IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 1 December 2023, accepted 5 January 2024, date of publication 17 January 2024, date of current version 29 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3355319

RESEARCH ARTICLE

Exploring a Self-Replication Algorithm to Flexibly Match Patterns

PAUL LEGER[1], (Member, IEEE), HIROAKI FUKUDA[2], NICOLÁS CARDOZO[3], AND DANIEL SAN MARTÍN[1]
[1]Escuela de Ingeniería, Universidad Católica del Norte, Coquimbo 1781421, Chile
[2]Shibaura Institute of Technology, Tokyo 135-8548, Japan
[3]Systems and Computing Engineering Department, Universidad de los Andes, Bogotá 111711, Colombia
Corresponding author: Paul Leger (pleger@ucn.cl)

ABSTRACT Pattern matching algorithms have been studied on numerous occasions, mainly focusing on performance because of the large amount of data used in a matching process. However, a strong focus on performance can entail particular issues like the lack of flexibility to match patterns. As a consequence, programming developers need to tweak matching algorithms in contortive ways or create new specialized ones altogether if their specific needs are not supported. Inspired by the self-replication behavior of cells in biology, we explore and evaluate the design and implementation of an algorithm to flexibly match patterns, named Matcher Cells. Through the composition of simple rules applied to cells, developers can adjust the matching semantics of this algorithm to different needs. We describe this algorithm using a pure functional language as a recipe for any Turing-complete programming language and then offer an object-oriented architecture for languages like Java. To show the flexibility of our proposal, we use a concrete implementation in TypeScript to describe two applications, from different domains, that use pattern matching in a stream of tokens. Additionally, we carry out performance and developer experience empirical evaluations with undergraduate students using Matcher Cells. Finally, we discuss the pros and cons of using a biological-based algorithm, exploiting the compositions of rules, to match patterns.

INDEX TERMS Pattern matching, self-replication algorithms, string matching, context-aware systems.

I. INTRODUCTION

Pattern matching algorithms [1] check the occurrences of a pattern in a *sequence of tokens*. Such patterns are usually expressed using abstractions (*e.g.,* automata [2]), or languages (*e.g.,* regular expressions [3]). Although these algorithms have undergone extensive historical study, they continue to be a focal point of attention in contemporary times. This interest is attributed to their wide-ranging applications across several domains, including but not limited to spam filters, digital libraries, natural language processing, word processors, web search engines, parsers, computational molecular biology, and screen scrapers [4], [5]. A common characteristic among these applications is the abundant availability of large datasets that require filtration, extraction, and processing to uncover valuable data for researchers and practitioners. Thus, pattern-matching techniques should demonstrate their efficiency by identifying one or more patterns within datasets in a relatively short timeframe [5]. They should also possess the necessary flexibility and user-friendliness to accommodate pattern matching without requiring developers to possess an in-depth understanding of pattern matching algorithms or the need to fine-tune existing algorithms to meet their specific requirements [6].

One specific context that exemplifies the needs for flexible and extensible pattern matching algorithms is web scraping, which involves the practice of retrieving content from websites to store in repositories like databases or CSV files [7], [8]. Within the sphere of web scraping, a diverse array of pattern matching techniques are employed, including regular expressions (Regex), HTML Document Object Model (DOM), and XPATH. Nonetheless, these

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

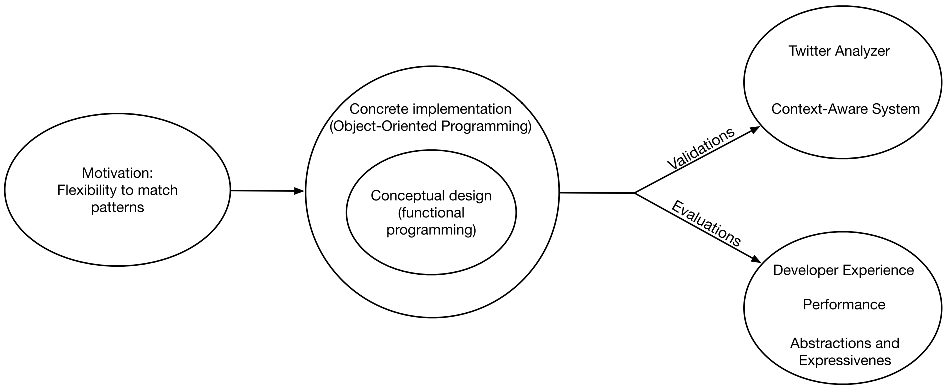

Motivation: Flexibility to match patterns
Concrete implementation (Object-Oriented Programming)
Conceptual design (functional programming)
Validations
Evaluations
Twitter Analyzer
Context-Aware System
Developer Experience
Performance
Abstractions and Expressivenes

FIGURE 1. **Methodology used in this study.**

methodologies have been perceived as not so flexibility, and complex in terms of implementation, and dependence on the structure of the data source [9]. For instance, websites often employ similar yet inconsistent templates for creating pages of the same category. Therefore, over time, the inner structure of a webpage can change without prior notification due to periodic updates in the layout, which may imply rewriting the matcher pattern algorithm to get the desired data. Consequently, these changes could impact the time, effort, and cost associated with web scraping and data extraction tasks [9]. A web scraping tool, which considers an approximate or flexible pattern matching algorithm, can help address the issue related to the website evolution.

In this paper, we present an algorithm founded on the principles of Biologically Inspired Computing (BIC) [10], which provides researchers with the basis to create flexible algorithms for pattern matching. This algorithm centers around a self-replication algorithm called Matcher Cells, which takes inspiration from the self-replicating behavior of biological cells to articulate a broad spectrum of matching semantics [11]. This work extends an initial proposal of Matcher Cells [12], where it was mainly employed for matching program execution traces in the context of aspect-oriented programming [13]. It is worth noting that, to the best of our knowledge, the application of BIC concepts in the realm of pattern matching remains relatively unexplored.

This paper extends our work in the following aspects:

1) A *mature* description of Matcher Cells using the Scheme programming language [14] to provide a generic implementation of our proposal that works on Turing-complete programming languages. We selected Scheme as a functional language that provides few and simple constructs to formally describe, as much as possible, a generic implementation.
2) An architecture to realize Matcher Cells in object-oriented languages like Java. We exemplify this architecture with a concrete implementation in TypeScript [15] for NodeJS (v16) [16], available at: https://github.com/pragmaticslaboratory/match-cell-base (rev. `e4c556d`).
3) Two case studies validating Matcher Cells. First, a pattern matching tool to analyze streaming services as social network sites (*e.g.,* Twitter [17]). Second, a context-aware system [18] that adapts the difficulty of addition exercises tasked to students, according to their performance (*i.e.,* context). Both case studies are available at: https://pragmaticslaboratory.github.io/matcher-cells-study-cases [19].
4) An experience evaluation of our proposal incorporating 23 undergraduate students from Universidad Católica del Norte (Chile). The usability of Matcher Cells is evaluated using the System Usability Scale (SUS) [20] approach.
5) A preliminary performance evaluation with a comparison to other two pattern matching algorithms, *brute-force* and KMP.
6) A discussion about the trade-off between programming abstractions and expressiveness of Matcher Cells.
7) A deep reference frame that contains proposals related to Matcher Cells, that is flexible-pattern matching algorithms.

