

Pan-European University, Faculty of Informatics

Interim Research Report

# Definition of query language for object-centric processes

of the project

## Requirements and formal definition of a low-code language based on object-centric processes - LowcodeOCP

Authors:

Gabriel Juhás, Milan Mladoniczky, Juraj Mažári and Tomáš Kováčik

Contact:

[gabriel.juhás@paneurouni.com](mailto:gabriel.juhás@paneurouni.com), [milan.mladoniczky@paneurouni.com](mailto:milan.mladoniczky@paneurouni.com)

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia  
under the project No.

09I03-03-V04-00493

May 15, 2026

Version: v0.1

## **Abstract**

Querying process models, running instances of processes, and logs of their execution is important for developers and analysts of process driven applications. In this report we present the design and implementation of a domain-specific query language for the object-centric language Petriflow. After we provide a brief overview of existing query languages and the motivation behind creating our own query language, we define a simplified version of the extended Backus–Naur form of the query language over objects of Petriflow classes. We also provide details of the query language implementation in the Netgrif Platform.

# Contents

|                                                |            |
|------------------------------------------------|------------|
| <b>Contents</b>                                | <b>i</b>   |
| <b>List of Figures</b>                         | <b>ii</b>  |
| <b>List of Tables</b>                          | <b>iii</b> |
| <b>1 Introduction</b>                          | <b>1</b>   |
| <b>2 Petriflow</b>                             | <b>3</b>   |
| 2.1 Motivation example . . . . .               | 4          |
| <b>3 Petriflow Query Language</b>              | <b>7</b>   |
| 3.1 Resource attributes . . . . .              | 9          |
| 3.2 PFQL query of motivation example . . . . . | 9          |
| <b>4 Implementation</b>                        | <b>11</b>  |
| 4.1 Filters . . . . .                          | 13         |
| <b>5 Future work</b>                           | <b>14</b>  |
| <b>6 Conclusion</b>                            | <b>16</b>  |

# List of Figures

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | Request process with a token in the place <i>submitted</i> waiting for approval.                                                                                                                                                                                                                                                                                                                                                                                                          | 4  |
| 2.2 | Action with a search query executed by pressing a button within <i>Bulk approval</i> task of a <i>Bulk approval</i> process. The implementation performs two main steps. First, it searches for instances of the <b>request_query</b> process where the <i>name</i> and <i>surname</i> attributes match those of the current <i>Bulk approval</i> case. Second, it retrieves the corresponding <i>Approve request</i> tasks (with ID equals <i>t2</i> ) for the identified cases. . . . . | 5  |
| 2.3 | After executing action in <i>Get requests</i> button the forms of <i>Approve request</i> tasks of cases with name and surname equal to name and surname of the <i>Bulk approval</i> are displayed as subforms of the <i>Bulk approval</i> task . . . .                                                                                                                                                                                                                                    | 5  |
| 4.1 | Example of a query language used in custom filter in the user interface of Netgrif Platform. . . . .                                                                                                                                                                                                                                                                                                                                                                                      | 13 |

# List of Tables

|     |                                                      |   |
|-----|------------------------------------------------------|---|
| 3.1 | List of data types and supported operators . . . . . | 9 |
|-----|------------------------------------------------------|---|

# Chapter 1

## Introduction

Searching and querying a database is an essential part of every application. It is important for developers to have a single, powerful query language to use. Process-driven applications have three distinct concerns for such query languages:

1. Querying process models - answering questions like *"In which processes is the 'Person verification' REST service called?"* or *"In which processes does the legal department participate?"*.
2. Querying instances of processes - answering questions like *"Which instances of the claim process were made using this email address?"* or *"How many instances of the claim process are in the state 'waiting for answer from legal department'"*.
3. Querying execution history of processes - answering questions like *"How many times was a claim made?"* or *"What is the average time for an employee to answer a customer's request?"*.

Many process query languages exist, but they mainly focus on querying processes (1.) and their execution history (3.). The Business Process Query Language (BP-QL) focuses on querying business processes, answering questions like *"Can I get a price quote without giving my credit card details first?"* [Bee+08]. Another query language, BPMN-Q, also addresses process definitions and searches for patterns within a repository of business processes, using queries like *"Activity B is immediately followed by activity C"* [Awa07]. Celonis PQL was designed for business users and translates their process-related business questions like *"How many purchase orders violated segregation of duties for activities 'Request Approval' and 'Grant Approval'?"* [Vog+22].

