application currently supports the specification of filter criteria, which can be concatenated by the user through a single global conjunction in a flat hierarchy. The underlying query language supports more complex queries using any combination of conjunctions, disjunctions and brackets. There is a user requirement to extend the expressiveness of the client search queries so that the full scope of the query language can be leveraged meaningfully. One problem is how such complex search queries can be graphically structured and visualized in a clear and comprehensive way.

There exist different approaches for the graphical visualization of program code and query languages. Some of these approaches also support the graphical editing of these representations by the user. Examples for such frameworks or tools are Blockly, Scratch or Node-RED. In this publication, an analysis of the applicability of such frameworks in the existing web application is presented. For this purpose, operational constraints and exclusion criteria that must be fulfilled for the use in our application are identified. This results in a selection of a framework for a future implementation.

Keywords: Graphical Programming Language, Database, Web Interface, Boolean Expressions, Filter Mechanisms, Complex Search Queries, Analysis

1. INTRODUCTION

Today's sensors and analysis systems produce huge amounts of data and one of the main challenges still is how to enable users to find relevant data in time. Users want to be able to search data according to the capabilities of the system. Such data is usually stored in a type of data storage such as a database or a data warehouse. Accessing such a data store using textual search queries can be very complex and susceptible to input errors. Accordingly, graphical user interfaces offer the possibility to significantly reduce complexity for the user and thus support him in formulating his query in a way that prevents errors. In this publication we discuss our thoughts on how to provide users with an easy way to present and edit complex search queries using a graphical representation.

We consider the use case of a web client implemented by us, which accesses a data storage system operated by us. The data storage system offers a search interface to access the data filtered based on the search criteria. The query interface functionality allows complex formulations of queries that seem to be difficult to model in

Further author information: (Send correspondence to Daniel Haferkorn.)
Daniel Haferkorn: E-mail: Daniel.Haferkorn@iosb.fraunhofer.de https://orcid.org/0000-0002-2258-7784
Christian Kerth: E-mail: Christian.Kerth@iosb.fraunhofer.de https://orcid.org/0000-0002-9851-1751
Roland Rodenbeck: E-mail: Roland.Rodenbeck@iosb.fraunhofer.de https://orcid.org/0000-0001-9148-3658
Andre Schöbel.: E-mail: Andre.Schoebel@iosb.fraunhofer.de https://orcid.org/0000-0003-2265-6505

an user interface (UI). The implemented web client does not provide all formulation options that are offered by the query interface. The aim of the publication is to investigate possibilities for extending the web client with a function for creating and processing search queries using graphical tools.

In the following section 2, we discuss our problem area in detail. Afterwards we present existing possibilities for graphical modeling in section 3 and introduce graphical code editors and other graphical modeling tools. For one of the presented tools we show an exemplary implementation in section 4. In the conclusion in section 5, we give a summary on the work discussed in this publication and an outlook on possible further work.

2. PROBLEM DESCRIPTION

In our use case we discuss the data storage implemented by us according to STANAG 4559 [1–3]. We also have developed a web-based graphical UI (GUI) that can be used to formulate search queries to access this data storage. The web client makes it possible to create filters that are being applied on the search interface of the data storage. The set of formulatable filter criteria is defined based on the data model with which the data storage is operated. Several filters can be linked together in the Web Client with AND conditions. The search interface itself expects that queries are formulated in a specific query language, the Boolean Query Syntax (BQS), that is defined in the standard AEDP-17 [4]. As the name implies, the BQS defines queries with a Boolean [5] structure, similar to queries in the Structured Query Language (SQL) that is used for querying relational databases. The following listing shows an example of a BQS query.

```
(NSIL_COMMON.type = 'IMAGERY' AND NSIL_CARD.sourceLibrary = 'ISAF_HQ')
OR (NSIL_COMMON.type = 'VIDEO' AND ( NSIL_PRODUCT:NSIL_FILE.extent >= 0.0 AND
    NSIL_PRODUCT:NSIL_FILE.extent <= 30.0 ) )
OR NSIL_COMMON.type = 'REPORT'
```

Listing 1: Example of a BQS Query

In general, filter criteria in a BQS query can be defined for all attributes that are defined as “searchable” in the data model. Depending on the type of attribute (e.g. text, number, timestamp, coordinate) different filter types are allowed. In the definition of the BQS, any combination of filter criteria by conjunctions (AND), disjunctions (OR) and negations (NOT) is possible.