Roadmap: Fig. 1 shows the methodology followed in this article. Section II presents two flexibility issues in pattern matching algorithms with their consequences, focused mainly on performance. After the motivation, Section III presents Matcher Cells, our self-replication algorithm to flexibly match patterns through a conceptual design. The design is followed by a concrete implementation of Matcher Cells in TypeScript, named MCJs. Section IV validates our implementation with two applications: a Twitter analyzer and a context identifier for context-aware systems. Additionally, we present our user experience evaluation from three perspectives: (1) developer experience, (2) performance, and (3) abstractions and expressiveness. Finally, the paper discusses different algorithms for pattern matching in perspective of our proposal in Section VI, leading to Section VII with the conclusion and avenues of future work.

II. FLEXIBILITY IN PATTERN MATCHING

With a large number of pattern matching algorithms available in the body of literature [21], pattern matching is currently

TABLE 1. Description of pattern matching algorithms.

Algorithm	Complexity		Usage scenarios
	Preprocessing	Matching	
Naïve (Brute-force)	-	$O((n-m+1)m)$	Small patterns and search spaces
Rabin-Karp [24]	$\Theta(m)$	$O((n-m+1)m)$	Multiple matches over large search spaces, image matching
Knuth-Morris-Pratt [1]	$\Theta(m)$	$\Theta(n)$	DNA sequence analysis, data compression
Boyer-Moore [25]	$\Theta(m)$	$O(mn)$	Text editors, command substitution, intrusion detection
BNDM [26]	$O(m(1+\frac{\|\Sigma\|}{\omega}))$	$O(\frac{nm^2}{\omega})$	DNA Sequence matching
BOM [27]	-	$O(mn)$	DNA Sequence matching
Suffix tree [28]	$O(n)$	$O(m)$	String search in bioinformatics, data compression, and data clustering
Suffix array [29]	$O(n)$	$O(m\log(n)+km)$	Data compression, find all matches of a pattern, find longest repeated words
Suffix automaton [30]	$O(M^3\|\Sigma\|)$	$O(m^2)$	String search, counting different strings
Aho-Corasick [31]	$O(n)$	$O(n+l+z)$	String search, find shortest strings
Bit-parallelism [32]	-	$O(\frac{nm}{\omega})$	Multiple matches, approximate matches

used in several fields such as string matching [1], execution trace matching in aspect-oriented programming [22], and intrusion detection [23]. However, as most of these algorithms mainly focus on their performance or algorithmic complexity, some issues can appear when more flexibility is needed. We now present existing pattern matching algorithms, followed by the issues detected in existing approaches.

A. MATCHING ALGORITHMS AND RESTRICTIONS

There are eleven main algorithms describing the families of pattern matching algorithms, as described in TABLE 1. For the evaluation of the algorithms, we assume a data set of n tokens and a pattern of m tokens. Naïve or *brute-force* algorithms enumerate all possible matchings of a pattern by checking the satisfiability of each potential matching of the pattern in the complete dataset. This algorithm implies comparing the pattern with every data point, leading to a worst-case performance (*i.e.,* time complexity) of $O(n*m)$. Brute-force algorithms have proven useful when matching over small datasets, due to their simplicity. However, the algorithm looses efficiency whenever the matching string has too many prefixes to match the pattern (*e.g.,* pattern $p =$ "*ddde*" to match string $s =$ "*dddddddddddde*").

The Rabin-Karp algorithm [24] offers an optimization over the naïve approach, and a general framework for other string matching algorithms by preprocessing the pattern string. The algorithm uses modulo equivalences and rolling hashes to process the string, taking the module of the pattern elements, and the search string taking windows of size m, with a time complexity of $\Theta(m)$. Given this, the complexity of the matching algorithm reduces the complexity to $\Theta(n-m+1)$. Given the use of hashed data, the Rabin-Karp algorithm is applicable for situations where there might be many matches in the string, processing them faster. Additionally, the algorithm is used to match bitmap objects (*e.g.,* images) easier.

The Knuth-Morris-Pratt (KMP) algorithm [1] uses sequences of pre-processed tokens. The matching time complexity of KMP is $O(n)$ in its best case time complexity. At first glance, KMP shows a much better performance than brute-force algorithms, however, KMP's performance can degrade to $O(n + m)$ if a sequence of tokens does not allow KMP to *appropriately* reuse information of previous partial matchings of the patterns. For example, consider a pattern $p =$ "*aaa*" and a sequence $s =$ "*aabaab*"; here, the matching fails every time the pattern is about to match, *i.e.,* when a "*b*" is found. KMP is useful for matching large data sets, like DNA sequence analysis or image processing (*i.e.,* its parallel version) with a better performance than naïve or Rabin-Karp.

The Boyer-Moore (BM) algorithm [25] flips the algorithm to match from the tail-end of the pattern. The algorithm uses a bad match heuristic to move forward in the search of the pattern, skipping over the characters before the bad match, until the first match in the pattern with the bad character. The time complexity of the algorithm is $O(n * m)$ in the worst case, given its dependence on the token sequence. The BM algorithm is used for text editors, command substitution, or intrusion detection.

BNDM (Backward Non-deterministic DAWG Matching) [26] is a bit-parallel algorithm based on suffix automata, extending BDM for faster execution. The algorithm uses a table to store *bitmasks*, a sequence of bits to keep/clear bits from another sequence, for each token to search. The algorithm shifts the search window according to the word size (ω), detecting matches whenever the last word bit is 1. The time complexity of the algorithm is $O(\frac{n*m^2}{\omega})$ for general patterns, but can be $O(n*m)$ (as in BDM) if the pattern used is smaller than the word size. BNDM is applicable in general matching scenarios but most efficient for searching patterns smaller than the memory word size.

The BOM (Backward Oracle Matching) algorithm [27] expands on the Boyer-Moore suffix matching by matching prefixes (*i.e.,* using a suffix oracle on the reverse pattern). As a consequence, BOM optimizes search window shifts, obtaining a better performance. On average, the performance of BOM is $O(n * \log_{|\Sigma|}(m)/m)$, but degrades to $O(m * n)$ in the worst case. BOM is used for DNA sequence matching and general string matching.

So far, the described algorithms have gained linear speedups proportional to the size of the data, *i.e.,* $O(n)$. In order to improve such execution time, it is possible to preprocess the data. Suffix data structures are used for such a purpose, keeping track of suffixes from the data set. Suffix trees [28] use common sequences in edges going down the tree. Tree leaves then contain the index in which the suffix, going down to that leave, starts in the data set. The processing of the suffix tree takes $O(n)$ time, as it requires going through the complete data set. However, once built, it is possible to directly match any possible pattern in $O(m)$ time. Given this property, suffix trees are of special interest for matching problems, as for example the case of DNA sequence search. Similar to Suffix trees, Suffix arrays [29] propose an alternative of matching algorithms that compromise search time performance to increase flexibility of matching patterns. Finding all k matches on a dataset takes $O(m * \log(n) + k * m)$ time, with an additional best-time of $O(n)$ to build the array. Suffix arrays are built by extracting all possible suffixes of the dataset, keeping their start index, and then sorting said array. Suffix automata [30], [33], similar to the previous two cases, are used to pre-process the dataset to search patterns. The difference between the automaton and the array and tree structures lies in the memory used for its construction, suffix automata presenting an optimization on the space used.

