A Query Language for OPC UA Event Filters

Florian Duewel, Andreas Ebner and Julius Pfrommer
Fraunhofer IOSB, Fraunhoferstraße 1, 76131 Karlsruhe, Germany
florian.duewel@iosb.fraunhofer.de, andreas.ebner@iosb.fraunhofer.de, julius.pfrommer@iosb.fraunhofer.de

***Abstract*—The OPC UA standard combines industrial communication with information modeling. In order to remove the need for continuous polling, OPC UA clients can create subscriptions to be notified about data changes and events. OPC UA Event Filters provide powerful constructs for the server-side selection of relevant events. This feature is however underused in practice, also because of the complexity of Event Filters. Event Filters are expressed in a "byte-code" encoding. Assembling this encoding by hand is difficult and error-prone. This paper discusses the semantics of OPC UA Event Filters and proposes a query language to facilitate their definition. A formal grammar for the query language is provided in the BNF format. An implementation of a corresponding parser is available as part of the open62541 library.**

I. Introduction

OPC UA [1] defines not only a communication protocol, but also object-oriented information modeling. A client can then interact with the information model over the network. An important feature of OPC UA are subscriptions for the asynchronous notification of data changes or events to a subscribed client. OPC UA events have a wide range of applications, such as the initiation of transitions in behavioral state machines, returning results from asynchronous operations, or in general communicating changes within an underlying system that is represented by an OPC UA server.

In complex applications, servers can emit events at a high frequency. This can lead to an information overflow on the network and in the client-side capacity for handling them. To alleviate this, OPC UA defines a mechanism for the server-side filtering of events. This reduces the resource consumption, as it prevents unnecessary communication.

The power of OPC UA Event Filters is however underutilized in practice. Although powerful, Event Filters are currently difficult to use. It is quite cumbersome to create the data-structure representing a filter "by hand", and there exists little tooling for handling the full complexity of Event Filters. In this work we introduce a query language to make OPC UA Event Filters more usable. The query language is complete in the sense that any valid Event Filter can be represented by it.

The remainder of this paper is structured as followed. Section II reviews similar approaches where language based approaches are defined for event handling. Afterwards, Section III specifies OPC UA Events, as well as mechanisms to access them. Section IV details the concept of OPC UA Event Filter, explaining its logic and key elements. Section V then introduces our query language, which simplifies the generation of OPC UA Event Filter. Finally, section VII sums up our approach and gives a short outlook about possible extensions.

II. Related Work

A range of OPC UA companion specifications define Event-Types. For example the AutoID companion specification that transfers the information read from RFID tags via events [2].

The authors of [3] describe an early application of OPC UA for complex event processing. The work of [4] combines OPC UA events with SPARQL-based queries to retrieve additional information from the underlying information model.

In [5], Goldschmidt et al describe a DSL (Domain Specific Language) for the easy generation of OPC UA Event Filters. Their work is based on the LINQ syntax from the C# language ecosystem. While promising, their DSL incomplete as of their publication: (i) There is no syntax defined for the Simple-AttributeOperand data structure defining the select-clause of an Event Filter; (ii) reuse of partial filter results via the element-index operand is possible in OPC UA but not described in the DSL, (iii) the authors do not provide a formal BNF definition [6] for their DSL.

Besides Event Filters, the OPC UA specification defines a full query service that uses many similar concepts. The query service however has not been implemented by any of the common OPC UA SDKs to date. Some authors have developed a mapping between OPC UA and ontologies [7], [8], opening the way for query mechanisms such as SPARQL.

Besides event-based industrial communication, also the control logic for industrial automation can be event-based, for example in IEC 61499 [9]. Also the proposed OPC UA companion specification for IEC 62541 uses events [10].

Some communities have previously developed DSL for event handling. For example for the monitoring of distributed systems [11], streaming-based Complex Event Processing (CEP) [12] and in object-oriented databases [13]. The latter are conceptionally closest to our work as OPC UA information models are essentially an object-oriented database. Differently, the "active" object-oriented databases are themselves consumers of internally emitted events and link them to an action in the ECA (Event-Condition-Action) paradigm. In OPC UA, the receivers of events are mainly clients that are connected over the network. Although the local processing of events is often also supported, there are no dedicated features for local event handling in the OPC UA standard.

