MoTion: A new declarative object matching approach in Pharo

Aless Hosry, Vincent Aranega, Nicolas Anquetil

MoTion: A new Declarative Object Matching Approach in Pharo

Aless Hosry[a], Vincent Aranega[b], Nicolas Anquetil[c]

[a] *Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[b] *MetaCell LLC, Cambridge, MA 02142, United States*
[c] *Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*

Abstract

Pattern matching is an expressive way of matching data and extracting pieces of information from it. The recent inclusion of pattern matching in the Java and Python languages highlights that such a facility is more and more adopted by developers for everyday development. Other main stream programming languages also offer pattern matching capabilities as part of the language (Rust, Scala, Haskell, and OCaml), with different degrees of expressivity in what can be matched. In the meantime, in graphs, pattern matching takes a slightly different turn; it enhances the expressivity of the patterns that can be defined. Smalltalk currently offers little pattern matching capability to find specific objects inside a large graph of objects using a declarative pattern. In Pharo, the closest library to classical pattern matching that exists is the `RBParseTreeSearcher`, which allows to express specialized patterns over a Pharo Abstract Syntax Tree to find some inner node. The question arises of what features a flexible pattern matching language should have. In this paper, we review the features found in different existing pattern matching languages, both in General Purpose Languages (like Java) and in declarative graph pattern matching languages. We then describe MoTion, a new pattern matching engine for Pharo smalltalk, combining all these features. We discuss some aspects of MoTion's implementation and illustrate its use with real case examples.

1. Introduction

The evolution of software engineering increasingly implies the need to search through data in memory and extract specific information in various domains, including *reverse engineering* [12], *analysis* [27] and *model transformation* [16]. There are various techniques to help express precisely and succinctly what is searched and to perform the search after that. Pattern matching is the process of checking whether a given pattern matches a given value or not [16]. This technique can be used for ensuring that data has the right "shape", but also to search for and extract information from data.

Many General-Purpose Languages (GPL) incorporate this as a core feature (Haskell, Java, OCaml, Python, Rust, Scala) [14], one of the latest being Java, introducing officially Pattern Matching for switch expressions and statements in Java 17. Graph pattern matching languages proved to be able to deal with a large variety and amount of data extracted from databases and represented in graphs. They are famous for being able to express patterns in a declarative way and for enabling path traversals of nodes and edges deeply inside any graph [27]. To simplify and make the pattern more declarative, some features are often implemented in graph pattern matching language, for example:

- *Non-linear patterns* allow developers to relate to found information many times in the same pattern, ensuring that a piece of data is actually in several places [10] (*e.g.,* the `name` of a component must be the same as the name of one of its sub-components);
- *List matching patterns* represent optional or repeated entities and allow developers to express a pattern over the shape of a list (*e.g.,* ensuring that the last element is a specific value) [5];
- *Deep recursive patterns* allow developers to consider that two pieces of data must be connected through one or many intermediate data, which is particularly handy when dealing with recursive data.

In Object-Oriented programming, objects are structured data that can be matched based on their type and/or field values. We refer to pattern matching in object-oriented languages as *object pattern matching*. Similarities exist conceptually between pattern matching in object-oriented languages and graphs, but they do not support the same set of features. More specifically, non-linear patterns, deep recursive patterns, and *List matching* patterns are currently not supported natively by pattern matching in object-oriented languages. In this paper we report on our efforts to implement a pattern matching extension to the Pharo programming language that supports all these features. We explore an approach to pattern definition influenced by pattern matching for graphs and for objects to pro-

Email addresses: `aless.hosry@inria.fr` (Aless Hosry), `vincent@metacell.us` (Vincent Aranega), `nicolas.anquetil@inria.fr` (Nicolas Anquetil)

pose to developers a way of searching and extracting information from objects. This is concretized in MoTion [1], a novel object matching language implemented in Pharo. MoTion allows the definition of declarative patterns, combining features from both worlds: pattern matching for graphs and for objects. Using MoTion, developers can define flexible patterns, disregarding the depth of the information being searched for and the structure's complexity of any object.

The structure of the paper is the following: In Section 2 we propose examples from the literature illustrating the need for pattern matching in modern software development. Section 3 presents the state of the art, the two main approaches for pattern matching (graph or object pattern matching) and a list of features that we found in the reviewed pattern matching languages. Then, in Section 4 we describe the syntax of MoTion, a new graph object pattern matching language implemented in Pharo. Some implementation details are discussed in Section 5 promoting the flexibility and extensibility of MoTion. We comment on some real use cases (Section 6) where MoTion was used by developers in software analysis tasks. From these use cases, we derived some lessons learned (Section 7) on the more useful features of MoTion or what improvement path lays before us. We close the paper in Section 8 with our conclusions and discussion of future work.

2. Motivation

When programming applications deal with large amount of data, developers often need to search for specific objects in the data. This happens when working in real world domains such as biology, transport and social networks. An example of such graph analysis is illustrated by the work of Thakkar et al., describes how to extract the age of the eldest person knowing a person named "Marko" in a big graph of people [27].

Software engineering daily activities also involve searching for source code elements. This is often done by representing the source code as a graph of objects (for example in an Abstract Syntax Tree) and searching for the "right object" inside this graph. For example in [13] Hosry et al. created a tool to detect dependencies between software elements, in a given system, that could be written in different programming languages. This involves searching for software elements having very specific properties (eg. carrying a specific annotation for a Java element, or having a specific attribute in XML). These software elements can be buried deeply in the graph of objects representing the whole system. In another example, Mohamed and Kamel [20] describe how to reverse engineer an application, looking for design pattern instances. For this, the authors suggested using static code analysis and following a set of heuristics, like identifying inheritance between classes and node selection inside methods based on their types. Finally, searching is also needed when transforming (or refactoring, or restructuring) the code into a new form. This can be done by looking for a specific software element and transforming it into a new one, or by modifying it in a new form [16].

Doing this kind of search in a GPL (General Programming Language) means going through sets of objects, selecting some of them, and then navigating to their children looking for specific properties (like attributes with a given value), ...

A better solution for this is to use a pattern matching language that will allow describing the sub-graph of objects one is looking for and letting it find all the matching occurrences in the whole graph of objects. The common pattern matching tools relieve the users from specifying how to traverse the whole graph, letting them concentrate on the description, in a specific notation, of the searched pattern.

Listing 1: Rascal example

```
1 public ColoredTree
2   makeGreen(ColoredTree t) {
3     return visit(t) {
4         case red(l, r) => green(l, r)
5     };
6 }
```

Listing 1 is an example extracted from the work of Klint et al. of a transformation rule expressed in Rascal, where method `makeGreen` defined in line 2 takes `ColoredTree t` as input parameter and replaces red nodes by green ones [16]. The `visit` in line 3 is used to traverse all tree nodes and apply the rule expressed in line 4 using `=>` operator, where the Left Hand Side (LHS) pattern `red(l,r)` is transformed into the Right Hand Side (RHS) pattern `green(l,r)`. This example illustrates the capabilities of pattern matching to represent the "shape" of data by describing a pattern: the LHS of line 4, and how information is extracted from this pattern: the `l` and `r` variables.