The Aho-Corasick algorithm [31], [34] builds a finite state machine with additional links between internal nodes to speedup transitions between failed matches. In particular, Aho-Corasick is used to match multiple patterns in the data set, with a time complexity of $O(n + l + z)$ with l as an upper bound on patterns' size, and z the total number of appearances of the patterns, and a preprocessing time of $O(n)$. This algorithm is used mainly for string search, as for example finding the smallest string of a given length, that contains k strings.

The bit-vector parallel algorithm for string matching [32]. This algorithms is built from the ideas of the dynamic programming algorithm for string matching, taking advantage of a bit-mask representation of the matching dynamic programming matrix. The matrix captures the bit representation of the difference between that data and the pattern, being able to manipulate the matrix in parallel using bit-wise operations. This approach constitutes a significant matching speedup with a time complexity of $O(\frac{n*m}{w})$, where w is the word size. This algorithms is of special interest in bioinformatics and DNA sequencing, as it is able to detect pattern matches with a bounded error between the pattern and its match string in the data set, for example a maximum error of 3 molecules.

Taking into account existing matching algorithms, we note that when considering a large amount of data as the sequence of tokens, some pattern matching algorithms like BM or brute-force, in their worst case, are not usable in practice (*cf.,* Section V-B). From this analysis, we can conclude that it is necessary to know the features of many pattern matching algorithms as a specific strategy may be used to boost performance, depending on the sequence of tokens and used patterns.

B. UNKNOWN SEQUENCES OF TOKENS AND PATTERNS

To exemplify the shortcomings of existing matching algorithms, with respect to their flexibility, consider a scenario where a web application has a filtering policy to prevent malicious requests that can affect its availability, compromise security, or consume excessive resources (*e.g.,* application firewalls for Amazon Web Services [35]). Here, the use of algorithms like KMP or BM may not work appropriately because the token sequence and the pattern to search may be unknown beforehand. More precisely, when it is necessary to filter for malicious requests, we may not know exactly which patterns, that represent malicious requests, should be used. For example, we may need to extend the filtering policy with new kinds of patterns if new variants of security attacks are observed. In this example, if we are using exact pattern matching (*i.e.,* a key-value table), changing the pattern model to use regular expressions may be necessary; that is, a completely different implementation is needed (*e.g.,* using a deterministic finite automaton).

In application domains such as stream processing over the Internet, patterns and their conditions to be matched (*e.g.,* a period of time) can vary on the fly; meaning that the semantics of pattern matching algorithms should be flexibly adaptable without tweaking or changing these algorithms.

III. MATCHER CELLS

Through small entities with simple rules, self-replication algorithms [11] allow developers to flexibly express the semantics of a process. This is because each rule defines a portion of the semantics, and their composition defines the full semantics. The composition process allows developers to easily adjust or create new semantics (on the fly), bringing a flexible expressiveness to define matching semantics.

This section defines a self-replication algorithm, named Matcher Cells, to flexibly match patterns, brining a variety of matching semantics that allows developers to express different the pattern matching algorithm semantics (Section 1). This section is organized as follows. We first start introducing self-replication behaviors in cells to describe how to use these behaviors to match patterns. The section then describes how to express a wide range of pattern matching semantics through compositions of cell behavior rules. Using the language Scheme [36], in Section III-D, we describe a generic recipe to implement our proposal in different paradigms such as programming languages like JavaScript. We finally present a concrete implementation of Matcher Cells in TypeScript, a typed version of JavaScript.

A. SELF-REPLICATION ALGORITHMS

Self-replication algorithms are inspired by cellular behavior [11]. Concretely, these algorithms emulate the reactions of a set of biological *cells* to a sequence of *reagents* in a *solution*. Fig. 2 shows the different possible reactions of a cell to a reagent, which can be:

1) the creation of an identical copy of the cell, or with a small variation to persist in the solution,
2) nothing,
3) death, or
4) a combination of the previous ones.

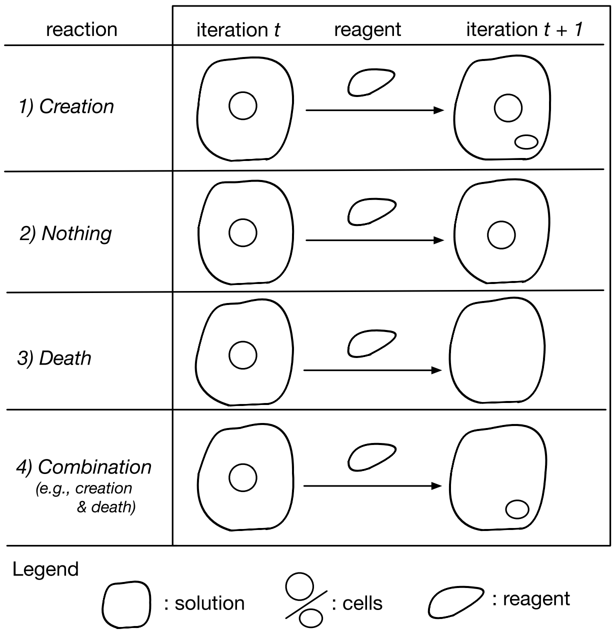


FIGURE 2. **Different reactions of a cell to a reagent.**

An algorithm that follows self-replicating behavior is defined by a pair (*Seeds*, *R*), where *Seeds* is the first set of cells into a solution, and *R* is the set of rules that govern the evolution of the solution using combinations of reactions. Additionally, if we consider that a solution is an *autopoietic system* [37], this solution can add or remove cells to maintain itself after all the current cells react to a reagent.

B. MATCHING PATTERNS

To match patterns, Matcher Cells' algorithm borrows concepts from self-replication algorithms, giving flexibility to match patterns. For this proposal, we consider reagents as *tokens* that must be matched, and cells to contain patterns of tokens and metainformation. We use the notation c(P) to represent a cell c that must match a pattern P, and c($) for a *matched cell*, which is the match of a pattern. When a cell creates a new cell, the new cell can gather metainformation like the link to the parent or the time when the cell was created. Fig. 3 illustrates the reaction to reagent a of two solutions:

Solution 1 - c(a): a cell that must match the pattern a (*i.e.,* only one token), creating a cell c($) when a is matched.

Solution 2 - c(a→b): a cell that must match a pattern that is composed of the sequence a→b (*i.e.,* a and then b). When this cell matches a token a, it will create a new cell, c(b), which must match the token b.