In this report, we provide a definition of our Petriflow Query Language (PFQL) using an Extended Backus-Naur Form (EBNF) and its implementation within the Netgrif Platform. A primary motivation for a query language specifically designed for Petriflow objects (classes) is the necessity to establish and manage dynamic relationships between different

process instances (cases). At this stage, PFQL is intentionally focused on the static state and the current marking of objects. This functionality is essential for identifying and retrieving specific cases based on their current data and state, as well as for locating tasks (transitions) required for synchronization. Furthermore, such querying capabilities are vital for UI composition, where retrieved tasks are dynamically integrated as sub-forms within parent process forms.

In the following sections, we provide definition of our Petriflow Query Language (PFQL) using an Extended Backus-Naur Form (EBNF), and implementation of the query language in the Netgrif Platform.

# Chapter 2

## Petriflow

Petriflow object-centric processes integrate *data attributes*, *lifecycle* in the form of extended Petri nets, where transitions represent activities and tasks, and *user interfaces* as forms that can be associated with tasks. Data attributes themselves are XML objects with type, id, title, validations, etc. (see petriflow.org for more details). As a lifecycle place/transition nets [DJ01; Des05; DR15], with inhibitor [AJ15], read [Vog02], and reset arcs [Aal+09] are used with possible variable weight of arcs given by values of data attributes of type number or markings of places, inspired by self-modifying nets [Val78]. Forms are subsets of data attributes together with an information whether a data attribute on a form is required, editable, or read-only.

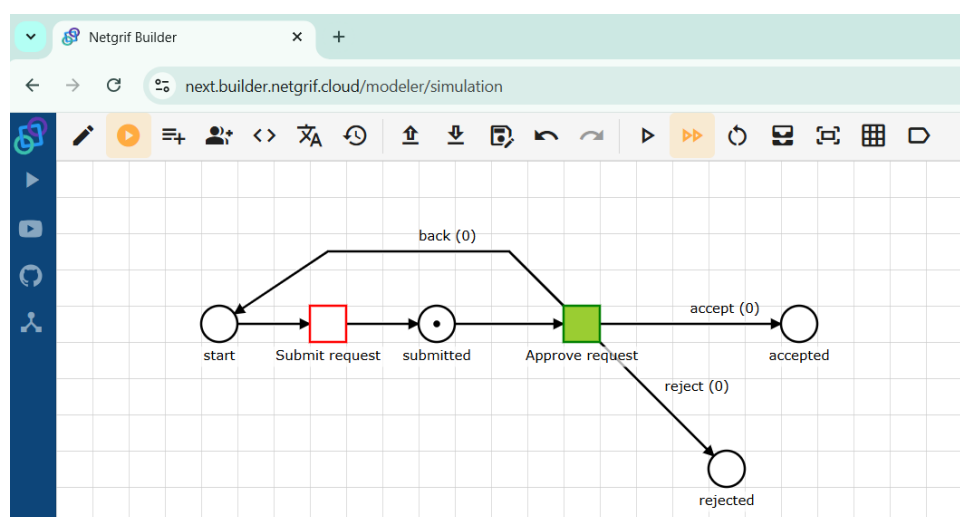
Relations between instances of object-centric processes are modelled on the data layer. Data attributes can reference one or many related instances similar to concepts of foreign keys in SQL databases, references in object oriented programming, or one-to-one and one-to-many relationships in entity-relationship or class diagrams.

Searching and retrieving data of related instances requires a use of a query language. Petriflow language does not strictly define which database should be used to store all data, therefore it needs its own query language. This query language should fulfil following requirements:

- database-agnostic - user should not be required to learn a specific database query language in different implementations that use Petriflow language,
- single data structure - user should not be concerned how are the processes and instances stored and which format or data structure,
- enhance low-code capabilities - the query language should be easy to learn and intuitive.

With these requirements in mind we designed a query language inspired by SQL as its queries are readable almost like sentences, they have a simple and structured notation and users do not need to know any specific data format like JSON.