3. State of the art

When it comes to pattern matching, two main approaches exist: pattern matching with graph query languages, and pattern matching in GPLs. Both of these approaches propose to the developer a set of possibilities when it comes to declaring a pattern, and how the result is returned. In this section, we review the work related to declarative matching over data in graphs and in GPLs.

3.1. Pattern Matching for Graphs

The main goal of graph pattern matching is to find all matches between a collection of pattern variables and target graph nodes that meet a set of requirements [18].

[1] https://github.com/alesshosry/MoTion

According to Krause et al. [18]: A pattern is made up of a predicate set called a pattern graph, and an optional nested formula. The pattern nodes, or the nodes of the pattern graph, are represented by a set of pattern variables over which the predicates and the formula are defined. When a pattern is matched to a target graph, it means that (i) every pattern node is matched to a target node of the same type, (ii) every pattern edge is matched to a target edge of the same type, (iii) all predicates are satisfied, and (iv) the logical formula is satisfied. This mapping of pattern variables to target graph nodes is referred to as the target nodes of the match.

A vast range of data from many domains can be represented by graphs [19] where Resource Description Framework (RDF) [7] graph and Property Graph (PG) [2] are commonly used to represent this data [8].

An RDF graph is equivalent to a set of triples, of node-labeled edge-node, to represent the data [27, 8], and SPARQL [6] is the language that is capable of handling large-scale analytical operations over RDF graphs [27]. It uses a syntax with patterns expressed in triple form (subject-predicate-object) [18] using a combination of variables and specific resources or literals. The objects and subjects match nodes, and the predicates match edges.

Listing 2: SPARQL example

```
1 SELECT DISTINCT ?name
2 WHERE {
3   ?instance athlete ?athlete .
4   ?instance medal Gold .
5   ?athlete label ?name .
6 }
```

Listing 2 is a simple example of a SPARQL query to extract the names of every gold medal (excluding duplicates) from an RDF graph (see Figure 1). Line 1 is used to retrieve distinct (duplicates removed) names from the data returned by the query. The patterns are expressed in the `where` clause between lines 3 and 5. Names suffixed by "`?`" are variables that can match any node in the graph. Line 3 matches `?instance` and `?athlete` to 2 nodes related by an `athlete` edge. Line 4 requires that the `?instance` also be linked to a `Gold` node by the `medal` edge. Line 5 matches the `?name` variable to the node related to the `?athlete` node through a `label` edge. This `?name` is returned by the query (line 1).

Key points:

- SPARQL allows expressing the structure of the data looked for without worrying about how deep in the whole graph of data it is or how complex it is.
- In lines 3 and 5, `?athlete` has been reused to match the same value multiple times in the same pattern.
- Additionally, patterns are expressed in a declarative way allowing developers to add as many patterns as they need.

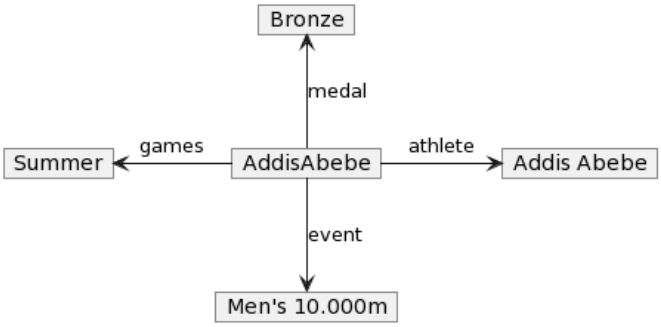

Figure 1: Example of a simple RDF graph

A PG models the data as a mixed multi-graph, where both nodes and edges can be labeled and attributed [8]. Various querying languages can be used for PGs like Gremlin APIs [23], and some declarative graph query languages like Cypher by Neo4j [11], GSQL by TigerGraph [9], PGQL by Oracle [28], or Graph Pattern Matching Language (GPML) which is able to run CRUD operations [8]. In addition, there are some experimental solutions in both industry and academia, like G-CORE [1].

Gremlin is considered a graph traversal language, allowing the exploration of complex relationships through various nodes in the graph using traversals like Recursive Traversal [23].

Listing 3: Gremlin recursive example

```
1 g.V().has("name","marko").
2 repeat(out()).times(5).
3 values("name")
```

A Gremlin traversal is a sequential movement through the steps, which are represented by nodes and edges in a data graph. It initiates from all nodes in the graph and traverses the path until the endpoint predefined by the developer is reached successfully among the graph. Listing 3 is a recursive traversal expressed in Gremlin that consists of selecting 5 persons named *marko*. In line 1 `g.V()` is the starting point of the traversal where `g` refers to the target graph of the match and `V()` denotes all nodes selection of `g`. Then, `.has("name", "marko")` filters the nodes to only those for which the property `name` is equal to *marko*. In line 2, `repeat(out())` will repeat the previous match to get all possible results. Then `.times(5)` limits the repetition to 5 occurrences. And finally for line 3, `values("name")` property of value name is retrieved after moving recursively five steps forward.

Key points:

- The main advantage of this matching technique is that it helps the developers specify a path to be traversed, with unlimited numbers of nodes and edges. It is adopted by many other languages for matching PG graphs like Cypher and PGQL.

- It is also possible to specify repeated searches to return all possible matches.
- Repeated searches risk falling into an endless cycle in the presence of cyclic relationships. Therefore, they can be limited to a maximum number of repetitions using `times()`.

3.2. Pattern Matching in GPLs

Pattern matching is one of the main features of functional programming languages [25] like Haskell[2]. With the evolution of object-oriented programming, pattern matching has increasingly found its way into this paradigm [17]. General Purpose Languages now have pattern matching capabilities like Java, which implemented pattern matching in the Amber[3] project; Python[4]; or Rust[5].

Such languages involve matching objects based on their types and/or field values, and many of them introduced the conditional match like "`switch` *subject* `case` *pattern*" in Java or Python.

Listing 4: Java Pattern Matching Example Java 21

```
1  record Point(int x, int y) { }
2  enum Color { RED, GREEN, BLUE; }
3  ...
4  String typ;
5  switch (obj) {
6    case null     -> typ="Null pointer";
7    case String s -> typ="A String";
8    case Color c  -> typ="A Color";
9    case Point p  -> typ="A Point";
10   case int[] ia -> typ="An Array of int";
11   default       -> typ="Something else";
12 }
```

Listing 4 exposes a pattern matching example in Java, where object matching is applied using the `switch case` statement. Lines 1 and 2, define the data structures that are used in the matching. Line 4 is a variable that will hold a description of the type of `obj` (line 5). The switch statement performs the matching by looking for the first `case` that will match `obj` in lines 6 to 11.

Key points:

- Object pattern matching helps define structural objects to be matched.
- Some languages allow to match an object not only on its structure, but also on the value an attribute should have.