The capabilities of the search interface and the semantic power of the BQS are therefore not fully utilized by the client, as it only supports to connect filters using conjunctions. Since it is desirable from the user's point of view to be able to define more meaningful search filters, the additional available language constructs of the BQS should be usable in the GUI of the client.

The problem with complex search queries has been considered for other access clients in the past. One such software was created to define complex filters in the synchronization between data storages. This application was implemented as a desktop application using the Netbeans Rich Client Platform [6]. That means its source code cannot be directly reused for the development of a web-based client. The experience gained with the implementation of the application at that time may well serve as a basis for the work ahead. Consider the following figure 1, which shows a UI for the definition of filters and an example of a complex filter definition.

Conjunctions and disjunctions are represented here in the form of boxes. To the right of them, sub-filters linked by them are displayed. On the far left is a disjunction containing two conjunctions and a filter can be seen. The filter contains an attribute product type with the value 'report'. The first conjunction describes a filter on the product type 'image', which is linked by the conjunction to a filter on the data storage source. The second conjunction describes a filter on the product type 'video', linked to a filter on a specific file size. With this, a hierarchy of filters is described by the disjunction and the conjunctions.

The client at that time already supported most of the functions that we want to have implemented in the current system. It enables logical combinations of conjunctions and disjunctions, the definition of restrictions for arbitrary and any number of attributes and the definition of filter criteria on attributes in any order (hierarchical classification as seen in 2a instead of faceted classification as seen in 2b).

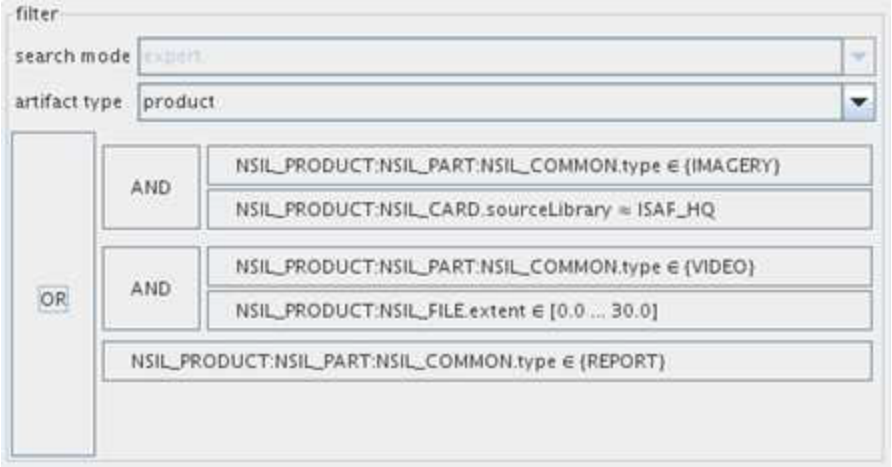
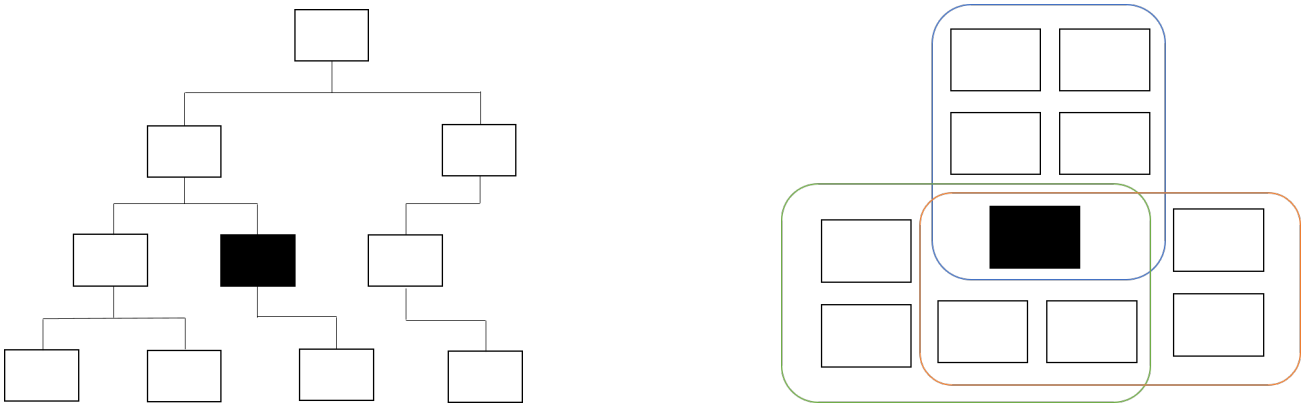