III. OPC UA Events

The subscription mechanism of OPC UA supports the monitoring of either DataChanges (changes to node attributes in the information model with a sampling interval) or Events. Latter are "discrete" and not subject to sampling. Their semantics is closely coupled to the object-oriented information model in OPC UA: EventTypes are special ObjectTypes that define the variables and child objects that an instance of the EventType must contain. Hence, Events can be thought of as temporary OPC UA objects that carry the information of the Event.

The source for each OPC UA Event is the BaseEventType, which defines all general characteristics of Events. From this BaseEventType, OPC UA defines a set of SubTypes, whereat each of these SubTypes serve a different purpose. SystemEventTypes are generated in case that some server-event occurs within the server, or the underlying system. The ProgressEventType indicates the progress of an operation, the AuditEventType results from an action by a client on the server, ModelChangeEvents occur in case of changes within the AddressSpace and SemanticChangeEvents are related to semantic modification in the AddressSpace of the server. Each EventType supported by a server is represented as an ObjectType in its AddressSpace without a particular NodeClass associated to it [14].

OPC UA Events provide a set of standard attributes such as a unique EventId, the Time when the Event was emitted, a Severity that indicates the urgency of the Event, or a Message to characterize the Event, among others. Since OPC UA information modeling allows TypeDefinitions and inheritance between types, all aforementioned types of Events can be extended and customized. Here, vendors can extend the existing attributes of Events with arbitrary information and thus, provide context to the Event's occurrence. Besides enriching Events with additional context, the definition of new EventTypes may also support the categorization of Events by introducing a new semantic instead of a new context [15].

Figure 1 illustrates the definition of a custom EventType. The *DrillEventType* is defined in namespace 1 and is a SubType of the BaseEventType. Semantically, the new EventType is supposed to notify about the completion of a drilling operation. Besides inheriting the properties from the BaseEventType, it introduces a set of five new properties. The *PartId* and the *OrderId* are supposed to identify the workpiece on which the operation was performed. The *Diameter* and *FeedForce* properties provide process parameter from the operation and lastly, the EventType has a *Result* property with a custom enumeration as type, to indicate whether the operation was successful or not.

Clients can access occurring events via the OPC UA Subscription Service [16]. Subscriptions report Notifications to clients through Notification Messages. Each of these messages carries a unique identifier, so that clients can check whether a Notification Message has already been processed. In general, subscriptions are designed to work independent of the actual communication between clients and servers, ensuring that data or Events are not lost in case of short communication interrupts. To receive the concrete information about a DataChange or an Event, clients attach MonitoredItems to their subscriptions.

Events are emitted by objects in the information model. Starting there, the Event "bubbles upwards" in the information

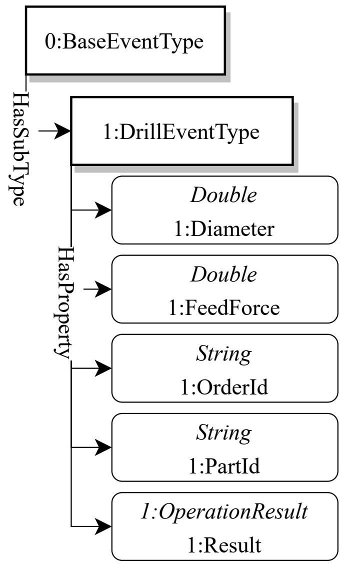


Fig. 1. Definition of the EventType DrillEventType and the Enumeration OperationResult

model hierarchy and is again emitted by each of its parents. A MonitoredItem for Events attaches to an object and listens for the Events emitted by the object. MonitoredItems might set filter criteria, such as Event Filter. Besides, MonitoredItems provide a Sampling Interval that determines the frequency with which the server samples an item. The filters evaluate each item after it is sampled and in case that their criteria are matched, a Notification is generated and then reported to the client-side via the subscription mechanism. The generated Notifications can be seen as data structures to describe the occurrence of DataChanges and Events.