We found several libraries that complement programming languages by introducing pattern matching capabilities. Tom [21] [4] and Rascal [16] integrate with Java, Kiama [26] integrates with Scala, and pyZtrategic [22] integrates with Python. Tom and Rascal cover features that were not covered natively by Java like *Path traversal*, *Recursive traversal*, *Object matching* and *List matching*. Kiama, Rascal and pyZtrategic enable transformation through the definition of strategies. These strategies employ pattern matching to identify the terms requiring transformation. Strategy execution in Kiama can proceed in different directions either top-down or bottom-up. This capability opens up the possibility of traversal in various directions, but it still requires additional investigation concerning its relevance in pattern matching.

3.3. Pattern Matching Language Features

We now consider what features have been proposed in different pattern matching tools/languages. This will be an inspiration for designing our own pattern matching language.

Klint et al. [16] state that a rich pattern language should provide string matching based on regular expressions, matching of abstract patterns, and matching of concrete syntax patterns. To reach a stage where developers are able to express any pattern compatible with the shape of the object they are looking for, a list of "features" must be provided by the language and applied using multiple operators or methods.

Previously in the literature, some authors have listed features supported by Rascal or Python [16, 17].

We first consider features found in graph pattern matching. *Graph matching* is famous for specifying patterns similarly to database SQL queries.

Declarative patterns help the developers define specific patterns that look like the results of matching, without caring about how these patterns will be matched. Using this paradigm leads to reduced development's time, increased maintainability, quick learning for pattern expression, and live preview changes without impacting the whole analysis process [15]. We oppose it to *Imperative paradigm* which consists of defining the computational steps to complete a task (the matching process).

Path traversal refers to visiting elements (i.e. nodes and edges) in a graph in some algorithmic fashion [24]. It helps the developers traverse nested structures while matching patterns.

Recursive traversal is needed to apply recursive search over deep structures, especially when developers ignore the depth of elements being searched for.

Repeated search is a feature related to the number of returned matches, where some languages can repeat the search to find all possible matches found, while others stop searching after finding the first match. A more flexible solution offers to specify the maximum

[2]Haskell https://www.haskell.org/tutorial/patterns.html
[3]Amber project https://openjdk.org/projects/amber/
[4]Python https://peps.python.org/pep-0000/
[5]Rust https://doc.rust-lang.org/book/ch18-00-patterns.html

number of matches that are expected, allowing to repeat the search without incurring the risk of infinite loops.

We now consider additional features of object matching. Some of these features would not make sense in graph patterns (such as *object matching*). We also add here some features that are inspired by pattern matching in functional languages such as non-linear patterns:

Object matching is dedicated to match objects based on their types and properties, which can be methods with return values or instance variables.

Literals (strings, numbers, integers) simply match themselves. They can be used to specify the value of an object's property.

Non-Linear pattern (sometimes called *unification*) allows the developers to use the same variable multiple times that should always match the same value in the pattern.

Wildcards represent a placeholder, an anonymous property that can be matched and is not used afterwards.

Nested pattern allows sub-patterns definition inside a pattern.

List matching supports the matching of a sequence of patterns taking into consideration their order. For example specifying that a pattern must be matched at the end of a list or in the middle,...

Logical matcher allows the possibility of combining multiple patterns in a boolean expression.

Negation may allow to express a simpler pattern when searching for bindings that do not conform to a particular criteria.

3.4. Some existing Object Pattern Matching Languages

We studied existing pattern matching languages to understand what features they offer. The goal of our matching language will be to offer all these features. We limited ourselves to Object Oriented GPLs. We considered the top OO languages used in 2022 according to github[6]: C#, C++, Java, Javascript, Python, Ruby, Typescript (three more languages are not OO: C, PHP, Shell). We added Rust and Scala that are well known for their pattern matching capabilities. And we added a library in Pharo (RBParseTreeSearcher) because this is the language we are working with.

Table 1 shows the features supported by different OO languages that have native pattern matching capabilities.

[6] https://octoverse.github.com/2022/top-programming-languages, consulted on may 2nd, 2024

Without surprise, *Object matching* is well supported. *Nested pattern* and *Wildcard* are also two features that are common. On the other hand, it shows that features like *Path traversal*, *Recursive traversal*, *Repeated search*, *List matching* and *Non-Linear pattern* are not universally supported by object matchers.

We are interested in creating an object matching language that could be used to match objects in models, taking into consideration that features like *Repeated search*, *Non-Linear pattern*, *List matching*, *Recursive traversal* and *Path traversal* are also important for such matching languages, in order to provide developers with the possibility to create patterns in a flexible way, allowing deep matching for deeply recursive models.

4. MoTion

MoTion is a new object pattern matching language in Pharo. A pattern matching language works on a finite set of objects that we will call a *model*. Examples of models are: the Pharo AST of a method, the DOM of an XML document, the objects loaded from a JSON file,... MoTion can deal with Pharo objects independently of the model containing the data. MoTion combines both features for graph pattern matching and object matching listed previously, and by doing so, it enables expressing patterns declaratively and applying matches to complex object structures.

In the next sections, we give a first overview of the pattern language with a concrete example. We also present the grammar of the pattern matching language and explain the semantics of each element of the language.

4.1. Simple pattern example

Before explaining in detail the syntax of MoTion, we give a simple example of a pattern used to detect all classes or interfaces that extend a given interface (named `'Remote'`). It works on Famix models. Famix [3] is a family of programming language meta-models that can represent programs. It is part of the Moose software analysis platform. FamixJava is the meta-model that allows representing Java programs, it defines classes such as `FamixJavaInterface` or `FamixJavaClass` that represent respectively Java interfaces and Java classes. These entities have properties like `name` that holds the name of an entity, or `superInheritance` that relate a class to its superclass.

Listing 5: External dependencies searching pattern

```
1 FamixJavaModel % {
2  #'allTypes>entities' <=>
3   FamixJavaInterface % {
4    #'superInheritances>superclass>name'
5     <=> 'Remote' .
6    #'isStub' <~=> true.
7  } as: 'foundInterface'.
8 }
```

Characteristics	C#	Java	Pharo (4)	Python	Ruby	Rust	Scala
Paradigm	D&I	D&I	I	D&I	D&I	D&I	D
Path traversal				x		x	x
Recursive traversal			x				
Repeated search			x				
Object matching	x	(1)	(5)	x	x	x	x
Wildcard	x	x	x	x	x	x	x
Nested pattern	x	x	x	x	x	x	x
List matching	x	(2)	x		x		
Literals	x	x	x	x	x	x	x
Logical matcher	x	x		x	x	x	x
Negation	x	(3)		x	x		
Non-Linear pattern	x	x	x				

Table 1: Pattern matching characteristics in Object Oriented programming languages.
(1) Matching Types, (2) Planned, (3) For types only, (4) RBParseTreeSearcher, (5) Only Pharo AST nodes.

The pattern starts by matching an instance of class `FamixJavaModel` (line 1) and looking in its `allTypes` property (line 2). This means that if it is not given a FamixJavaModel, it will not match anything.

`allTypes` returns a special object that is a wrapper around a collection of all types defined in the model (classes, interfaces,...). To get this collection, we use its `entities` property. Note that if the object returned by `allTypes` has no `entities` property, then the pattern just fails to match anything.