Figure 1: Legacy application using several filters



(a) Hierarchical search (b) Faceted search

Figure 2: Hierarchical search compared to faceted search

It has already been noticed in the old software that a drag & drop function would be useful in addition to the existing functions for creating and linking filters. This could be used to move sub-filters between conjunctions and disjunctions. Therefore this feature will be considered in the further development.

In the following chapter, the current state of the art will be analyzed in order to be able to select candidates for implementation.

3. STATE OF THE ART

The idea of creating queries using graphical visualization has been relevant for quite some time. Already in the early 90s, a possible visual query language was introduced in the publication "A Visual Query Language for Graphical Interaction With Schema-Intensive Database" [7]. The queries that can be generated with it can be specified as SQL [8] or Query By Example [9]. The visual language internally uses a prologue-like prediction based query language. Similar to our case, the authors use a schema-intensive data model. They stated that the level of complexity of a query language correlates with its associated data model. This increases the necessity for authors to create a simple database access option using a graphical query language while at the same time providing high semantic power. This confirms our intention to integrate a framework for a graphical query editor into our web application.

In the literature there is further work on Visual Query Languages and also evaluations of it. In the publication "User Interface Evaluation of a Direct Manipulation Temporal Visual Query Language" [10], the authors evaluated their Temoporal Visual Query Language (TVQL). The limitations and strengths of graphical programming

languages in a specific domain have been discussed the publication “Data retrieval from building information models based on visual programming”[11]. The authors conclude that a new intuitive mechanism for “data retrieval” has been found through graphical programming.

Research has also been carried out in this direction in recent years. This has lead to the development of several visual programming frameworks. An example for such a framework is called Node-RED [12]. In this framework, the programming is carried out using so-called flows that are depicted by networks comprised of nodes and edges. This framework is widely used in the context of Internet of Things [13 and 14]. Here, nodes represent inputs, outputs and processing steps. Individual nodes are connected to each other by edges. There can be several input and output nodes, which can be connected by a complex network of different nodes with different processing steps. Such flows are also presented in [5] as a sequence of individual filter executions, with the nodes here representing individual filters. The nodes can therefore be understood as representations of individual filters, bound together as conjunctions by the edges connecting the nodes. However, as shown in figure 3, an explicit representation of conjunctions as nodes is also conceivable. The advantage of this representation form is a uniform representation of conjunctions and disjunctions.