In both solutions shown in Fig. 3, the cell reaction is *creation*, and the links between them are stored in the metainformation of the created cell. Additionatly, Solution 2 shows the seed cell, c(a→b), gathers its creation time information, which is passed to the new cell when it is created.

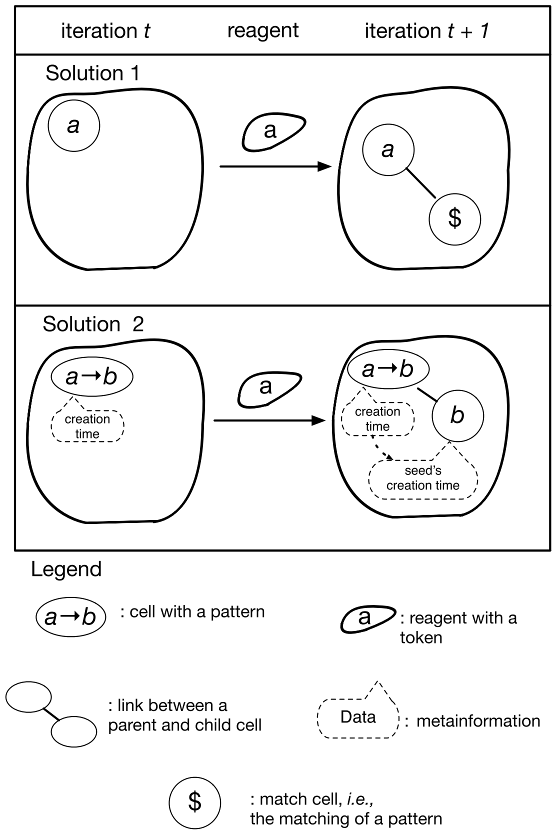


FIGURE 3. **Reactions of two solutions to a reagent** a**. Solution 1 contains a cell that must match** a**. The cell in solution 2 must match** a→b**.**

Fig. 4 shows three different evolutions of the same solution with a cell c(a→b) and a sequence of tokens a→a→b→b. The first evolution ends with only one match (*i.e.,* c($)) because a cell dies when this one creates new one(s); for this reason, c(a→b) dies when it matches a token b, and c(b) when it matches b. The second evolution ends with four matches because no cell dies when there is a match. In the third evolution, the solution evolves with two matches because the seed (*i.e.,* c(a→b)) never dies, but other cells die when they match a token. *Which is the correct evolution?* The answer will depend on the semantics required by programmers, opening up the flexibility for programmers to choose the most convenient option for their case.

C. MATCHING PATTERNS WITH FLEXIBLE SEMANTICS

As Fig. 4 shows, matching semantics strongly affects how a pattern is matched as well as how many times each pattern will be matched with a specific sequence of tokens.

In our self-replication algorithm, Matcher Cells, different matching semantics can be expressed through compositions of simple *rules*, which can be *per-cell* or *per-solution*:

1) **per-cell rules**: They are applied to each cell in a current manner using the same token.
2) **per-solution rules**: They are applied after all per-cell rules. Aiming to emulate the behavior of an autopoietic system, per-solution rules point to keep a solution useful to continue the developer-defined matching process.

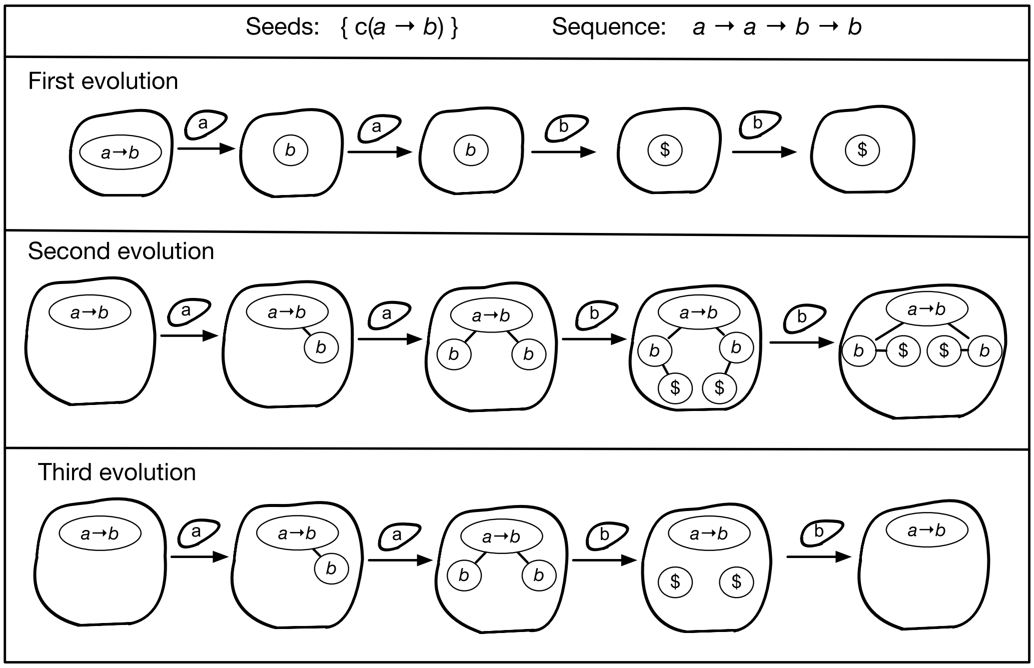


FIGURE 4. Three different evolution scenarios of a solution to the same sequence of tokens and pattern. The first evolution ends with a match, the second one ends with four matches, and the last one ends up with two matches.