IV. OPC UA Event Filters

Event Filters serve two purposes: (i) They select the fields that are to be transmitted in every notification and (ii) they define filter criteria that are evaluated server-side to decide whether an Event instance is relevant and shall be communicated. These two parts of the Event Filter are specified in the select-clause and the where-clause respectively.

A. Select-Clause: SimpleAttributeOperand

A select-clause is an array of *SimpleAttributeOperand* (SAO) data structures in the OPC UA type system. A SAO describes a value to be read from an event instance. The event is communicated to the client in a notification message that contains a *Variant* value for every SAO in the select-clause. The SAO datastructure contains the following elements:

TypeDefinitionId The NodeId of the EventType that describes the value to be read. All SubTypes of the specified EventType are valid as well.

BrowsePath Array of QualifiedNames that define a simplified BrowsePath. All entries in the BrowsePath are linked by (subtypes of the) hierarchical references.

AttributeId Attribute to be read from the target node

IndexRange Range of values to be read if the target is a (multi-dimensional) array.

B. *Where-Clause: Operators and Operands*

The where-clause is defined as a list of *ContentFilter-Element* values in the OPC UA type system. For simplicity we just call them filter elements. The list of filter elements resembles more to a byte-compiled interpreter code than to a human-readable filter expression.

Each filter element consists of an operator (an enum to a predefined function) and a list of operands. The operands are arguments for the application of the operator. The OPC UA Specification Part4: Services [16] defines 16 operators for Event Filter (Figure 8). Each operator takes one or more operands as its arguments. The filter operands are one of three distinct types:

ElementOperand (EO) The index of another filter element in the list.

Literal A value in a *Variant* container.

SimpleAttributeOperand (SAO) The SAO points to a value inside the event instance.

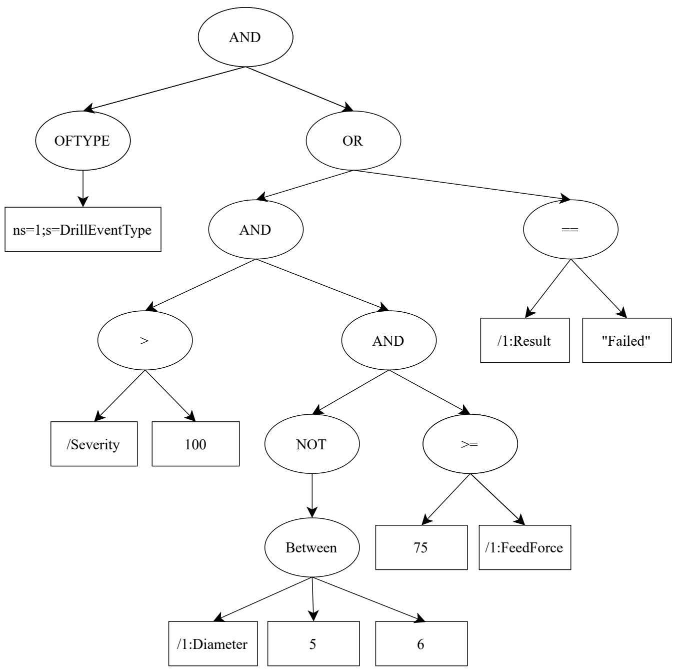
Using ElementIndex operands, multiple elements can be combined to a structured expression. Figure 2 contains an example where a ContentFilter is shown both as a tree-structure of nested expressions and serialized as a list of filter elements. Note that after the evaluation of the ContentFilter, the result of the first operator in the list determines whether the event is considered relevant or not.

The shown example uses a tree-structure where every element appears only once. It is however possible to reuse elements in multiple places via their ElementIndex. But it is not allowed to form cycles by chaining ElementIndices. This is ensured by the rules that an ElementIndex must always point to an index higher than the current element when using it for an operator.

The illustrated ContentFilter is related to the *DrillEventType* defined in Figure 1. This Event can be emitted in a context, where drill operations are performed on products within a shop floor. The filter criterion of the ContentFilter in Figure 2 can be describe as followed: Return the Event if it has the *DrillEventType* and either the drill operation has failed, or the drill operation was successful, but the feed force during the operation was greater than 75 or the diameter of the drilled hole is not between 5 and 6. An application scenario for such a filter could be the extraction of faulty parts from production, where the *OrderId* and *PartId* are returned from the occurring event, so that these parts are sorted out from a batch, or are post processed.