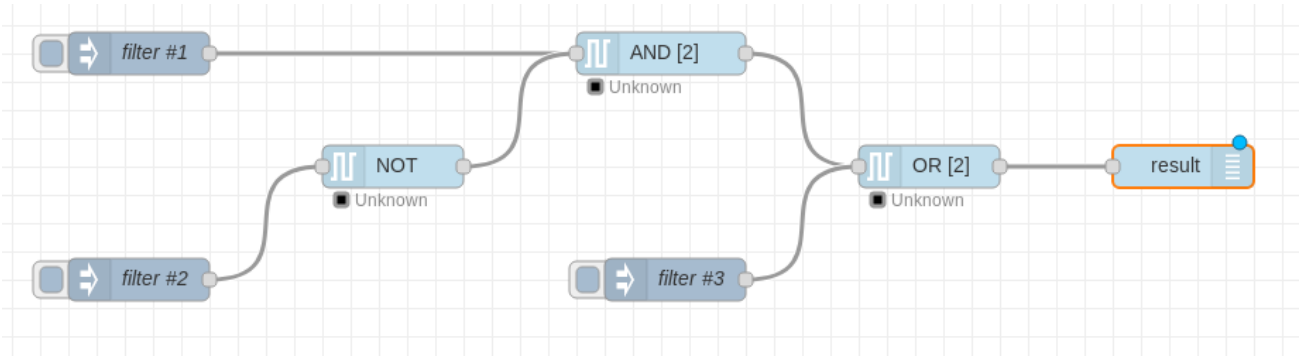


Figure 3: Example of a Node-RED flow using boolean nodes

A different framework for representing filters is Blockly, which was published by Google in 2012 as an open source project [15]. The Blockly framework is used in various projects, such as the programming language NEPO [16]. As the name implies, Blockly represents elements in the form of blocks that can be combined to form larger elements or function blocks. The blocks have different shapes and clearly defined connection points to visually show which blocks can be connected in which way. An example of different Blockly blocks is shown in figure 4.

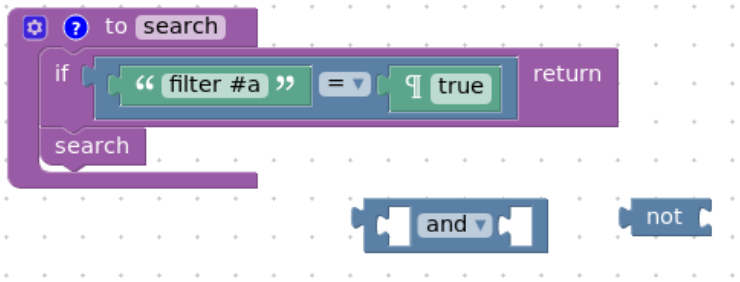


Figure 4: Example of some Blockly blocks

Since Blockly was originally developed for graphical programming, the standard blocks deal with corresponding code structures, such as variables, texts, mathematical formulas, loops and boolean logic. However, it is also possible to define new block types with the framework. In Blockly one can define input- and ouput-types for a block which means that two blocks can only be connected if their input- and output-type match.

This is a big difference to Node-RED, where in general every node can be connected to every other node and it is only defined whether a node has inputs, outputs or both. Due to its role as a graphical programming

language, Blockly provides framework conditions that are more suitable for graphical query languages when compared to Node-RED.

In addition to Node-RED and Blockly, there is another form of representation for multi-level filters, so-called Faceted Searches [17]. Such filters are often used on shopping portals, but also for searching online catalogues of museums and other indexed content. They allow multi-dimensional cuts through a search catalogue to be performed sequentially and a so-called "drill-down" into the result set. It is important that there is a search index for the different dimensions, which can be used to create the facets. As a rule, facets are linked exclusively via conjunctions. They therefore do not meet the complexity requirements necessary for the complete presentation of a BQS, which can define an arbitrary hierarchy of conjunctions and disjunctions. Therefore, we will not consider the Faceted Searches further in the following. In the next section we present an example implementation for Blockly with own block types suitable for the BQS instead of the Blockly basic types.

4. IMPLEMENTATION

In the previous chapter, various possibilities were presented to program visually, e.g., with Node-RED. A prototypical application using the Blockly library will be introduced in this section.