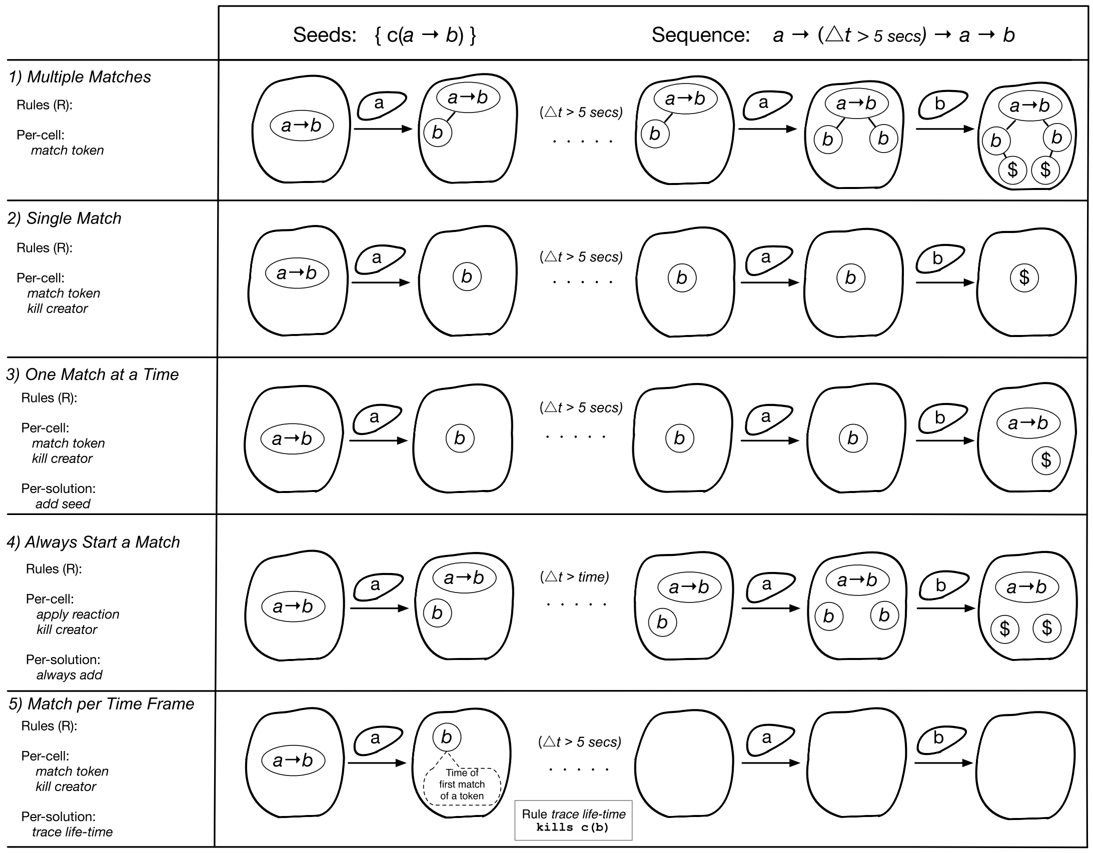


FIGURE 5. Using the same pattern and sequence of tokens, five different results because of matching semantics.

An example is the per-cell rule named match token, which executes the reaction of each cell to a token as shown in Fig. 2. Using this and other rules, Fig. 5 illustrates five different potential matching semantics of our algorithm:

1) MULTIPLE MATCHES

Using only the match token rule, Matcher Cells provides *multiple matching* semantics. For example, Fig. 5. shows that c(a→b) generates two c($)s when it meets the sequence of a→a→b.

2) SINGLE MATCH

This semantics is provided by the composition of match token and kill creator rules. This new rule kills the parent cell when a new cell is created. For example, c(a→b) creates only one c($) because c(a→b) is killed by the rule when c(b) is created as it is shown in Fig. 5.2.

3) ONE MATCH AT A TIME

Programmers often need to execute an action every time that the same pattern occurs, *e.g.,* when a security flaw occurs [38]. However, the composition of match token and kill creator will kill the seed cell such as c(a→b), resulting in a single match. Using the composition of match token and kill creator together with add seed, a per-solution rule that works after previous rules is applied; then it is possible to match the same pattern every time it occurs (Fig. 5.3). The add seed rule creates a copy of the seed cell whenever there is no cell in the solution, allowing the matcher to start a new matching process (*i.e.,* with a new cell and solution).

4) ALWAYS START A MATCH

The same pattern might start simultaneously, making simultaneous processes of matching work at the same time. For example, this semantics can be useful to capture simultaneous multi-intruders [39]. However, the previous compositions of rules does not allow Matcher Cells to start a new match if there is a matching process already executing. If we replace the add seed rule with always add, the algorithm will always be able to start the process of a new matching (Fig. 5.4). Note this semantics does not keep a link between the seed and its child cell because the kill creator rule kills the seed, and the always add rule inserts a new seed. Although this matching semantics and *multiple matches* lead to the same number of matches, both semantics are different. This is because the *multiple matches* semantics allows the matcher to continue a match from any part of a pattern already matched, instead the *always start a match* semantics can only start a match from the beginning of a pattern. To illustrate this difference, consider that the sequence of tokens has b as suffix, *i.e.,* a→a→b→**b**. With this extended sequence, *multiple matches* would have two new matches, while *always start a match* semantics would not have any new match when the last b token is received by the solution.

5) MATCH PER TIME FRAME

Suppose the scenario presented in Section Unknown Sequences of Tokens and Patterns, where malicious requests occur in a short period of time (*e.g.,* seconds). Here, patterns should be matched before Δ*time* elapses. Using the trace time-life rule, all cells are killed when the time period elapsed from the first token match is greater than Δ*time*. Fig. 5.5 shows an example of such situations.

D. EXPRESSING CELLS AND RULES

Cells and rules can be expressed using different programming paradigms' abstractions (*e.g.,* objects). To illustrate our proposal with one of the most straightforward programming abstractions, we use functional programming, as realized in Scheme [36]. As such, cells and rules are entirely expressed using only plain functions. The benefit of using a pure functional abstraction is that this description can be used as a recipe for any Turing-complete programming language.

As token and pattern definitions strongly depend on the application domain (*e.g.,* string matching), they are not considered as part of Matcher Cells' core. In the validation section, we discuss two applications, where concrete implementations for tokens and patterns are presented.

1) CELLS

A cell is a function composed of a pattern and its metainformation, which may create other cells when it reacts to a token (Fig. 2). In our proposal, the signature and implementation of a cell are:

```
;;Signature
;; Cell: Pattern x MetaInf -> (Token) -> MetaInf U List<Cell>
(define (Cell pattern meta-inf)
  (lambda (token null)
    (if (null? token) meta-inf
        (let ([result (react token pattern)])
          (if (pattern-matched? result)
              (return-list-with-new-cells result))
              (return-empty-list)))))
```

A Cell function returns its metainformation whenever it is called without a token, otherwise it returns the reaction to the token. The result of a cell reaction is a (possibly empty) list of matching cells.

2) RULES

We define per-cell and per-solution rules, which correspond to the functions applied to a list of cells (Section III-C). Application of a rule may remove or add other cells from the cells list. A per-cell rule is applied to each cell into a solution, which consumes a token of its sequence to match. A per-solution rule is applied to resultant cells after applying all per-cell rules. The most basic example of a per-cell rule is identity, which given a cell, it returns it untouched. An example of a per-solution rule can be remove-match-cells, which removes all match cells that are into a solution. Both rule examples are presented in the following Snippets with their corresponding signatures.

```
;;Signature
;; Per-Cell Rule: Token x List<Cell> -> List<Cell>
(define (identity token cells)
  cells)

;;Signature
;; Per-Solution Rule: List<Cell> x Pattern -> List<Cell>
(define (remove-match-cells cells pattern)
  (remove-match-cells cells))
```