For example, consider a *Bulk Approval* process containing a *Bulk Approval* task, designed to approve multiple requests simultaneously. The instance of *Bulk Approval* process may contain data attributes for a specific *name* and *surname*, such as '*John Doe*'. To function correctly, this process must establish a **one-to-many relationship** with all instances of the *request\_query* process depicted in Figure 2.1, that share the same attributes of *name* and *surname*. To create this link, the system must perform a lookup to identify and retrieve all relevant case objects. In this example, it is realized by an action triggered by pressing *Get requests* button in the *Bulk approval* task of the *Bulk Approval* process. The action containing the query is in Figure 2.2.



In this query we search for process instances of a process with *identifier* equal to *'request'*, data attribute *name* with value *'John'*, and data attribute *surname* with value *'Doe'*, we could use a *QueryDSL* framework for MongoDB:

```
findCases{
    it.processIdentifier.eq(workspace + 'request_query')
    .and(it.dataSet.get('name').value.eq(name.value))
    .and(it.dataSet.get('surname').value.eq(surname.value))
}
```

This query can be hard to read for a non-programmer and brings a risk of slow performance because data attributes (*dataSet*) are not indexed; in the worst case, MongoDB

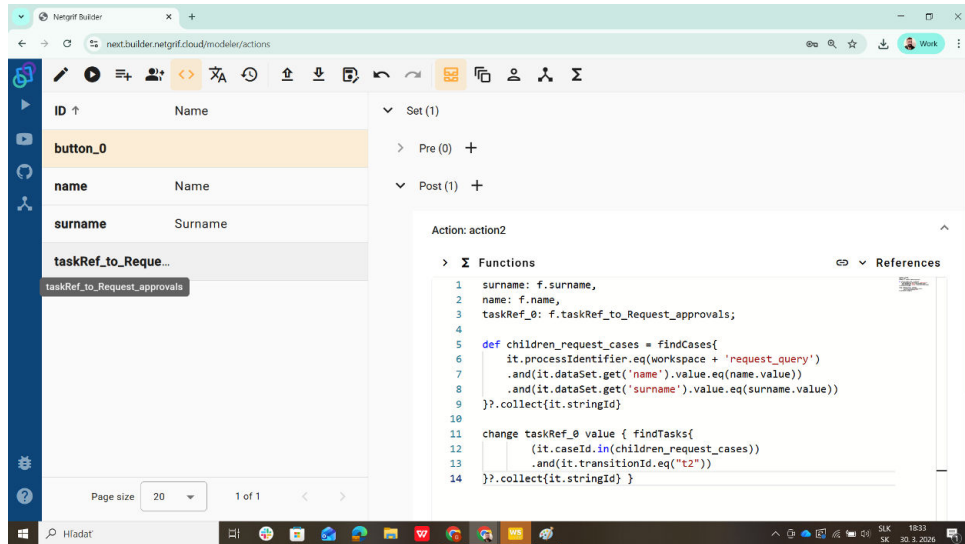


Figure 2.2: Action with a search query executed by pressing a button within *Bulk approval* task of a *Bulk approval* process. The implementation performs two main steps. First, it searches for instances of the *request\_query* process where the *name* and *surname* attributes match those of the current *Bulk approval* case. Second, it retrieves the corresponding *Approve request* tasks (with ID equals *t2*) for the identified cases.