Blockly [15, 18] is an open source library to add an editor to an web application in that a user can program visually. It uses the visual metaphor of blocks that can be dragged around and combined with each other to form bigger and more complex structures. Like tiles in a jigsaw puzzle these blocks have connectors and slots where they can be connected. The resulting structures can then be processed by the framework to build code for many programming languages.

The decision was made in favor of Blockly because it was most suitable for the use case of visually formulating BQS expressions. Blockly contains the option to extend its code generation feature, making it possible to generate a different target format like BQS. Node-RED on the other hand defines processing chains for process flows. It can be adapted for own purposes, but since Blockly directly represents code blocks, we consider it more suitable than Node-RED for further examination.

4.1 Prototypical application to generate BQS using Blockly

Blockly was used to create a prototypical application which was named blockly-to-bqs (see figure 5). In this application users can drag and drop blocks into a workspace. These blocks then can be connected to build a complex query string for products on a data storage according to STANAG 4559 AEDP-17. The blocks in the workspace are being converted into a BQS string on the fly. The BQS is defined in the Extended Backus-Naur Form (EBNF) for the BQS (see J-2.1.3 BNF Definition [4]).

For our application creating the blocks for BQS was primarily done using the block factory of Blockly. The block factory is a Blockly app that allows a user to program custom blocks using Blockly blocks. It generates JavaScript code for the block and the generator which can be used in a Blockly application.

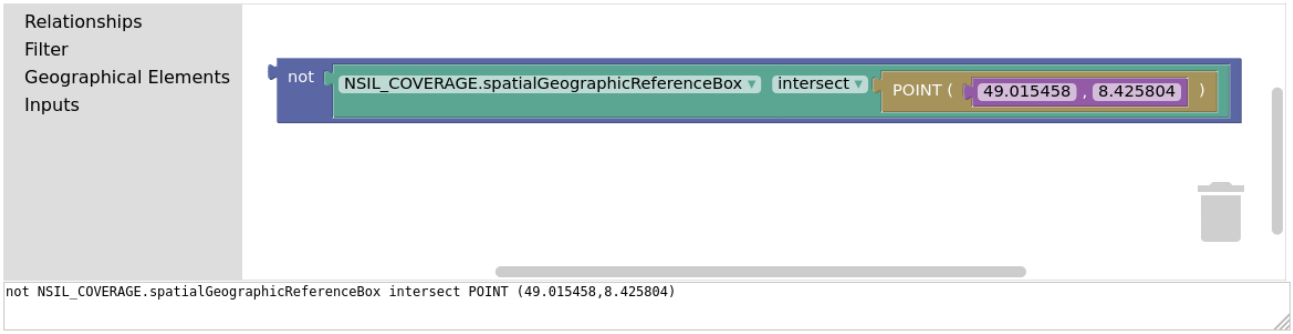


Figure 5: Prototypical application blockly-to-bqs

4.2 Blocks of blockly-to-bqs

In this prototype app there are four types of block categories which are: relationships, filter, geographical elements and inputs. Relationship blocks are relationships between filter blocks, e.g., the and-block connects two blocks with a logical and-conjunction. Filter blocks define filters for products based on a value of an attribute, e.g., the geographic filter block filters products based on the 'spatialGeographicReferenceBox' attribute. Geographical elements are geometric shapes in a geospatial context, e.g., the point-block specifies a point on earth by taking a geographic coordinate. Input blocks are blocks that provide various types of input, e.g., one can use the coordinate-block to specify the latitude and the longitude value in decimal degrees.

The attribute 'spatialGeographicReferenceBox' of the STANAG 4559 AEDP-17 datamodel describes a geographically referenced rectangle on earth, which determines the geographical location for a product. For further information about this attribute see table G-22 of AEDP-17 [4]. Consider the already created query in figure 5. The semantics of the BQS search is that all products should be queried where the point with latitude 49.015458 and longitude 8.425804 doesn't intersect the geographically referenced rectangle.