The result of line 2 is a collection of objects on which we apply the operator "`<=>`" with the sub-pattern in lines 3 to 7. The operator "`<=>`" is polymorphic and for a collection of objects, it will try to match its sub-pattern to any object in the collection.

This sub-pattern matches an object of class `FamixJavaInterface` (line 3), and looks in its `superInheritances` property (line 4), then in the `superclass` property of the returned object, then in the `name` property of this other returned object. Here again, the operator "`>`" is polymorphic and handles collections of objects (in the pattern `superInheritances>superclass`) differently than single objects (in the pattern `superclass> name`).

Line 5, if the name matches the string 'Remote'[7], the engine checks for the next sub-pattern (line 6) which states that the matched object of line 3 (an instance of `FamixJavaInterface`) should also have a `isStub` property that should not match the boolean `true` (this is different from saying it should match `false` because it could also be `nil`).

Finally in line 7, if a matching `FamixJavaInterface` is found, it is bound to the key `foundInterface` in the final result. This result will be returned, and the object matched can be retrieved from it.

4.2. MoTion Grammar

In this section we present the grammar of MoTion. It must be noted however that, due to the nature of Pharo, this grammar is somehow artificial because it is not implemented in a specific parser. MoTion is implemented as extension methods on existing classes (see also Section 5). In this grammar, the non-terminals ⟨*PharoClass*⟩, ⟨*PharodLiteral*⟩, ⟨*PharoSymbol*⟩, and ⟨*PharoString*⟩, refer to normal elements of Pharo (respectively, a class name, a literal, a symbol and a string).

⟨*Pattern*⟩ ::= ⟨*LiteralPattern*⟩
  | ⟨*PharoClass*⟩ ⟨*Percentage*⟩ '{' ⟨*Properties*⟩ '}'

⟨*LiteralPattern*⟩ ::= ⟨*PharoLiteral*⟩ '`asMatcher`'

⟨*Percentage*⟩ ::= '`%`'
  | '`%%`'

⟨*Properties*⟩ ::= (⟨*Property*⟩ ⟨*SpaceShip*⟩ ⟨*Value*⟩)*

⟨*Property*⟩ ::= ⟨*PropertyElement*⟩
  | ⟨*Traversal*⟩

⟨*SpaceShip*⟩ ::= '`<=>`'
  | '`<~=>`'

⟨*Value*⟩ ::= ⟨*PharoSymbol*⟩
  | ⟨*Pattern*⟩
  | ⟨*NonLinearPattern*⟩
  | ⟨*ListPattern*⟩

⟨*PropertyElement*⟩ ::= ⟨*PharoSymbol*⟩

⟨*Traversal*⟩ ::= ⟨*PathTraversal*⟩
  | ⟨*RecursiveTraversal*⟩

⟨*PathTraversal*⟩ ::=
  ⟨*PharoSymbol*⟩ ('`>`' ⟨*PathName*⟩)*

⟨*PathName*⟩ ::= ⟨*PharoString*⟩ | '`_`'

⟨*RecursiveTraversal*⟩ ::=
  ⟨*PharoSymbol*⟩ '`*`' ('`>`' ⟨*PathName*⟩ '`*`')*

[7] The `Remote` interface of Java RMI.

⟨*NonLinearPattern*⟩ ::= '`#@`' ⟨*PharoString*⟩

⟨*ListPattern*⟩ ::= '`{`' (⟨*ListItem*⟩ '`.`')* '`}`'

⟨*ListItem*⟩ ::= ⟨*NonLinearPattern*⟩
| ⟨*PharoLiteral*⟩
| '`_`'
| '`*_`'

4.3. Pattern operators

We now explain the semantics of the pattern matching language constructs while showing how they implement the features listed in Section 3.

- ⟨*LiteralPattern*⟩s are Pharo literals used as patterns. For example `'A sample text here' asMatcher` and `1 asMatcher`. Literal patterns match exactly their literal value. This is useful for specifying the value that a property of an object must have.

- The ⟨*SpaceShip*⟩ operator tries to match a ⟨*Property*⟩ (of an object) on the left with a ⟨*Value*⟩ on the right. Note: the tilde version is a negation, it specifies that the ⟨*Property*⟩ should not match the ⟨*Value*⟩; It is the only way to specify a negation in MoTion.

  As noted before, it is a polymorphic operator depending on the content in the ⟨*Property*⟩. If this is an object, the operator tries to match this object to the ⟨*Value*⟩; If it is a collection, the operator tries to match any element of the collection to the ⟨*Value*⟩.

- To define an *object pattern*, one specifies its type using the ⟨*PharoClass*⟩ followed by the ⟨*Percentage*⟩ operator like in: `ClassA % {}` . '%' matches direct instances of the class, whereas '%%' matches instances of the class or any of its subclasses.

- These two operators can express sub-patterns and the properties of the matched object inside the curly braces. Object properties are instance variable accessors.

  The curly braces act as a conjunction of sub-patterns specifying the values that properties should match. It can be seen as a *Logical matcher*.

  The following pattern matches an object of class `ClassA`, with a ⟨*Property*⟩: `property1`, having the ⟨*Value*⟩: `aValue1`, and `property2` having the ⟨*Value*⟩: `aValue2`.

  ```
  ClassA % {
    #'property1' <=> aValue1.
    #'property2' <=> aValue2.
  }
  ```

  The sub-patterns could also be more complex (see below, *Nested pattern*).

  This mechanism contributes to the seamless addition of various properties, in a *declarative* way.

- The ⟨*Percentage*⟩, combined with the ⟨*SpaceShip*⟩ operator, also allows to express *Nested pattern* where a first object is matched, then a second object in one of the properties of the first is matched. One may express a sub-pattern on this second object. For example, the following pattern matches an instance of `ClassA` with `aValue1` in its `property1`, and an instance of `ClassB` in its `property2`. This second object must have `aValue3` in its `property3`.

  ```
  ClassA % {
    #'property1' <=> aValue1.
    #'property2' <=> ClassB %% {
      #'property3' <=> aValue3.
    }
  }
  ```

- *Non-Linear pattern* is obtained using the "`@`" operator followed by a name (for example: `@x`). This allows to store a matched object in the "variable" to reuse it somewhere else in the pattern.

- *Wildcard* ("`_`") can be used to indicate a property whose name is not known, when one only cares for its value:

  ```
  ClassA % {
    #_ <=> aValue.
  }
  ```

  This pattern matches an instance of `ClassA` with an unnamed property matching the value `aValue`.

- The "`>`" operator implements *Path traversal* by allowing to "chain" multiple properties in a pattern. Such paths help reducing complex pattern's expression, by accessing a chain of objects and their properties:

  The following pattern first matches an instance of `ClassA`, then it takes the object in its `property1` and the value in `property2` of this second object. This value should match `aValue`.

  ```
  ClassA % {
    #'property1>property2' <=> aValue.
  }
  ```

  This notation allows expressing in a concise way a path in a graph of objects. The same result could be obtained with the pattern:

  ```
  ClassA % {
    #'property1' <=> Object %% {
      #'property2' <=> aValue
    }
  }
  ```

Note that the "`>`" operator is also polymorphic. Similarly to "`<=>`", if one of the objects in the path is a collection, the operator will look for an element of this collection that allows to continue the search, that is to say that has a property matching the remaining part of the pattern.