C. *Ternary Logic*

Like SQL, OPC UA Event Filters use ternary logic for truth-values. That is, every operand translates to either true/-false/null. The following truth tables for the underlying K_3 Kleene Logic [17] show how values are logically combined.



Index	Operator	Operands
0	AND	EO 1, EO 2
1	OFTYPE	Literal ns=1;s=DrillEventType
2	OR	EO 3, EO 9
3	AND	EO 4, EO 5
4	>	SAO /Severity, Literal 100
5	AND	EO 6, EO 8
6	NOT	EO 7
7	BETWEEN	SAO /1:Diameter, Literal 5, 6
8	>=	Literal 75, SAO /1:FeedForce
9	==	SAO /1:Result, Literal "Failed"

Fig. 2. Tree of a ContentFilter with its operator-operand relations and the corresponding serialization as a list of ContentFilterElements.

$\vee$	f	n	t
f	f	n	t
n	n	n	t
t	t	t	t

$\wedge$	f	n	t
f	f	f	f
n	f	n	n
t	f	n	t

$\neg$	
f	t
n	n
t	f

In the evaluation of a filter, the operands to a logical operator are first resolved and then explicitly cast to Boolean literal. Whenever this fails the operand becomes a null value. See Algorithm 1 for details.

D. *Event Filter Evaluation*

The OPC UA specification defines the ContentFilter structure but not a formal semantics for their evaluation. For better understanding, we show in Algorithm 1 the core evaluation algorithm for the where-clause. The content filter is an ordered list of elements $E = [e_1, e_2, \ldots, e_N]$. The elements $e = (f, O)$ are comprised of a filter operator f with operand arguments O. The results array R carries either a literal or a ternary truth value of every element. In our notation we write $R[i]$ to access element i of the ordered list R.

By evaluating the elements "backwards" it is ensured that resolving an ElementIndex to a results entry always succeeds. Because an ElementIndex can only point to a higher index that was already evaluated before. As described before, the result of the first element $R[0]$ defines whether the event is considered relevant overall.

Besides the formal definition we want to point the readers to our implementation of OPC UA Event Filters in the C-language as part of the open62541 project.[1]

Algorithm 1 OPC UA Event Filter Evaluation

Input: List of ContentFilterElements E with length $|E| = N$

```
// Initialize results array of length N
$R \leftarrow [\mathtt{n}, \ldots, \mathtt{n}]$
// Resolve operand to a literal value
procedure resolve(o)
    if $type(o) =$ ElementIndex then
        return $R[o]$
    else if $type(o) =$ SimpleAttributeOperand then
        return $read(o)$
    return $o$ // Already a literal
// Cast literal to ternary truth value
procedure ternary(o)
    if $o \in \{\mathtt{t}, \text{"true"}, \text{"1"}\}$ then
        return t
    else if $o \in \{\mathtt{f}, \text{"false"}, \text{"0"}\}$ then
        return f
    return n
// Evaluate operators from the back
for $i = N - 1 \ldots 0$ do
    $(f, O) \leftarrow E[i]$
    $O \leftarrow [resolve(o) : o \in O]$
    if $f \in \{\text{AND}, \text{OR}, \text{NOT}\}$ then
        $O \leftarrow [ternary(o) : o \in O]$
    $R[i] \leftarrow f(O)$
return $ternary(R[0])$
```

V. Proposed Event Filter Query Language

The Event Filter query language consists of three logical building blocks: the select-clause, the where-clause and the for-clause (Figure 3). The select-clause corresponds to the OPC UA Event Filter select-clause definition and sets a list of values to be returned from the event. The where-clause spans the Event Filter's ContentFilter. In cases where no where-clause is defined, the statement creates an event subscription without any filter criteria. The for-clause defines named query fragments to simplify long expressions (see Section V-C).

As an additional feature, the query language allows in-line comments (Figure 4), which can occur anywhere between recognized tokens (per the BNF definition). Comments are inserted in the same way as in C-code using either (i) two slashes to comment out the remainder of the line and (ii) a pair of `/*`, `*/` to comment out the whole section in between. The latter can create comments that span over multiple lines.