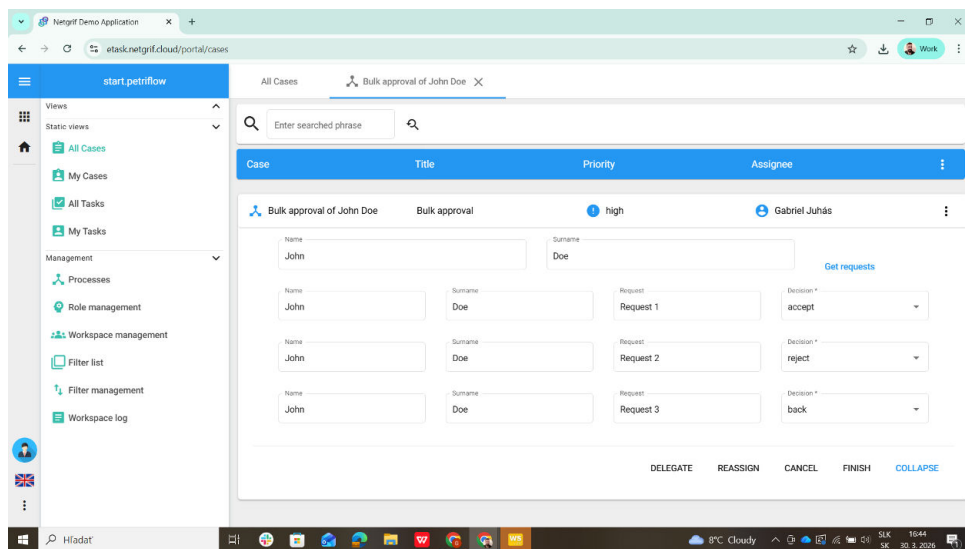


Figure 2.3: After executing action in *Get requests* button the forms of *Approve request* tasks of cases with name and surname equal to name and surname of the *Bulk approval* are displayed as subforms of the *Bulk approval* task

has to iterate over all records. To solve this problem, we would need to use Elasticsearch with its own *Query string* query:

```

findCasesElastic(
  "processIdentifier: '${workspace}request_query'
  AND dataSet.name.textValue.keyword: '${name.value}'
  AND dataSet.surname.textValue.keyword: '${surname.value}'")

```

This query is more readable, but the structure of stored data is different. A developer must know that for searching an exact string value, they can directly use the *processIdentifier* field but must use the specific *keyword* field for the data attributes' text value. A second query language makes the platform unnecessarily hard to learn and use, even with fairly simple queries. Therefore we define our own query language that unifies and simplifies queries in Petriflow.

## Chapter 3

# Petriflow Query Language

To showcase the composition of a PFQL query we provide a simplified version of the PFQL EBNF. This simplified version omits differences between resources and their attributes and uses special sequences (denoted by a pair of question marks) to define attributes and their values.

Searching for resources is the only goal of PFQL each query needs to define resource its searching and predicates that it needs to fulfil. Optionally the query can define paging and sorting.

```
query = statement resource where predicates [ paging ] [ sorting ]
```

PFQL defines statements *search* for retrieving resources, *count* for getting the number of resources that fulfil the query, and *exists* for finding if any resource fulfil the query.

```
statement = 'search' | 'count' | 'exists' ;
```

Petriflow defines following *resources*: process, instance of a process (case), task of an instance, user, and role[Juh+21]. To distinguish between searching for single resource and searching multiple resources we define both singular and plural form.

```
resource          = single_resource | multiple_resources ;
single_resource    = 'process' | 'case' | 'task' | 'user' | 'role' ;
multiple_resources = 'processes' | 'cases' | 'tasks' | 'users' | 'roles' ;
```

The traditional *where* clause can be simplified to a single ':' character. If a Petriflow interpreter provides interfaces for searching each resource directly than both *resource* and *where* clauses are redundant and could be omitted. For example calling `search("cases : processId eq 'request'")` would be equivalent to `searchCases("processId eq 'request'")`.

```
where = ' where ' | ' : ' ;
```

Query predicates allow to use logical operators *and* and *or* ('&' and '—' respectively) and their nesting using parenthesis. The nesting of predicates is defined through recursion in the EBNF.

```

predicates = term { or term } ;
term       = predicate { and predicate } ;
predicate  = comparison | ' ( ' predicates ' ) ' ;
comparison = attribute operator value ;
or         = ' or ' | ' | ' ;
and        = ' and ' | ' & ' ;

attribute = ? resource attribute ? ;

operator  = symbol_op | word_op ;
symbol_op = ' == ' | ' != ' | ' < ' | ' > ' | ' <= ' | ' >= ' ;
word_op   = ' eq ' | ' neq ' | ' lt ' | ' gt ' | ' lte ' | ' gte ' ;

value     = ? string or number ? ;

```

The list of common operators include equal (*eq* or '=='), not equal (*neq* or '!='), less than (*lt* or '<'), greater than (*gt* or '>'), less than or equal (*lte* or '<='), and greater than or equal (*gte* or '>='). Other operators such as in range (*in*( : )), in list (*in*( , )), and similar (*like* or '=') are defined only for specific data attribute types.