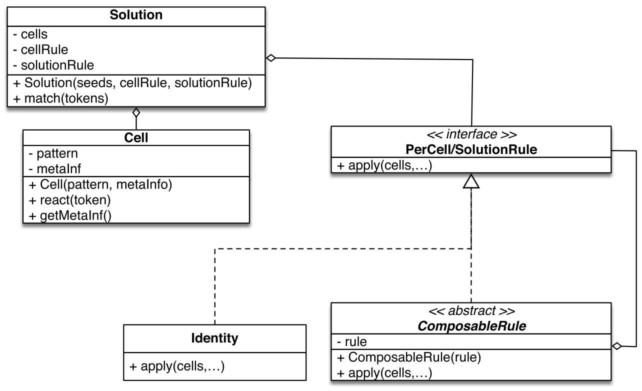


FIGURE 6. **Main components of Matcher Cells' architecture.**

Additionally, we introduce *composable* rules. A composable rule is a function that takes a rule (say rule 1) as parameter and returns a new rule (rule 2), implying that a rule is applied first and then its composition rule. We posit kill creator as a composable rule:

```
;;Signature
;; Composable—Rule: Rule —> Rule
(define (kill—creator rule)
  (lambda (cells token)
    (let ([new—cells (rule cells token)])
      (remove* (map get—creator new—cells) new—cells))))
```

Using composable rules, we can create different semantics like the ones shown Fig. 5. In the code snippet below, we present the multiple match and single-match semantics using composable rules, both starting with the identity rule.

```
(define multiple—match (match—token identity))
(define single—match (kill—creator (match—token identity)))
```

E. AN OBJECT-ORIENTED ARCHITECTURE

This section now shows an object-oriented architecture for Matcher Cells that can be used in languages like Java, JavaScript, and TypeScript. Fig. 6 shows the core components defined in our proposal. A solution carries out the matching process and is created with seeds, and the composition of per-cell and per-solution rules. Both types of rules evolve the cells in the solution. In addition, a developer can use of the *Composite* design pattern [40] to create composable rules, where the identity rule is the *leaf* component of this design pattern. A cell contains the pattern that must match, and metainformation that is cloned with potential mutations when passed to children cells. With the react method, a cell reacts to a token, potentially returning the creation of new cells.

We provide a concrete implementation of our object-oriented architecture, named MCJs, using TypeScript, a typed version for JavaScript (one of the most used programming languages to develop Web applications [41], [42]). In the following code snippet, we illustrate the matching of pattern *abc* using this object-oriented implementation. Lines 1 to 5 define the pattern *abc*. Lines 4 and 5 use an object composition to set the sequence. The solution, which is composed of a cell and two rules, is defined in Line 9. Line 13 uses the defined solution to match the pattern with the input "abc".

```
1  let a:Token = new Token("a");          //match a
2  let b:Token = new Token("b");          //match b
3  let c:Token = new Token("c");          //match c
4  let ab:Sequence = new Sequence(a,b);   //match ab
5  let abc:Sequence = new Sequence(ab,c); //match abc
6
7  let seed:Cell = new Cell(abc, new MetaInformation());
8
9  let sol:Solution = new Solution([seed)],        //list of seeds
10                                new OnlyOneMatch(), //cell rule
11                                new AddSeed());     //solution rule
12
13 sol.match("abc"); //find one match
```

IV. VALIDATION

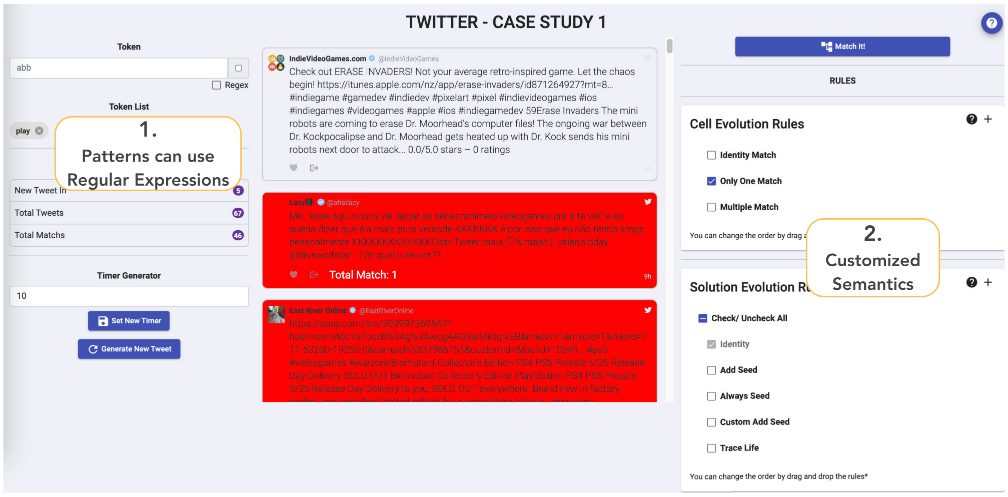
This section validates the usability and applicability of Matcher Cells through two applications, from different domains, that focus on matching stream sequences of tokens, a *distinguishing feature* of our proposal. The first application is the identification of tweets in the Twitter feed. The second application is the implementation of context identifications in a context-aware system [18]. Both implementations are available online [19].

A. APPLICATION 1: IDENTIFYING TWEETS

Fig. 7 shows our emulated Twitter environment using a real data set of 100,000 tweets related to the video game subject during 2020. Every 60 seconds (a customizable parameter), these tweets appear in the Web page's feed (the central panel in the figure). A user can identify specific tweets through the matching of a pattern in one tweet, as the figure shows with a red background. In this application, the token sequence corresponds to appearing tweets as time passes; this behavior is similar to that of streaming services.

As tweets are freely written, the same concept in a tweet might be written in different ways, for example, "play","play station", or "ps1" all refer to the same concept. Additionally, a concept can be expressed more times than others, identifying potentially more enthusiastic tweets. Taking into account the previous two observations, this Web application exhibits two features of Matcher Cells: *regular expressions* and *multiple matching* semantics. We highlight that, although the definition of a specific pattern language is not part of our proposal, it is not difficult to use a pattern language specification.

In this application, the multiple matches semantics is used to identify the intensity of a pattern. Fig. 7 shows that users can select this semantics in the Web page. As regular expressions (regex) are used to match different strings that can represent a same concept because these strings follow a similar structured form, users can enable the use of regexs to match similar tweets in this application. To implement regex in our proposal, we define cells that if they do (not) match a token, these cells create other cells that expect to match the following term in a regex. For example, Fig. 8 shows the matching process of a^+b. When token a is matched, the



TWITTER - CASE STUDY 1

Token

abb

☐ Regex

Token List

play

1. Patterns can use Regular Expressions

New Tweet In 5

Total Tweets 67

Total Matchs 46

Timer Generator

10

Set New Timer

Generate New Tweet

IndieVideoGames.com @IndieVideoGames

Check out ERASE INVADERS! Not your average retro-inspired game. Let the chaos begin! https://itunes.apple.com/nz/app/erase-invaders/id871264927?mt=8... #indiegame #gamedev #indiedev #pixelart #pixel #indievideogames #ios #indiegames #videogames #apple #ios #indiegamedev 59Erase Invaders The mini robots are coming to erase Dr. Moorhead's computer files! The ongoing war between Dr. Kockpocalipse and Dr. Moorhead gets heated up with Dr. Kock sends his mini robots next door to attack... 0.0/5.0 stars – 0 ratings

Lacy: @afrailacy

Me: "esse aqui nunca vai largar as séries/animes/videogames pra ir te ver" e eu queria dizer que é a mais pura verdade KKKKKKK é por isso que eu não tenho amgs pessoalmente KKKKKKKKKKKKKCitar Tweet maia ♡'s maah || valen's bday @favswolfstar · 12h qual o de vcs??