[1]https://github.com/open62541/open62541/blob/master/src/server/ua_subscription_eventfilter.c

```
EventFilter ::=
    SelectClause [WhereClause] [ForClause]
```

Fig. 3. BNF definition for the overall Event Filter

```
Comment ::= (LineComment | SectionComment)
LineComment ::= '//' <comment>
SectionComment ::= '/*' <comment> '*/'
```

Fig. 4. BNF definition for comments

A. Select-Clause

The select-clause is a comma-separated list of Simple-AttributeOperands that should be returned for the event in case that the filter criteria are met. When query fragments are used (cf. Section V-C) the parser needs to verify that all elements of the select-clause resolve to a SAO.

```
SelectClause ::= 'SELECT' SelectClauseElement
    [',' SelectClauseElement]

SelectClauseElement ::=
    SimpleAttributeOperand | FragmentName
```

Fig. 5. BNF definition of the select-clause

We propose a human-readable syntax for expressing Simple-AttributeOperands (SAO). It is feature-complete so that all possible SAO can be expressed with it. Figure 6 shows its formal BNF definition. The following are example SAO expressions.

- $\underbrace{\text{ns=2;s=TruckEvent}}_{\textbf{TypeDefinitionId}}\underbrace{\text{/3:Truck/5:Wheel}}_{\textbf{BrowsePath}}\underbrace{\text{[1:4]}}_{\textbf{IndexRange}}$
- $\underbrace{\text{/3:Truck/5:Wheel}}_{\textbf{BrowsePath}}\underbrace{\text{\#Value}}_{\textbf{AttributeId}}$
- $\underbrace{\text{\#BrowseName}}_{\textbf{AttributeId}}$

Even though all elements in the BNF definition of a SAO are optional, a statement must consist of at least a single / or # to be valid. In case that the TypeDefinitionId is not defined, its value defaults to the BaseEventType, in case of an undefined AttributeId, it defaults to the Value attribute of the target node.

The BrowsePath expression is modeled after the OPC UA RelativePath, for which a human-readable encoding is defined in Part 4, A2 of the standard. The QualifiedNames inside a BrowsePath can however contain characters that collide with the overall query expression, such as parentheses and commas. The RelativePath encoding in the standard defines the &-escaping of the special characters "`/ . <> : # ! &`". For the Event

Filter query language we extend the set of characters that need to be &-escaped to also include “`,()[] `” (the last character is a white-space).

```
SimpleAttributeOperand ::=
    [TypeDefinitionId] [SimpleBrowsePath]
    ['#' Attribute] [NumericRange]

TypeDefinitionId ::= NodeId

NodeId ::= [ns;=<uint16>]
    ( i=<uint32> | s=<escaped-string>
    | g=<guid>   | b=<base64-string> )

SimpleBrowsePath ::=
    '/' QualifiedName ['/'QualifiedName]

QualifiedName ::=
    [<uint16> ':'] <escaped-string>

Attribute ::= 'NodeId' | 'NodeClass' | ...

NumericRange ::= '[' RangeDimensions ']'

RangeDimension ::=
  <uint32> [':' <uint32>] [',' RangeDimension]
```

Fig. 6. BNF definition for the SimpleAttributeOperand

The example select-clause illustrated below determines two values to be returned from a filtered event. First, the Severity value of the event. This is defined in the BaseEventType, so we do not need to defined the TypeDefinitionId. Second, the value attribute of the PartId variable of the event is returned. The PartId variable is defined in the DrillEventType so we have to refer to the origin EventType in the TypeDefinitionId.

```
SELECT /Severity,
       ns=1;s=DrillEventType/1:PartId
```

B. Where-Clause

The where-clause statement shown below corresponds to the ContentFilter illustrated in Figure 2. Note that we leave out the `ns=1;s=DrillEventType` DataTypeDefinitions in the SAO to simplify the exposition. As a result, the TypeDefinitionId defaults to the BaseEventType. Consequently, the type definition of the SAO is not considered, since, as described in the OPC UA Standard [16], the SAO is applied in an Event Filter, so that the BrowsePath is directly evaluated instead. See the description of precedence and associativity rules later in this section.

```
WHERE OFTYPE ns=1;s=DrillEventType AND
      ((/Severity > 100 AND
        NOT /1:Diameter BETWEEN [5,6] AND
        75 >= /1:FeedForce)
       OR /1:Result == "Failed")
```