Iterating over a large amount of resources can be a performance heavy task that will require paging. The *paging* clause define a page number (starting at zero) and optional size of a single page. Both values are non-negative integers.

```

paging      = ' page ' page_number [ ' size ' page_size ] ;
page_number = ? non-negative integer ? ;
page_size   = ? non-negative integer ? ;

```

Similar to SQL we provide sorting capabilities using the *sort by* following by the name of attribute resources should be sorted by and optional order. The order can be either ascending - *asc*, or descending - *desc*. If not provided the default sorting order is ascending.

```

sorting = ' sort by ' ordering { ' , ' ordering } ;
ordering = attribute [ ' ' order ] ;
order    = ' asc ' | ' desc ' ;

```

| Type       | eq | neq | lt | gt | lte | gte | like | in range | in list |
|------------|----|-----|----|----|-----|-----|------|----------|---------|
| ObjectId   | ✓  | ✓   |    |    |     |     |      |          | ✓       |
| ObjectId[] | ✓  | ✓   |    |    |     |     |      |          | ✓       |
| String     | ✓  | ✓   | ✓  | ✓  | ✓   | ✓   | ✓    | ✓        | ✓       |
| String[]   | ✓  | ✓   |    |    |     |     |      |          | ✓       |
| Number     | ✓  | ✓   | ✓  | ✓  | ✓   | ✓   |      | ✓        | ✓       |
| Date       | ✓  | ✓   | ✓  | ✓  | ✓   | ✓   |      | ✓        | ✓       |
| DateTime   | ✓  | ✓   | ✓  | ✓  | ✓   | ✓   |      | ✓        | ✓       |
| Boolean    | ✓  | ✓   |    |    |     |     |      |          |         |
| State      | ✓  | ✓   |    |    |     |     |      |          |         |

Table 3.1: List of data types and supported operators

### 3.1 Resource attributes

Petriflow resources has different data attributes with different data types. In the final PFQL EBNF we need to define different types of predicates, paging, and sorting, for each resource type. In table 3.1 we provide an overview of all supported operators for all attribute types.

Process resources have following attributes (and types): *id* (ObjectId), *identifier* (String), *version* (Version), *title* (String), and *creationDate* (DateTime).

Case resources have following attributes: *id* (ObjectId), *processId* (ObjectId), *processIdentifier* (String), *title* (String), *creationDate* (DateTime), *authorId* (ObjectId), list of *places* with *marking* (Number), list of *tasks* with *state* (State) and *userId* (ObjectId), and list of process *data* attributes with *value* (ObjectId, ObjectId[], String, String[], Number, Date, DateTime, or Boolean) and *options* (String[]).

Task resources have following attributes: *id* (ObjectId), *transitionId* (String), *title* (String), *state* (State), *userId* (ObjectId), *caseId* (ObjectId), and *processId* (ObjectId).

User resources have following attributes: *id* (ObjectId), *name* (String), *surname* (String), and *email* (String).

Role resources have following attributes: *id* (ObjectId) and *title* (String).

### 3.2 PFQL query of motivation example

Using the PFQL and our implemented search service, we can demonstrate the action triggered by the **Get requests** button (see Figure 2.2). The implementation performs two main steps. First, it searches for instances of the **request\_query** process where the *name* and *surname* attributes match those of the current *Bulk approval* case. Second, it retrieves the corresponding *Approve request* tasks (with ID equals *t2*) for the identified cases. The resulting script is implemented as follows:

```
def children_request_cases = searchCases(
```

```
"processIdentifier == '${workspace}request_query'
& data.name == '${name.value}'
& data.surname == '${surname.value}'"
)?.collect{it.stringId}

change taskRef_0 value { searchTasks(
    "caseId in ${children_request_cases}
    & transitionId == 't2'"
)?.collect{it.stringId}}
```

# Chapter 4

## Implementation

Netgrif Platform was designed for modelling and execution of object-centric processes (see proceedings of international workshops on object-centric processes in [GGR25; DP24]) in a low-code language Petriflow [Juh+21]. It uses Netgrif Application Engine (NAE) to interpret and execute Petriflow processes. NAE backend is written in Java using the Spring Boot framework and stores the data of Petriflow processes, their instances, tasks of process instances, users, and roles in MongoDB as main database and Elasticsearch as a search engine.

MongoDB has a limited number of indexes and only certain metadata of Petriflow resources are indexed in the MongoDB. Data attributes of Petriflow object-centric processes marked as indexed are then indexed in Elasticsearch. This means that developers need to know which database to search in which cases.

The PFQL was implemented as part of Netgrif Platform to address these issues. As you see, syntax of the PFQL is inspired by Elasticsearch queries, which are easier to read compared with QueryDSL. Abstract search service was implemented using the *ANother Tool for Language Recognition* (ANTLR) to generate a parser for the PFQL EBNF.