- MoTion allows to perform *Recursive traversal* through a "`*`" operator combined with the *Path traversal* operator "`>`". In a chain of objects, one may know the initial property and the final one, but not know how long the chain of objects is.

```
ClassA % {
  #'property1>repeatedProp*' <=> aValue.
}
```

  This pattern will match first an instance of `ClassA`, then the object in its property `property1` then it will match a chain of objects all having a property `repeatedProp` and one of them containing the value `aValue`. The match ends with this last object.

- The ⟨*RecursiveTraversal*⟩ operator may also be combined with a *wildcard* ("`_`").

```
ClassA % {
  #'property1>_*>propN' <=> aValue.
}
```

  This pattern will match first an instance of `ClassA`, then the object in its property `property1` then a chain of objects with unknown properties ending with an object having a property `propN` with value `aValue`.

- It is possible to match *List matching* using the ⟨*ListPattern*⟩ and declaring how the list should look like. Note that this is not the same operator as ⟨*Percentage*⟩ (see above). This operator allows to express that given elements in a list should match specific patterns.

```
{#'@x'. #'@x'}
```

  This pattern matches a list containing exactly two elements that are the same (use of a named variable).

- The repetition operator ("`*`") may also be used in a list to indicate an unspecified number of elements.

```
{#'@x'. #'*_'. #'@x'}
```

  This pattern, matches a list with first and last elements equal and of unspecified length (obviously at least 2).

  Note that `'*_'` is used in list matching whereas `'_-*'` is a repeated *wildcard* used in *Recursive traversal*.

- To express that one element is part of a collection, MoTion offers a shortcut. To check if the value 5 is part of a collection (contained in the property `someProperty` of an instance of `ClassA`) one can use the pattern:

```
ClassA % {
  #someProperty <=> {#'*_'. 5. #'*_'}
}
```

  But, thanks to the already presented polymorphism, of the ⟨*SpaceShip*⟩ operator, the same can be expressed with a shortcut:

```
ClassA % {
  #someProperty<=> 5
}
```

  This, however, could also match an instance of `ClassA` with a property `someProperty` containing exactly the value 5 (with no collection).

- Finally, there is another operator for *Logical matcher:* `orMatches:`. It allows to express a disjunction of two patterns (one or the other match). (Remember that ⟨*Percentage*⟩ implements a conjunction of patterns within the curly braces.)

```
ClassA % {
 #someProperty <=> (5 orMatches: 6)
}
```

  This pattern matches an instance of `ClassA` with a property `someProperty` matching the value 5 or the value 6.

4.4. Using MoTion

First, one gets a "matcher" by calling the `asMatcher` method. We showed an example of this at the beginning of Section 4.3: "`1 asMatcher`" creates a matcher that only matches the value "1".

Second, a matcher has a `match:` method that allows it to try to match the argument.

The result of `match:` is a `MatchingResult`. It includes a boolean property `isMatch` indicating whether the match was successful or not. It also has the property `matchingContexts` which is a collection of `MatchingContext` objects. Each of these contexts includes again a boolean field `isMatch` and a dictionary of its `bindings`.

The following creates a matcher that matches anything and binds it to the "foo" symbol (first line). The pattern is run on the string `'text'`. The last line will answer `true` as the match was successful.

```
pattern := #'@foo' asMatcher.
result := pattern match: 'text' .
result isMatch.
```

To get the binding of `foo` in this small example, one would do (`bindings` returns a dictionary and `at:` is the standard method to access an element of a dictionary):

```
result matchingContexts first
  bindings at: 'foo'.
```

This will return the string `'text'`.

Bindings can also be created with the `as:` method. It is used to bind the result of a pattern that will be kept in the result's bindings. For example, in Listing 5, it is used to store the result of the sub-pattern in lines 3 to 7 (the matched FamixJavaInterface).

Finally, to simplify getting the result of the bindings one is mostly interested in, there is a method `collectBindings:` that accepts a collection of (interesting) keys as a parameter and returns their values matched by a pattern. In case there is no match, the return is an empty collection.

```
pattern := #'@foo' asMatcher.
results := pattern
collectBindings: {#foo }
for: 'text' .
```

This puts in `results` a collection of dictionaries (here there is only one) with the binding for the `#foo` symbol. The result is a collection because there could be several matchings (for example with a disjunction operator). The collection holds dictionaries because we could ask for several bindings in the first parameter of the method.

5. Implementation Notes

MoTion uses the flexibility of Pharo syntax to implement the operators and enable the creation of additional operators or the specialization of existing operators.

For example the "% {}" operator is implemented as a method on `Class`[8] so that this expression is valid in Pharo:

```
ClassA % {
  ...
}
```

We saw in the previous section that some operators are polymorphic ("`<=>`" and "`>`"). This is implemented through a polymorphic `#match:withContext:` method (not further described here).

In the following subsections, we propose some examples of implemented extensions, but more details about this implementation can be found on the open source hub of the project. In addition, a chapter will be published in an upcoming book: New tools in Pharo[9].

In summary the implementation relies on:

- A class `Matcher` responsible for matching a pattern to a model with the method `match:` (see also Section 4.4). It has 19 subclasses performing some operators (like `%`) or literals as patterns,...
- Classes `MatcherResult` and `MatcherContext` that hold the result of a matching. An instance of `MatcherResult` is obtained as the returned value of the `match:` method (see above)
- Class `MotionPath` to implement the various path features: ⟨*PropertyElement*⟩ (ie. `#name`), *Wildcard* (ie. `#_`), *Path traversal* (ie. `>`),...

  It has six subclasses all implementing a method `resolveFrom:`.
- Six implementations of a method `asMatcher` added to pre-existing classes Array, Boolean, Class, Number, String, and Symbol. They convert a literal or Object to a pattern (ex: 'A sample text here' `asMatcher`).
- Methods `%` and `%%` implemented in Class to allow expressing *Object matching* (ie. ⟨*PharoClass*⟩ `%` {}).
- the ⟨*SpaceShip*⟩ methods (ie. `<=>` and `<~=>`) added to class `Object`

5.1. A simple extension

Listing 5 showed an example of a pattern on a Famix model. We actually implemented an extension of MoTion for Famix.

In Famix, the properties of entities can represent:

- "Famix *properties*" that contain "Famix primitive types" (Numbers, String or Boolean);
- *associations* that point to another Famix entity (a FamixJavaMethod *invokes* multiple other FamixJavaMethods);
- *composition* relationships (a FamixJavaClass *contains* multiple FamixJavaMethods).

Because the properties are meta-described, one can manipulate them programmatically. We therefore experimented with modifying the behavior of the path operator ("`>`") to navigate only *composition* relationships. We also added another operator to preserve the previous behavior of the path operator.