Total Match: 1 9h

East River Online @EastRiverOnline

https://ebay.com/itm/303997369547?hash=item46c7a7bccb%3Ag%3AwcgAAOSwMKtglxER&mkevt=1&mkcid=1&mkrid=711-53200-19255-0&campid=5337986751&customid=&toolid=10049... #ps5 #videogames #warzoneBiomutant Collector's Edition PS4 PS5 Presale 5/25 Release Day Delivery SOLD OUT Biomutant Collector's Edition PlayStation PS4 PS5 Presale 5/25 Release Day Delivery to you, SOLD OUT everywhere. Brand new in factory

Match It!

RULES

Cell Evolution Rules

☐ Identity Match

☑ Only One Match

☐ Multiple Match

You can change the order by drag a

2. Customized Semantics

Solution Evolution Ru

☐ Check/ Uncheck All

☑ Identity

☐ Add Seed

☐ Always Seed

☐ Custom Add Seed

☐ Trace Life

You can change the order by drag and drop the rules*

FIGURE 7. Web application that uses Matcher Cells to match tweets.

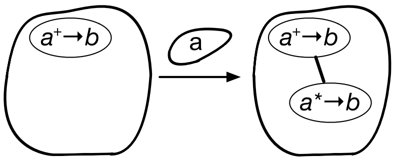


FIGURE 8. Using regular expressions in Matcher Cells.

cell creates a new cell with the pattern a^*b. Additionally, the following code snippet shows the use of functions that sketch the a^+ regex operator implementation for Matcher Cells in TypeScript.[1] While the star function returns a function that continues matching the same pattern (a^*) until a different token appears, the plus function applies star when there is a match ($a^+ = aa^*$).

```
function star(op:function) {
  return function inner(token:string):function {
    let result:function = op(token);
      if (/* result does not match */) return result; //a match cell
      if (/* result matches */) return inner;
} }

function plus(op:function) {
  return function inner(token:string) {
    let result:function = op(token);
    if (/* result does not match */) return inner; //end of the match
    if (/* result matches */) return star(token); //continue with star
 } }


let a_plus:function = plus("a"); //this function matches "a+"
```

Given that Matcher Cells ess more complex regular expression-based patterns to match URLs, Hashtags (#), or mentions (@). For example, a programmer might need to match all tweets containing a given URL.

B. APPLICATION 2: IDENTIFYING CONTEXTS

A context-aware system *adapts its behavior* according to the current *identified context* [18] from its surrounding execution environment. Fig. 9 shows a context-aware system that adapts the difficulty of addition exercises tasked to students, according to their performance (*i.e.,* context). In this system, we identify three contexts:

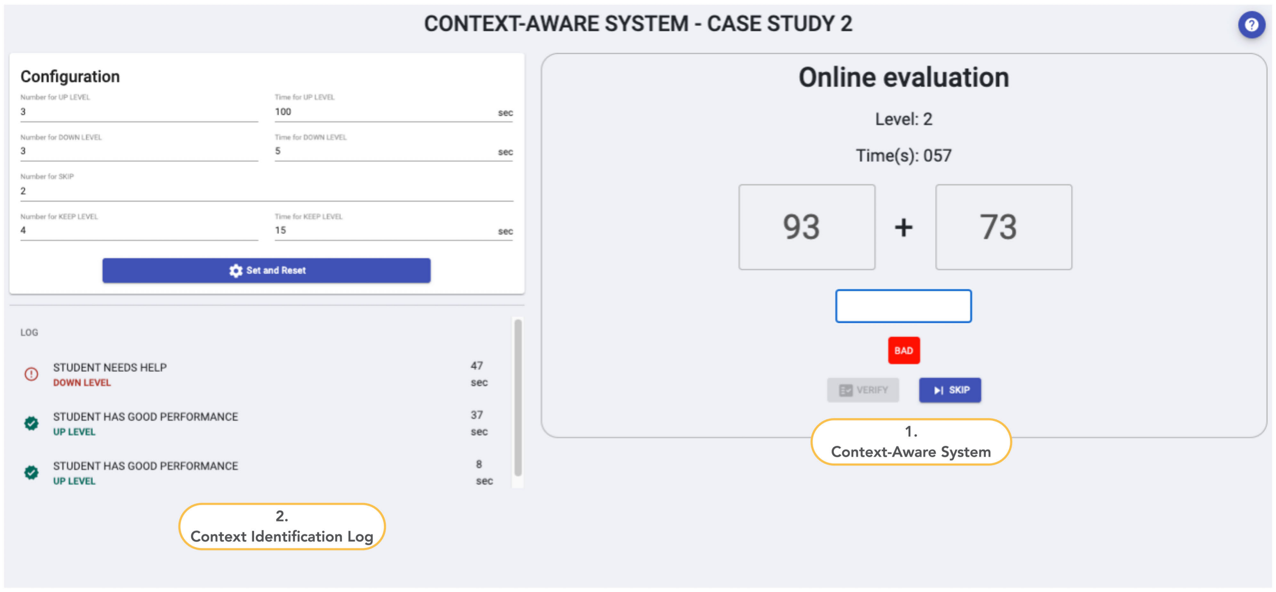
1) ***Good performance***. If a student answers a number (parametrized) of exercises *correctly* in a row, the system *increases* by one the number of digits of both terms in the following exercises.
2) ***Bad performance***. If a student answers a number (parametrized) of exercises *wrong* in a row, the system *decreases* by one the number of digits of both terms in the following exercises.
3) ***No performance evaluated***. If a student *skips* a number (parametrized) of exercises, the system *starts* from the first level (*i.e.,* additions with one digit).

The context-aware system can use the events associated with *correct*, *wrong*, and *skip* exercises to identify the previous contexts. Given that the Matcher Cells algorithm matches a sequence of tokens, we modified the callbacks of these events to add the creation of the tokens correct, wrong, and skip. These tokens represent the respective events, and the sequence of these tokens represents a stream of events. To implement this context-aware system, we added three Matcher Cells instances, where each instance is used to identify a particular context. When one Matcher Cells instance matches a pattern, the system executes its associated adaptation, *e.g.,* adding one digit in the addition terms in the *Good Performance* context.

V. EMPIRICAL EVALUATION

In addition to the usability and applicability validation of Matcher Cells in the previous section. We now turn our attention to the evaluation of the developer experience, algorithm performance, and programming abstraction expressiveness of self-replication algorithms for pattern matching.

[1]A full implementations of this and other operators are available on https://github.com/pragmaticslaboratory/match-cell-base. In this implementation, the functions are exchanged with objects.