The where-clause contains a top-operator that goes first in the final list of filter elements and determines the overall result of the Event Filter during evaluation. If the select-clause uses multiple operators, then they must all be referenced (recursively) from the top-operator. If a fragment name is used for the top-operator, then it must resolve to an operator (cf. Section V-C).

```
WhereClause ::=
    'WHERE' (Operator | FragmentName)
```

Fig. 7. BNF definition of the where-clause.

Each operator takes a defined number of operands as its arguments (Figure 8). For the operands we allow the three types of operands from the standard (described in Section IV) and also fragment names (cf. Section V-C).

```
Operator ::= AndOp | OrOp | NotOp | UnaryOp
    | BinaryOp | BetweenOp | InListOp

AndOp ::= Operand ('AND' | '&&') Operand

OrOp ::= Operand ('OR' | '||') Operand

NotOp ::=  ('NOT' | '!') Operand

UnaryOp ::= ('ISNULL' | 'OFTYPE') Operand

BinaryOp ::=
    Operand (
        'GREATEREQUAL'  | '>='
      | 'LESSEQUAL'     | '<='
      | 'EQUALS'        | '=='
      | 'GREATER'       | '>'
      | 'LESS'          | '<'
      | 'BITAND'        | '&'
      | 'BITOR'         | '|'
      | 'CAST'          | '->'
      | 'LIKE'
    ) Operand

BetweenOp ::= Operand 'BETWEEN'
    '[' Operand ',' Operand ']'

InListOp ::= Operand 'INLIST'
    '[' OperandList ']'
OperandList ::= [OperandList ','] Operand

Operand ::= '(' Operand ')'
    | SimpleAttributeOperand | Literal
    | Operator | FragmentName
```

Fig. 8. BNF definition of operators and operands.

Any operand of a where-clause can be surrounded by parentheses. This also makes the order of application for nested operators explicit. In the absence of parentheses the following operator precedence (order of application) is applied:

$$\text{OR} \prec \text{AND} \prec \text{NOT} \prec \text{Binary, BETWEEN, INLIST} \prec \text{Unary}$$

Operators of the same precedence are left-associative:

$$A \text{ AND } B \text{ AND } C \quad \text{is parsed as} \quad (A \text{ AND } B) \text{ AND } C$$

The following example where-clause includes four nested operators. The resulting ContentFilter table with ElementOperands of operator indices is shown in the table right after.

```
WHERE NOT /1:Diameter BETWEEN [5,6] AND
      75 >= /1:FeedForce
```

Index	Operator	Operands
0	AND	EO 1, EO 2
1	NOT	EO 3
2	BETWEEN	SAO /1:Diameter, Literal 5,6
3	>=	Literal 75, SAO /1:FeedForce

Literals (Figure 9) are static values to be used as operands. The query language allows different notations for literals. They are defined either as a typed literal and preceded by the type name, as a direct literal where the type can be uniquely detected (direct literal integers are parsed as 32-bit signed integers), or as an OPC UA Variant value in JSON encoding. (Variants are containers that can carry values of any OPC UA DataType.)

```
Literal ::=
    TypedLiteral | DirectLiteral | <JSON>

TypedLiteral ::=
    ['BYTE' | 'SBYTE' | 'INT16' | 'UINT16' |
     'INT32' | 'UINT32' 'INT64' | 'UINT64']
     <integer>
  | ['FLOAT | 'DOUBLE'] <float>
  | 'STATUSCODE' (<escaped-string> | <uint32>)
  | ['BOOLEAN'] ('true' | 'false')
  | ['STRING'] '"' <c-string> '"'
  | ['BYTESTRING'] <base64-string>
  | ['NODEID'] NodeId
  | ['EXPANDEDNODEID'] ExpandedNodeId
  | ['DATETIME'] <datetime>
  | ['GUID'] <guid>
  | ['QUALIFIEDNAME'] QualfiedName
  | ['LOCALIZEDTEXT] LocalizedText

DirectLiteral ::=
    '"' <c-string> '"' | <float> | <integer>
  | ('true' | 'false') | NodeId
```

Fig. 9. BNF definition for Literal operands.