5.2. Changing the syntax

Another experiment could be to change the syntax of MoTion to make it easier to understand for novice users. For this, we propose to replace the operators with keyword messages: The "% {}" operator could be replaced by the message "`instanceWithProperties:`"; "`<=>`" operator could be replaced by the message "`objectMatches:`",

[8] Actually, the method is `%` and the curly braces are the argument of the method.

[9] https://github.com/SquareBracketAssociates/NewToolsInPharo, consulted on May 2nd, 2024; book in writing at this date.

and the `<~=>` operator could be replaced by the message "`objectDoesNotMatch:`".

We illustrate these changes on the pattern example of Listing 5. The result is shown in Listing 6 with the changes highlighted in bold. It's important to emphasize that both patterns are valid and interpreted by MoTion in exactly the same way. We just added "synonym" methods.

Listing 6: MoTion operator replaced by Pharo message

```
1 FamixJavaModel instanceWithProperties: {
2  #'allTypes>entities' objectMatches:
3   ((FamixJavaInterface instanceWithProperties:{
4    #'superInheritances>superclass>name'
5    objectMatches: 'Remote' .
6    #'isStub' objectDoesNotMatch: true.
7  }) as: 'foundInterface')
8 }
```

The downside of this approach is that we need to put parentheses around the sub-patterns. Here, the "`as:`" message (line 7) could "collide" with the new keyword messages (`objectMatches:` on line 2 and `instanceWithProperties:` on line 3) and be mistaken for a composed keyword message (`objectMatches:as:` or `instanceWithProperties:as:`). This is an issue that did not arise with the use of symbols because of the precedence of binary messages in Pharo. Note also that, instead of the inner parentheses, we could actually have created the `instanceWithProperties:as:` method that would first call `instanceWithProperties:` and then `as:` on its result.

5.3. Iguala

The last example is different as it consists of the re-implementation of MoTion in Python. Reimplementing MoTion in another OO language requires:

- Reimplementing the classes and methods listed at the beginning of Section 5;
- Reflection capability from the language to be able to compute the properties (attributes) of any object;
- Preferably, the ability to add methods to existing classes (call extension methods in Pharo).

With Python, this was possible, and the project is called Iguala[10]. This project re-implements entirely the object model of MoTion and it required 1 day work to be accomplished. Some operators had to be changed because Python does not extension methods. Thus the "% {}" operator is redefined as `match()[]` in Iguala (see Listing 7, lines 1 and 2). Its behavior remains the same, but the class name is passed as a parameter of `match()` and square brackets are used to contain the patterns or the properties. Syntactically, the square brackets represent "slices" in Python, each element inside them is a Python slice where the start of the slice is the path to match, and the end of the slice is the pattern associated to this path. Therefore, in this notation , `<=>` is replaced by colon `:` (lines 2, 3, and 4). MoTion's `<~=>` operator is expressed as `is_not` (line 4). Finally, the `as:` message of MoTion becomes `@` operator in Iguala (line 5).

Listing 7 shows the same Listing 5 example re-defined with Iguala.

Listing 7: Iguala pattern definition

```
1 match(FamixJavaModel)[
2   'allTypes>entites': match(FamixJavaInterface
        ) [
3     'superInheritances>superclass>name' : '
          Remote',
4     'isStub' : is_not(True)
5   ] @ 'foundInterface'
6 ]
```

6. Use cases

We used MoTion in some of our projects and presented it to other people to use in their projects. We report here some of these experiments. We will summarize the lessons learned from these experiments in Section 7.

6.1. External dependencies

MoTion was used in a project on external dependencies detection [13] that deals with polyglot software, developed using several programming languages at the same time. This is the case for example of GWT applications that use Java and XML, or RMI systems where two applications (client and server) must cooperate.

In order to be able to detect dependencies in these projects, we used MoTion to create more generic patterns that could be reused for different frameworks. For example, searching for an XML attribute was used for GWT applications, but could be reused in other cases.

Our running example (Listing 5) comes from this experiment. It was already presented and explained in Section 4.1.

We noticed in this work, the use of multiple features of MoTion together for many patterns:

- structured patterns for complex objects such as `FamixJavaModel % {...}` (line 1) and `FamixJavaInterface % {...}` (line 3);
- traversal paths expressed in this listing in lines 2 and 4, that allow matching chains of objects;
- negative search (line 6).

[10] Iguala stands for "*igual a*", "equals to" in Spanish. https://github.com/doubleBlind/iguala

6.2. Refactoring source code

Our next use case is a developer who used MoTion for a refactoring task over a Java application with a model that contains more than 1.5 million entities.

The problem was to detect all the invocations of method `get` on an object `config` (the receiver) with argument a specific key (`config.get(aKey)`). This needed to be done on a representation of the AST of the method to be able to modify the AST after.

The developer created the pattern in Listing 8.

Listing 8: Reverse engineering pattern

```
1 FASTJavaMethodEntity % {
2  #'children*' <=> FASTJavaMethodInvocation % {
3   #'receiver>name' <=> #'config'.
4   #name <=> #get.
5   #'arguments>primitiveValue' <=> aKey.
6  } as: #configInvocation
7 }
```

The pattern was applied to FAST-Java, a member of the Famix family specializing in modeling Java ASTs. It starts by matching a FASTJavaMethodEntity (ie. a method node) and looks at its children for a FASTJavaMethodInvocation (line 2). Because this invocation could be at any depth in the AST, he used the "`*`" operator (*Recursive traversal*). On the invocations matched, it looks for the receiver's name which should be "config" (line 3). The name of the invocation (method invoked) should be "get" (line 4). The argument of the invocation should be an object with the property "primitiveValue" matching `aKey` (line 5). Here, the key is a parameter that can change for different searches.

We noted in this work:

- The *Recursive traversal* which was necessary because the invocation is at different depths in the AST in different methods.
- The ease of use, the developer was able to work alone after a small presentation of MoTion syntax of only half an hour.
- The need for "named sub-patterns" that could be reused to compose complex patterns. Actually this feature exists but the developer did not think about it. The solution is simply to put a pattern in a variable that can be reused to build more complex patterns.

  As an example Listing 9 presents a rewritten version of the pattern represented in Listing 8 (even though there is no sub-pattern reuse there). Variables are highlighted to help read the pattern.

Listing 9: Reverse engineering pattern decomposed

```
1  childrenPath := #'children*'.
2  receiverNamePath := #'receiver>name'.
3  argsVal := #'arguments>primitiveValue'.
4
5  subPattern := FASTJavaMethodInvocation % {
6   receiverNamePath <=> #'config'.
7   #name <=> #get.
8   argsVal <=> aKey.
9  } as: #configInvocation.
10
11 FASTJavaMethodEntity % {
12  childrenPath <=> subPattern.
13 }
```

6.3. Software Quality Assessment

The Iguala project is currently used in `napari-hub-cli` [11], an open source tool developed for the Chan Zuckerberg Initiative responsible for the static analysis of plugins developed for Napari [12] an interactive viewer for multi-dimensional images. The tool checks and assesses the quality and conformity of dedicated plugins. The quality of a plugin is established by the presence or absence of some files, as well as the way those files are structured. The files that are analyzed by the tool are of different natures: yaml configuration files, Python code, markdown documents, ...