Each query is parsed into Java objects and analysed to determine which database should be queried. Then the query is transformed into the specific query language of the database and corresponding service is used to evaluate that query. If all queried attributes are indexed in MongoDB then the query is transformed into a MongoDB query. Otherwise the query is transformed into an Elasticsearch query. Security context is also added according to the role-based access control. This ensures that users can access only those resources that they have permission to access.

The search service provides functionality for all PFQL statements search, count, and exists. It also provides a function to explain a query using the parse tree, which can help to understand which database is queried and why. Explanation of the previous query:

```
Searching single instance of resource CASE with Elasticsearch
page number: 0, size: 20
```

```
-- AND
-- processIdentifier == 'request'
-- data.name.value == 'John'
-- data.surname.value == 'Doe'
```

The service also provides similar explanation when an error occurs during parsing. For example the error output if instead of `'eq'` we put `'qe'` will be:

```

no viable alternative at input 'processIdentifier qe'
case: processIdentifier qe 'request' and data.name.value == 'John' ...
    ^^

mismatched input 'qe' expecting {<EOF>, PAGE, SORT_BY}
case: processIdentifier qe 'request' and data.name1.value == 'John' ...
    ^^

```

Providing explanation of a query and error output is essential for a wide adoption of the platform as searching instances and tasks is necessary even in small applications with a few processes.

## 4.1 Filters

Query language is not only usable in development of processes but also in creating filters by end users of the platform. These filters can be easily constructed in the graphical user interface (GUI) as seen in Figure 4.1 where the previous example query is being constructed. The filter building component guides users step-by-step and in each step provides options in the form of a drop-down menu. In the first step users select attribute for the predicate. In the second step users select an operator from the list of available operators for the given data type of the selected attribute. In the last step users provide a value they want to test against. Using the "+ AND" and "+ OR" buttons it is possible to chain multiple predicates. Each part of the predicate can be changed at any time by simply clicking on it.

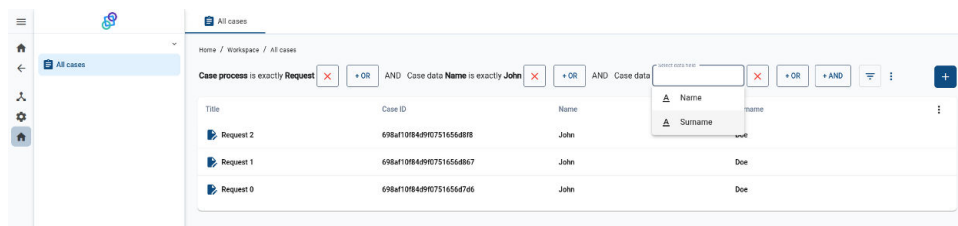


Figure 4.1: Example of a query language used in custom filter in the user interface of Netgrif Platform.

Filters and PFQL can be also used in processes in data attributes of types task reference (*taskRef*) and case reference (*caseRef*). For example in a task of bulk approval we can specify that only cases waiting for approval and belonging to the users department can be selected into a *caseRef* field.

# Chapter 5

## Future work

The PFQL can be extended in the future to cover also more complex querying process model and querying execution history of process instances.

Petriflow language defines events that creates a new instance of a process, assign and finish tasks, and change value of data attribute. Each occurrence of an event in Netgrif Platform is logged and stored in an event log collection. These event logs can be used as a source for business intelligence reports and dashboards by developers. Creating reports and dashboard is currently not a simple task and requires knowledge of Elasticsearch query language and aggregations. Implementing the PFQL for querying event logs could simplify this task and make it available for a wider group of users of the platform with less technical knowledge of Elasticsearch.

In other words, the current implementation of the Petriflow Query Language (PQL) provides a robust framework for state-based filtering, enabling users to query process instances based on their immediate data attributes and current marking. However, a significant potential for Petri net-based platforms lies in the ability to query not only the current state but the evolution of the process over time.

Our future research and development roadmap for the Petriflow framework includes the following key areas:

**Temporal Logic Extensions:** We plan to extend PFQL with operators from Linear Temporal Logic (LTL). This will allow for complex behavioral queries, such as "Find all cases where an invoice was approved without a prior purchase order" or "Identify processes where a specific task is eventually followed by a notification."