C. For-Clause

The for-clause can be used to simplify the select- and where-clause statements by pulling out query fragments and giving them a human-readable name. After resolving the query fragments, the for-clause disappears and is not part of the parsed Event Filter.

Query fragment names use the dollar sign as prefix. The fragment names are assigned using a ":=" expression. Query fragments can also be nested – of course avoiding infinite recursion. If a query fragment that defines an operator is used in multiple places, then the fragment is not duplicated in the final Event Filter. Instead the operator from the fragment is inserted only once and referred to from multiple places with an ElementOperand. So the reuse of query fragments enables filter expressions that are not tree-structures (but still acyclic).

```
ForClause ::= 'FOR' FragmentList

FragmentList ::=
    Fragment [ ',' FragmentList ]

FragmentName ::= '$' <escaped-string>

Fragment ::= FragmentName ':=' Operand
```

Fig. 10. BNF definition of the for-clause.

We now simplify the Event Filter from Figure 2 by the use of query fragments:

```
SELECT $sev,
       ns=1;s=DrillEventType/1:PartId

WHERE OFTYPE ns=1;s=DrillEventType AND
      (($gre AND $not AND 75 >= /1:FeedForce)
       OR $fail)

FOR $sev  := /Severity,
    $gre  := $sev > 100,
    $not  := NOT /1:Diameter BETWEEN [5,6],
    $fail := /1:Result == "Failed"
```

VI. Implementation Considerations

Figure 11 illustrates how a parser implementation can be attached to existing Event Filter implementations. Here, the parser implementation receives a query string that is first tokenized by a lexer. If the query string translates to a valid list of tokens, then the token are handed over to the parser, which translates it to the final Event Filter.

This returned Event Filter can then be deployed by a client, which creates a MonitoredItem with the filter. The grammar itself cannot ensure that a valid query results in an Event Filter that is valid for a particular information model. Here, a validation of the created Event Filter is performed by the server, which checks whether the requirements of operator-operand pairs are matched and that the SimpleAttributeOperands can be resolved. Also the operator operands are evaluated to determine their type and in case that their type does not match the status code "BadFilterOperandInvalid" is returned. Whenever the ObjectNode emits an Event, the attached MonitoredItem evaluates whether the filter criteria are matched and, if so, generates an Event Notification.

Our reference implementation is integrated with the open62541 OPC UA SDK [18]. The implementation is written in the C language and uses re2c [19] and Lemon (part of the SQLite project [20]) as the lexer and parser generator. Besides the integration with open62541, our implementation is available online in an HTML5 version at https://www.open62541.org/query-http/index.html. It translates queries to the Event Filter data structure in the JSON encoding. The resulting Event Filter can be used with any OPC UA SDK.

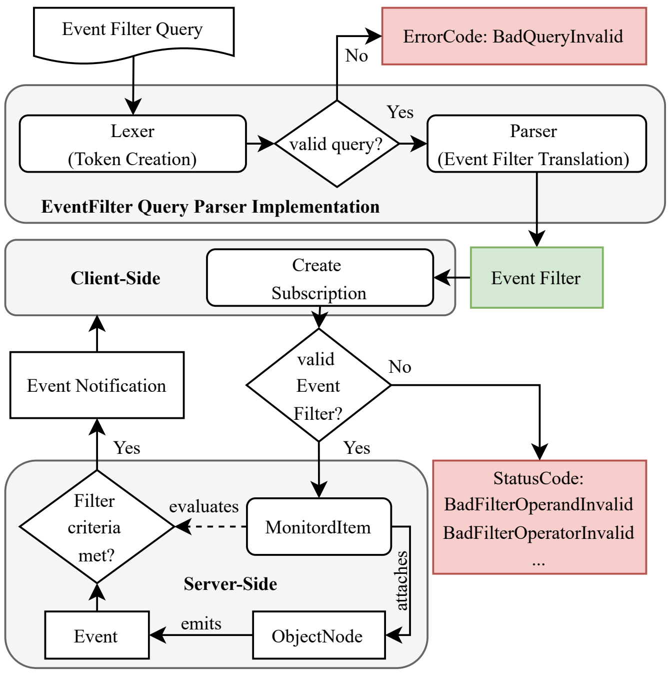


Fig. 11. Connection of the standalone Event Filter query language parser to existing Event Filter implementations of OPC UA SDKs