4.3 Implementation of blocks

In Blockly, there are three kinds of inputs which are value-, statement- and dummy-inputs (see figure 6). A value input (subfig. 6a) serves as a slot for another block. Whereas a statement input (subfig. 6b) has one connector that can be used to stack blocks. A dummy input (subfig. 6c) however doesn't have a connector or a slot. It can be used to add input fields like, e.g., text input fields or a drop down. A numeric input, e.g., was used in the coordinate-block to set the latitude and the longitude values.

(a) Value input (b) Statement input (c) Dummy input

Figure 6: Input types in Blockly

Consider listing 2 which is the JavaScript source code of the implementation of the not-block. The input's inline flag specifies whether the input blocks should be arranged horizontally or vertically. Each category defines their own color to distinguish between the four block types. Blocks can set the types of their output or can restrict the accepted value inputs via types. All relationship blocks specify the type q as their output type and take blocks of type q as their input value. The reason is that the BQS is recursively defined, which means, e.g., that an or-block accepts two other or-blocks as its input.

```
Blockly.Blocks['not'] = {
    init: function () {
        this.appendDummyInput().appendField("not");
        this.appendValueInput("Q").setCheck("q");
        this.setInputsInline(true);
        this.setOutput(true, "q");
        this.setColour(230);
    }
};
```

Listing 2: Source code of the not-block

4.4 Implementation of BQS generator

Generally the JavaScript generator from Blockly was used as a skeleton for the BQS generator. The basic API functions were adopted because at least these functions must be defined for the generator to work, e.g., the init- or the finish-function must be defined but their function body is empty. Other implementations of the basic API

functions are simple because they are not really needed in our context. E.g., the scrub function is used to handle comments but BQS doesn't allow comments. For each block there is a generator function, whose implementation is usually simple, if no transformations are necessary except concatenation of BQS strings. In case of Degree Minute Second (DMS) coordinates in which the EBNF of BQS requires a fixed length for the values, the generator functions become more complex. For example, the value 0 must be extended with an additional zero to 00 for the second value.

Consider listing 3 that shows the JavaScript code of BQS generator for the not-block. The input value, which is of type q, will be converted to a BQS string concatenated with the prefix "not". The constant *Blockly.BQS.ORDER_NOT* when calling the *valueToCode* function as well as when returning the code is the precedence value of the not-block, respectively the not-operator. The precedence value is used to automatically insert parenthesis in expressions according to the rules described in [19]. The operator precedences of BQS are defined in section J-2.2.5 of AEDP-17 [4].

```
Blockly.BQS['not'] = function (block) {
    var q = Blockly.BQS.valueToCode(block, 'Q', Blockly.BQS.ORDER_NOT);
    var code = "not " + q;
    return [code, Blockly.BQS.ORDER_NOT];
};
```

Listing 3: BQS generator of the not-block

4.5 Block design

BQS has no vertical elements like an if- or a loop-statement in a typical programming language. A BQS string is formulated as a horizontal sentence. Blocks of blockly-to-bqs application are therefore arranged horizontally rather than vertically to keep them consistent [18] with the BQS and thus enhance the learning effect.

4.6 Conclusion regarding Blockly app

One drawback of the design is, that the representation of the DMS-block and the generated BQS from that block diverges (see fig. 7). That means, the numeric input fields for the DMS values (see subfig. 7a) represent single-digit numbers with one digit but according to the EBNF the resulting BQS of single-digit numbers should have two digits (see subfig. 7b). Because of that, the visual representation of the DMS block and the behavior of the program is not conform to the expectation of the user. It also counteracts the learning effect for the user.

(a) DMS-block

```
00:00:00.0N,00:00:00.0N
```

(b) Generated BQS string from DMS-block

Figure 7: DMS-block with generated BQS string

It was also considered to reduce the number of blocks because users want to use as few blocks as possible to save manual effort to formulate queries. As an example, the input block for the point-block could use geographic coordinates per default instead of allowing the user to insert either a coordinate-block or a DMS-block.