**Safety and Liveness Properties:** By incorporating LTL and potentially Computation Tree Logic (CTL), we aim to provide built-in support for verifying safety (something bad never happens) and liveness (something good eventually happens) properties directly within the application engine.

**Behavioral Compliance:** This extension will transform Petriflow from a process execution engine into a self-verifying system. It will enable on-the-fly compliance checks, ensuring that business cases adhere to formal specifications throughout their entire lifecycle.

Leveraging Petri Net Semantics: Since Petriflow is grounded in formal Petri net theory, these temporal queries will be evaluated against the reachability graph or using state-of-the-art model-checking techniques, ensuring mathematical precision in query results.

# Chapter 6

## Conclusion

In this report we present the motivation behind the Petriflow Query Language, definition of the main concepts on a simplified EBNF, and showcase its application in the Netgrif Platform on an example query. The Petriflow Query Language not only provides a unified query language but also takes away the burden of learning different data structures and implementation details of underlying databases (MongoDB) and search engines (Elasticsearch).

# Bibliography

- [Aal+09] Wil M. P. van der Aalst et al. “Soundness of Workflow Nets with Reset Arcs”. In: *Trans. Petri Nets Other Model. Concurr.* 3 (2009), pp. 50–70.
- [AJ15] Mohammed A. Alqarni and Ryszard Janicki. “On Interval Process Semantics of Petri Nets with Inhibitor Arcs”. In: *Petri Nets*. Vol. 9115. Lecture Notes in Computer Science. Springer, 2015, pp. 77–97.
- [Awa07] Ahmed Awad. “BPMN-Q: A language to query business processes”. In: Jan. 2007, pp. 115–128.
- [Bee+08] Catriel Beeri et al. “Querying business processes with BP-QL”. In: *Information Systems* 33.6 (2008), pp. 477–507. ISSN: 0306-4379. DOI: <https://doi.org/10.1016/j.is.2008.02.005>. URL: <https://www.sciencedirect.com/science/article/pii/S0306437908000124>.
- [DP24] Jochen De Weerd and Luise Pufahl, eds. *Business process management workshops*. en. Lecture Notes in Business Information Processing. Cham: Springer Nature Switzerland, 2024.
- [Des05] Jörg Desel. “Process Modeling Using Petri Nets”. In: *Process-Aware Information Systems*. Wiley, 2005, pp. 147–177.
- [DJ01] Jörg Desel and Gabriel Juhás. ““What Is a Petri Net?” Informal Answers for the Informed Reader”. In: *Unifying Petri Nets: Advances in Petri Nets*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–25. ISBN: 978-3-540-45541-7. DOI: 10.1007/3-540-45541-8\_1. URL: [https://doi.org/10.1007/3-540-45541-8\\_1](https://doi.org/10.1007/3-540-45541-8_1).
- [DR15] Jörg Desel and Wolfgang Reisig. “The concepts of Petri nets”. In: *Softw. Syst. Model.* 14.2 (2015), pp. 669–683.
- [GGR25] Katarzyna Gdowska, María Teresa Gómez-López, and Jana-Rebecca Rehse, eds. *Business process management workshops*. en. Lecture Notes in Business Information Processing. Cham: Springer Nature Switzerland, 2025.
- [Juh+21] G. Juhás et al. “Petriflow language and Netgrif Application Builder”. In: CEUR Workshop Proceedings 2973 (2021), pp. 171–175.

- [Val78] Rüdiger Valk. “Self-Modifying Nets, a Natural Extension of Petri Nets”. In: *ICALP*. Vol. 62. Lecture Notes in Computer Science. Springer, 1978, pp. 464–476.
- [Vog+22] Thomas Vogelgesang et al. “Celonis PQL: A Query Language for Process Mining”. In: *Process Querying Methods*. Ed. by Artem Polyvyanyy. Cham: Springer International Publishing, 2022, pp. 377–408. ISBN: 978-3-030-92875-9. DOI: 10.1007/978-3-030-92875-9\_13. URL: [https://doi.org/10.1007/978-3-030-92875-9\\_13](https://doi.org/10.1007/978-3-030-92875-9_13).
- [Vog02] Walter Vogler. “Partial order semantics and read arcs”. In: *Theor. Comput. Sci.* 286.1 (2002), pp. 33–63.