VII. Summary and Outlook

The paper has proposed a query language for OPC UA Event Filters together with its formal BNF definition. Queries consist of a select-clause, where-clause and for-clause. The query language is feature-complete, all possible OPC UA Event Filters can be expressed with it. Going forward several future extensions are possible.

OPC UA defines a full *Query* service. It uses the same concepts as Event Filters with some additional operators and operand types. We are not aware of any OPC UA SDK providing server-side query support at this moment. An exploration of the topic is hence worthwhile.

OPC UA DataAccess (Part 8) defines a *DataItemType* variable type. This can carry a (vendor-specific) human-readable expression from which the value of the variable is computed. The example given in the standard is

```
(TempA - 25) + TempB
```

The interpreter for OPC UA Event Filters could be extended with additional operators to support numerical expressions to compute the value of DataItemType variables.

References

[1] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Springer Science & Business Media, 2009.
[2] OPC Foundation, "OPC UA for AutoId Devices Release," 2021.
[3] M. J. A. G. Izaguirre, A. Lobov, and J. L. M. Lastra, "OPC UA and DPWS interoperability for factory floor monitoring using complex event processing," in *2011 9th IEEE International Conference on Industrial Informatics*. IEEE, 2011, pp. 205–211.
[4] T. Westermann, N. Hranisavljevic, and A. Fay, "Accessing and interpreting OPC UA event traces based on semantic process descriptions," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2022, pp. 1–7.
[5] T. Goldschmidt and W. Mahnke, "An internal domain-specific language for constructing opc ua queries and event filters," in *European Conference on Modelling Foundations and Applications*. Springer, 2012, pp. 62–73.
[6] ISO-14977:1996(E), "Information technology – Syntactic metalanguage – Extended BNF," ISO/IEC, 1996.
[7] J. Pfrommer, S. Grüner, T. Goldschmidt, and D. Schulz, "A common core for information modeling in the Industrial Internet of Things," *at-Automatisierungstechnik*, vol. 64, no. 9, pp. 729–741, 2016.
[8] R. Schiekofer and M. Weyrich, "Querying OPC UA information models with SPARQL," in *24th IEEE international conference on emerging technologies and factory automation (ETFA)*. IEEE, 2019, pp. 208–215.
[9] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.
[10] M. Majumder and A. Zoitl, "A Proposal for OPC UA Companion Specification for IEC 61499 Based Control Application," in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*. IEEE, 2023, pp. 1–8.
[11] M. Mansouri-Samani and M. Sloman, "GEM: A generalized event monitoring language for distributed systems," *Distributed Systems Engineering*, vol. 4, no. 2, p. 96, 1997.
[12] G. Cugola and A. Margara, "TESLA: a formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010, pp. 50–61.
[13] K. R. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva, "SAMOS in hindsight: experiences in building an active object-oriented DBMS," *Information Systems*, vol. 28, no. 5, pp. 369–392, 2003.
[14] OPC Foundation, "OPC Unified Architecture Part 5: Information Model," 2022.
[15] ——, "OPC Unified Architecture Part 3: Address Space Model Release," 2023.
[16] ——, "OPC Unified Architecture Part 4: Services Release," 2023.
[17] S. C. Kleene, "On notation for ordinal numbers," *The Journal of Symbolic Logic*, vol. 3, no. 4, pp. 150–155, 1938.
[18] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, "Open source as enabler for OPC UA in industrial automation," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015, pp. 1–6.
[19] P. Bumbulis and D. D. Cowan, "re2c: A more versatile scanner generator," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 70–84, 1993.
[20] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, "Sqlite: past, present, and future," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, 2022.