`Iguala` was used for this task. The example in listing 10 shows how a pattern over a Markdown document is written to check if the Markdown document has a title and to bind it. This pattern works on objects created by `mistletoe` [13] a Markdown parser. The pattern matches a `Document` (line 1) and looks in its direct children for a `Heading` instance (line 2) of level `1` (line 3), which we can interpret as being the title of the document. From this, the content located in `children>content` is extracted and bound to the `title` variable (line 4).

Listing 10: Iguala pattern over a Markdown document

```
1 pattern = match(Document)[
2   "children": match(Heading)[
3     "level": 1,
4     "children>content": "@title"
5   ]
6 ]
```

Without `iguala`, the same search in Python would be expressed as shown in listing 11, using a pure imperative approach (lines 1 to 8), or using pattern matching (lines 10 to 24).

We noted in this work:

- The pure imperative implementation is about the same size as Iguala version, but it is more complex

[11] https://github.com/chanzuckerberg/napari-hub-cli
[12] https://napari.org/
[13] https://github.com/miyuchina/mistletoe

Listing 11: Extracting the title from a Markdown document in Python

```
def extract_title1(document):
  for child in document.children:
    if (isinstance(child, Heading)
        and child.level == 1
        and child.children
        and len(child.children) > 0):
      return child[0].content
  return None

def extract_title2(document):
  def search_in_list(l):
    match l:
      case []:
        return None
      case [Heading(
              level=1,
              children=[
                RawText(content=content)
            ]),
            *_]:
        return content
      case [_, *tail]:
        return search_in_list(tail)
  return search_in_list(document.children)
```

to read and requires more checks in order to be sure that no operation on `None` or an empty list will happen.

- The version using the pure pattern matching mechanism of Python is much longer. The complexity occurs here in two major points: Iterating on the collection in a functional way using pattern matching; and the need to express information about the intermediate objects (the `RawText` instance).

  Obviously, this code could have been rewritten differently to mix for loops and a simple `match` for matching the `Heading` instance. This would have simplified the recursion over the list, but would not have removed the use of the intermediate pattern for the `RawText` instance.

6.4. Backend for other pattern matching

In another work [14], we compared MoTion to `RBParseTreeSearcher`. It is a pattern matching language to search over the Pharo AST (and possibly rewrite the AST with `RBParseTreeWriter`).

We compared MoTion to `RBParseTreeSearcher` class, by applying a search over the same Pharo AST, with both matching languages. Listing 12, shows the two patterns, used to check if the AST contains a selector `#ifTrue:ifFalse:`.

The patterns in `RBParseTreeSearcher` syntax (line 1) look similar to the original Pharo source code, except that some operators are used to help describe the pattern. Here, we are using the backtick operator (`` `@ ``) to refer to a list of nodes in the AST. The pattern `` `@receiv `` can match multiple nodes that behave as a receiver for the `#ifTrue:ifFalse:` message, and the blocks can contain multiple arguments inside, that are different from each other after naming them `args1` and `args2`.

In MoTion (line 3 to 5), we defined a pattern of type `RBMessageNode`, and used a traversal path to match the selector searched for.

Listing 12: Pharo AST matcher

```
1 `@receiv ifTrue: [ `@args1 ] ifFalse: [ `@args2 ].
2
3 RBMessageNode % {
4   #'selector>value' <=> #ifTrue:ifFalse:
5 }
```

We noted in this work:

- With RBParseTreeSearcher, specifying patterns to detect the receiver and the list of arguments inside the blocks is mandatory, while in MoTion, this specification could be skipped as the developer is only interested in knowing if the selector is invoked in this code or not.
- We found that RBParseTreeSearcher is faster. This should be expected as it is a matching language dedicated to only Pharo ASTs, while MoTion is a generic and can match any object at any depth, implying more computation and thus more time to execute.
- RBParseTreeSearcher lacks some capabilities, like the ability to express path traversal (see Table 1 for the features of this matching language).

7. Lessons learned

In this section, we summarized the take-outs of our experiments, both positive and negative.

7.1. Comparison with pre-existing

We compared the performances of MoTion and RBParseTreeSearcher patterns for Pharo AST search. MoTion was slower. We suppose this comes from the fact that RBParseTreeSearcher is fine tuned for matching Pharo AST nodes. For example it lacks some capabilities that MoTion has and makes the matching more computationally demanding.

We are considering whether it would be possible to compile MoTion patterns to make them more efficient. This is the subject of future work.

We conducted an experiment comparing pattern matching (MoTion) with traditional programming approaches using the reverse engineering example described in Listing 8. For the traditional programming, we implemented a new class with three methods totaling 30 lines of code.

These methods are contained within a single class. To use them, an instance of the class must be created. The results obtained are equivalent to those achieved with the MoTion pattern encapsulation using `as: #configInvocation`.

In summary:

- The user code is longer, one class, three methods, 30 lines instead of a 7 lines pattern;
- *Recursive traversal* (`#'children*'` in MoTion) required to implement a recursive method;
- Searching in the list of arguments (`#'arguments> primitiveValue' <=> aKey`) required to loop over the values returned by `arguments` to check their `primitiveValue`;
- The *Non-Linear pattern* (`as: #configInvocation` in MoTion) simplifies collecting the results that can be later retrieved with `#collectBindings: for:`. Traditional programming requires taking care of how results are returned by each method to collect and return them at the end;
- The class and methods created are very specific to the problem considered and another pattern would require reinventing a new solution with a possibly very different strategy.

7.2. Most used features

In the experiments reported, we found the following features to be the most used:

Object matching is the main feature used in all patterns. This is due to the fact that we are in an object oriented language and the things to match are objects.

MoTion, however, unlike Tom or Rascal, can work with any object structure and doesn't require its own definition of the classes to express patterns on them.

We worked with many different models (FamixJava, FASTJava, XML-DOM, Pharo AST, mistletoe model) without having to specify anything specific in MoTion other than the patterns themselves.

Repeated Search helped developers to collect all the matches of a pattern. MoTion does not stop after the first match is found, it stops when the leaf is reached. For now, this option seems to be enough for most common uses. However, we set a future target of adding a feature to limit the number of searches, which will be useful for some cases to prevent cyclic looping. A difficult issue will be to find a succinct syntax to express this feature.

Traversal Path is very useful to match nested objects by simply describing the path to reach them in a concise way. This makes the pattern more readable and understandable by the user.

It was used for example in Listing 5 (line 4) to navigate from an object, to its `superInheritances`, then the `superclass` of this object, and finally the `name` of the last object.

Resursive traversal is very helpful in searching for properties with unknown depth in a tree, or even for unspecified property with a concrete value. It was used for example in Listing 8 (line 2) to find an invocation that was at an unspecified depth in the AST.