As was mentioned before, all blocks in the application are arranged horizontally rather than vertically. Large BQS expressions with many blocks are growing horizontally than vertically. For test purposes, to keep the BQS expressions more compact, we have programmed the relationship blocks like the or-block to be arranged vertically and not horizontally. The resulting new arrangement can be viewed in figure 8 and 9.

With more complex nesting of multiple or- and and-blocks the generated BQS string can no longer be seen intuitively. Nevertheless, it is easier for the user to visually see which relationship blocks belongs to another block.

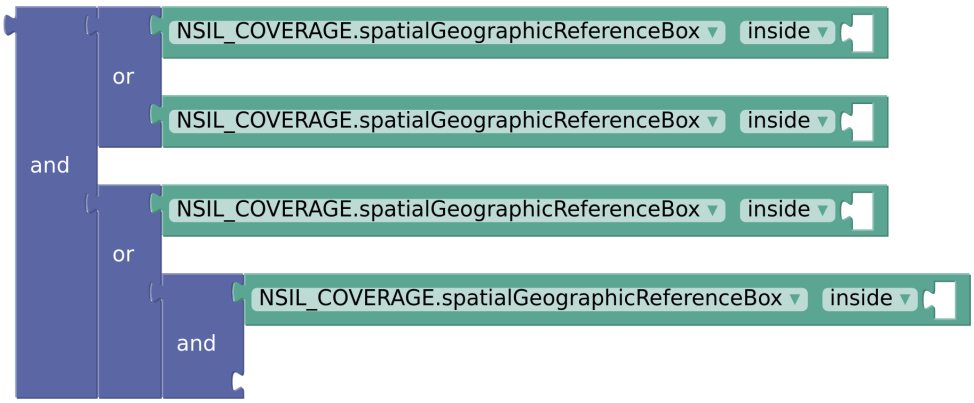


Figure 8: Or- and and-block vertically aligned

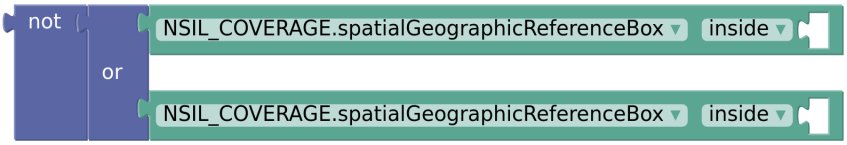


Figure 9: Not-block vertically aligned

5. CONCLUSION

In this publication an analysis for a graphical query editor was carried out. Various frameworks and applications were examined with regard to their suitability in our web application. First the imperative behind such a functionality was worked out and the state of the art was considered. Blockly turned out to be a possible candidate, and its usability was subsequently examined in more detail. A prototype implementation with Blocky was also carried out for this purpose. This implementation turns out to be technically straightforward to develop. In section 4.6 we evaluate this implementation. Blockly is therefore suitable as a framework for further use in our web application. Nevertheless, the usability is not optimal in the current prototype implementation of the blocky-to-bqs app. Especially creating very large and complex queries can be exhausting due to the large amount of blocks that need to be added and configured. Therefore, further work should be done on the usability experience and users should be involved in the process if possible.

The work on such a graphical query editor turns out to be useful and should be continued. Furthermore, our implementation needs to be tested by users. The purpose of this test is to determine the benefit and acceptance of such a graphical query editor. Ideally, such a test should be carried out as an A/B [20] test for each of the possible forms of presentation. In an A/B test a group of users is randomly divided into two subgroups. Each of the subgroups is shown a slightly different version of the user interface. The result of such test may then be integrated back into the implementation to improve the user experience.

Further work would be an analysis between search support and a learning tool BQS for later text input. This could be implemented, for example, in an auto-completion feature. In addition, further prototypical implementation of another framework (e.g. Node-RED) could also be carried out. This could then also be tested and compared on users. We expect to discover previously unknown advantages of these two frameworks with this approach.