There is a risk with this feature of entering an infinite loop if the graph is cyclic. In our example we used the `children` property which is a containment tree and assures us that the search will end.

For future implementations, we are planning to add a limit number for the recursive search like (`*numberOfSearches`) that will prevent it from running forever.

Wildcards not only helped developers express ignored properties, we also discovered another important usage of it. It was used by some developers to express multiple properties having common sub-properties, and the latter are the ones that interest the developers for search.

Non-linear pattern was very beneficial for developers dealing with cross-language applications, as it allowed them to express patterns in testing experiments to match the same value of a property in 2 different languages, like configuration keys defined in an XML file and referred to in a Java method.

Being able to express non-linear patterns was really useful in the context of matching YAML configuration files[14] (experiment not presented here), where those files can have very different structures and the same information needs to be matched in different parts of the data and at different depths.

On the opposite, list matching patterns were not used explicitly in our experiments. They appear in the "short cut form" in some patterns (eg. Listing 5, line 2) when a pattern matches a list and the "`<=>`" operators allow to look for one element of the list.

We used it only in one example (not presented here), in Iguala for the Napari project, to describe in one line the precedence of some values in a list.

7.3. Missing feature

Debugging patterns is a known difficulty in pattern matching.

MoTion can return a false match in cases where patterns were expressed incorrectly (the pattern does not actually match what the user wants). Some help is required

[14] Specifically, GitHub Actions files.

for the user to find these mistakes. We started to implement simple solutions, but more needs to be done.

Note, however, that the experiments were related to program analysis, and that may have biased the result based on the features most commonly used. To ensure and quantify well the degree of usefulness of each feature, a bigger study should be conducted, considering the use of MoTion or Iguala in different contexts.

8. Conclusion

In this paper, we introduce MoTion, a new generic object pattern matching language for Pharo Smalltalk. A pattern matching language specifically tailored to match Pharo ASTs is already included in Pharo. MoTion can, however, match ASTs and, more generally, any Pharo object, and it can be used on-the-fly, *i.e.,*: it's not required to redefine artifacts, like object signatures, to use it (opposed to Tom).

In order to create a new object pattern matching language that can offer developers some capabilities like searching among objects with deep depth, defining non-linear patterns, and applying list matching, we have extracted a couple of features known to be adopted by graph and object pattern matching and applied them to MoTion.

We gave an example of MoTion and then presented the syntax in detail. We explained the functionality of each operator and how a match can be applied using different Pharo messages implemented, like `#match:` and `#collectBindings:form:`.

We also presented the implementation of MoTion and how it can be extended if needed. We applied some small modifications to some operators to replace them with more human readable messages. We presented in parallel Iguala, which is the equivalent matching language of MoTion in Python, and gave an example in listing 7 equivalent to MoTion pattern presented in listing 5.

In order to prove its feasibility, we presented MoTion to a couple of developers familiar with Pharo, who work in the reverse engineering domain and software analysis for external dependencies extraction. Developers shared their positive experience and requested a couple of features, such as debugging, that we will introduce in our future work. We also compared MoTion syntax with RBParseTreeSearcher to check its feasibility for matching Pharo ASTs. We are planning for the future to change its backend to rely completely on MoTion, as we were able to find some cases where the match is not yielding the correct results at the end. Additionally, we showed how Iguala is being used as an open source tool to do a static analysis of the plugins developed.

Ultimately, we concluded by enumerating the lessons we had learned that had not been taken into account when our matching language was first implemented. For example, developers employed wildcards to indicate anonymous properties as well as the fact that multiple properties occasionally share the same sub-property, the value of which is the one we are interested in matching.

References

[1] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, et al. G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1421–1432, 2018.

[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[3] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, and M. Derras. Modular Moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, Dec. 2020.

[4] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: piggybacking rewriting on java. In *Proceedings of the 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van Der Ploeg, T. van der Storm, and J. Vinju. Modular language implementation in rascal–experience report. *Science of Computer Programming*, 114:7–19, 2015.

[6] W. W. W. Consortium et al. Sparql 1.1 overview. 2013.

[7] W. W. W. Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[8] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022.

[9] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Aggregation support for modern graph analytics in tigergraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 377–392, 2020.

[10] S. Egi, A. Kawata, M. Kori, and H. Ogawa. Embedding non-linear pattern matching with backtracking for non-free data types into haskell. *New Generation Computing*, 40(2):481–506, 2022.

[11] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[12] H. C. Gall, R. Klösch, and R. T. Mittermeir. Pattern-driven reverse engineering. In *SEKE*, pages 334–341, 1995.

[13] A. Hosry and N. Anquetil. External dependencies in software development. In *International Conference on the Quality of Information and Communications Technology*, pages 215–232. Springer, 2023.

[14] A. Hosry, V. Aranega, N. Anquetil, and S. Ducasse. Pattern matching in pharo. In *IWST*, 2023.

[15] G. B. Imbugwa, L. J. P. de Araújo, M. Khazeev, E. Enombe, H. Saliu, and M. Mazzara. A case study comparing static analysis tools for evaluating swiftui projects. In *Journal of Physics: Conference Series*, volume 2134, page 012022. IOP Publishing, 2021.

[16] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.

[17] T. Kohn, G. van Rossum, G. B. Bucher II, Talin, and I. Levkivskyi. Dynamic pattern matching with python. In *Proceedings*

of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, pages 85–98, 2020.

[18] C. Krause, D. Johannsen, R. Deeb, K.-U. Sattler, D. Knacker, and A. Niadzelka. An sql-based query language and engine for graph pattern matching. In *Graph Transformation: 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings 9*, pages 153–169. Springer, 2016.

[19] L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *Journal of the ACM (JACM)*, 63(2):1–53, 2016.

[20] K. A. Mohamed and A. Kamel. Reverse engineering state and strategy design patterns using static code analysis. *International Journal of Advanced Computer Science and Applications*, 9(1), 2018.

[21] M. Pierre-Etienne, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *International Conference on Compiler Construction*, pages 61–76. Springer, 2003.

[22] E. Rodrigues, J. N. Macedo, M. Viera, and J. Saraiva. pyztrategic: A zipper-based embedding of strategies and attribute grammars in python. In *ENASE*, pages 615–624, 2024.

[23] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10, 2015.

[24] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph data management: Techniques and applications*, pages 29–46. IGI global, 2012.

[25] S. Ryu, C. Park, and G. L. Steele Jr. Adding pattern matching to existing object-oriented languages. In *ACM SIGPLAN Foundations of Object-Oriented Languages Workshop*, volume 5. Citeseer, 2010.

[26] A. M. Sloane. Lightweight language processing in kiama. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–425. Springer, 2009.

[27] H. Thakkar, D. Punjani, S. Auer, and M.-E. Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin. In *Database and Expert Systems Applications: 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I 28*, pages 81–91. Springer, 2017.

[28] I. Șimonca, A. Corbea, and A. Belciu. Analytical capabilities of graphs in oracle multimodel database. In *Education, Research and Business Technologies: Proceedings of 20th International Conference on Informatics in Economy (IE 2021)*, pages 97–109. Springer, 2022.