REFERENCES

[1] NATO Standardization Office (NSO), "STANAG 4559." https://nso.nato.int/nso/nsdd/stanagdetails.html?idCover=8838 (2018). (Accessed: 24 March 2020).

[2] Essendorfer, B., Kerth, C., and Zaschke, C., "Evolution of the Coalition Shared Data Concept in Joint ISR," in [*Proceedings of the NATO IST/SET-126 Symposium on Information Fusion (Hard and Soft) for Intelligence, Surveillance & Reconnaissance (ISR), Joint Symposium IST-106 and SET-189*], (2015).

[3] Zaschke, C., Essendorfer, B., and Kerth, C., "Interoperability of Heterogeneous Distributed Systems," in [*Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security, Defense, and Law Enforcement Applications XV*], **9825**, 98250Q, International Society for Optics and Photonics (2016).

[4] NATO Standardization Office (NSO), "NATO STANDARD ISR LIBRARY INTERFACE-AEDP-17." https://nso.nato.int/nso/nsdd/apdetails.html?APNo=2272 (2018). (Accessed: 07 February 2020).

[5] Young, D. and Shneiderman, B., "A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation," *Journal of the American Society for Information Science* **44**(6), 327–339 (1993).

[6] "Netbeans Rich-Client Platform." https://netbeans.org/features/platform/ (2020). (Accessed: 24 March 2020).

[7] Mohan, L. and Kashyap, R. L., "A Visual Query Language for Graphical Interaction With Schema-Intensive Databases," *IEEE Transactions on Knowledge and Data Engineering* **5**, 843–858 (Oct 1993).

[8] "ISO/IEC 9075-1:2016 [ISO/IEC 9075-1:2016] Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)." https://www.iso.org/standard/63555.html (2016). (Accessed: 10 March 2020).

[9] Zloof, M. M., "Query-by-Example: A Data Base Language," *IBM Systems Journal* **16**(4), 324–343 (1977).

[10] Hibino, S. and Rundensteiner, E. A., "User Interface Evaluation of a Direct Manipulation Temporal Visual Query Language," in [*Proceedings of the fifth ACM international conference on Multimedia*], 99–107 (1997).

[11] Preidel, C., Daum, S., and Borrmann, A., "Data Retrieval from Building Information Models Based on Visual Programming," *Visualization in Engineering* **5**(1), 1–14 (2017).

[12] "Node-RED Guide." http://noderedguide.com/ (2020). (Accessed: 10 March 2020).

[13] Lekić, M. and Gardašević, G., "IoT Sensor Integration to Node-RED Platform," in [*2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*], 1–5 (March 2018).

[14] Rajalakshmi, A. and Shahnasser, H., "Internet of Things using Node-Red and Alexa," in [*2017 17th International Symposium on Communications and Information Technologies (ISCIT)*], 1–4 (Sep. 2017).

[15] "Blockly." https://developers.google.com/blockly (2020). (Accessed: 14 February 2020).

[16] Wulff, B., Wilson, A., Jost, B., and Ketterl, M., "An Adopter Centric API and Visual Programming Interface for the Definition of Strategies for Automated Camera Tracking," in [*2015 IEEE International Symposium on Multimedia (ISM)*], 587–592 (Dec 2015).

[17] Hearst, M., "Design Recommendations for Hierarchical Faceted Search Interfaces," in [*ACM SIGIR workshop on faceted search*], 1–5, Seattle, WA (2006).

[18] Pasternak, E., Fenichel, R., and Marshall, A. N., "Tips for Creating a Block Language with Blockly," in [*2017 IEEE Blocks and Beyond Workshop (B B)*], 21–24 (Oct 2017).

[19] "Operator Precedence." https://developers.google.com/blockly/guides/create-custom-blocks/operator-precedence (2020). (Accessed: 16 March 2020).

[20] "A/B Testing." https://en.wikipedia.org/wiki/A/B_testing (2020). (Accessed: 24 March 2020).