Configuration

Number for UP LEVEL: 3
Time for UP LEVEL: 100 sec
Number for DOWN LEVEL: 3
Time for DOWN LEVEL: 5 sec
Number for SKIP: 2
Number for KEEP LEVEL: 4
Time for KEEP LEVEL: 15 sec

```
LOG

STUDENT NEEDS HELP                 47
DOWN LEVEL                         sec

STUDENT HAS GOOD PERFORMANCE       37
UP LEVEL                           sec

STUDENT HAS GOOD PERFORMANCE        8
UP LEVEL                           sec
```

FIGURE 9. **Web application that uses Matcher Cells to identify the context associated with the student performance to resolve additions.**

A. DEVELOPER EXPERIENCE

Throughout this paper, we claim that Matcher Cells is simple to use by developers because of the rule composition; comparing to other pattern matching algorithms like regex, which is perceived as difficult by both students and professional programmers [43], [44]. This section presents the results of four evaluations related to developer experience. These evaluations were carried out by 23 undergraduate students in the third-year computer science program at the Universidad Católica del Norte (Chile). After a 40-minute session teaching Matcher Cells, the students are asked to answer five tasks of pattern matching, where they have to express the correct pattern and composition of rules in Matcher Cells, specifically using a Web interface [19] for MCJs. Table 2 contains a brief description of the five tasks, ordered by incremental complexity.

1) EXPERIENCE RESULTS

As a first result, we highlight that 100% of the developers recommended Matcher Cells for the development of pattern matching algorithms. The usability of Matcher Cells is evaluated using the System Usability Scale (SUS) [20] approach, which has been widely used for years in different contexts [45], [46], [47]. This usability evaluation works as a *proxy* to measure how error-prone is to use a matching pattern algorithm that requires to compose a set of rules before using it. The data used to create the charts and SUS evaluation are anonymized and available on http://pleger.cl/sites/matchercells/results.html (responses in Spanish and translated to English).

a: PERCENTAGE OF DEVELOPERS WHO SOLVED A TASK

Fig. 10 compares the percentage of participants who solved a task. For the first two tasks, which are the easiest ones, over

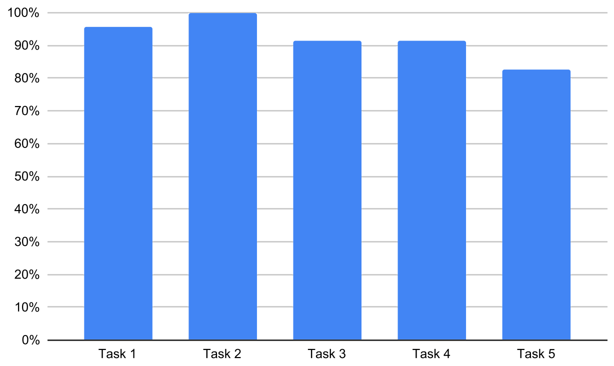


FIGURE 10. **Percentage of students that finished the task.**

95% of participants solved these tasks. The percentages of the remaining tasks had a lower success rate than the previous ones. The last task had the lowest percentage (close to 80%), given that this task is the most challenging one, as it requires the use of a time constraint to match a pattern.

b: AVERAGE TIME TO SOLVE A TASK

Fig. 11 shows the average solution time per task. For the first four tasks, the participants solved them significantly faster than the last task, which took almost double the time. As in the previous evaluation, we think this task took more time because it requires an extra configuration to be solved: the time to match a pattern.

c: HOW EASY A TASK WAS RESOLVED

Using a Likert scale [48] of five levels (from "Strongly agree" to "Strongly disagree"), we asked to the participants the question: *"How easy was the task?"*. The answers to the question are shown in Fig. 12. Although the "Strongly agree" option is not the most chosen in all tasks, there is a clear preference towards the "Agree" option; in fact, no participant

TABLE 2. Pattern matching tasks that developers asked to be solved.

Id	Description
Task_1	Use of the *Multiple Matches* rule to find four simultaneous matches
Task_2	Use of the composition of the *Single Match* and *Kill Creator* rules to find only one match
Task_3	Use of the composition of the *Single Match*, *Kill Creator*, and *add Seed* rules to find two consecutive matches
Task_4	Use of the composition of the *Multiple Matches* and *Always Seed* rules to find three matches that start with the same symbol
Task_5	Use of the composition of the *Multiple Matches* and *Trace Life-time* rules to find all possible matches that happen within 100 milliseconds

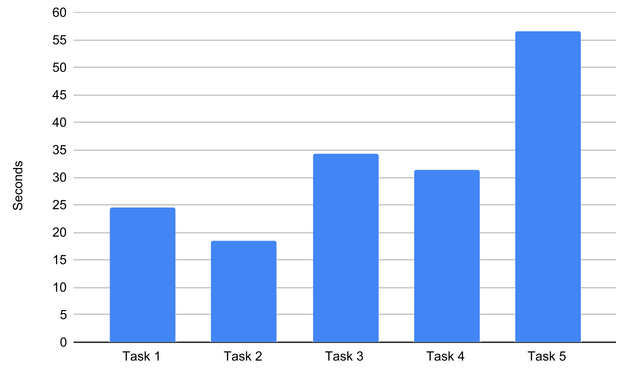


FIGURE 11. Average time per task.

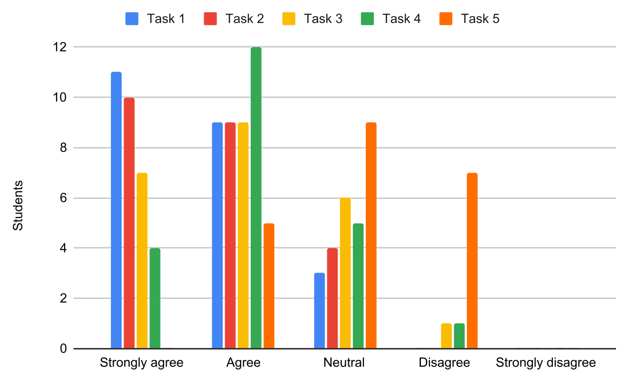


FIGURE 12. Using a Likert scale: *how easy was the task?*

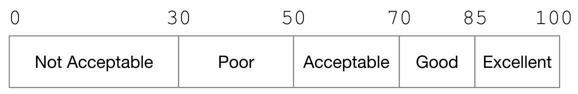


FIGURE 13. Using SUS, the value range determines how usable an interface is.

chose the "Strongly disagree" option. Using this evaluation, we might claim that the use of Matcher Cells is not complex.

d: USABILITY

To evaluate Matcher Cells in terms of usability, we used the SUS approach [20]. To use SUS, the participants who use a product (*e.g.,* software) are asked to score the following ten sentences using the five-level Likert scale:

1) "I think that I would like to use this system frequently"
2) "I found the system unnecessarily complex"
3) "I thought the system was easy to use"
4) "I think that I would need the support of a technical person to be able to use this system"
5) "I found the various functions in this system were well integrated"
6) "I thought there was too much inconsistency in this system"
7) "I would imagine that most people would learn to use this system very quickly"
8) "I found the system very cumbersome to use"
9) "I felt very confident using the system"
10) "I needed to learn a lot of things before I could get going with this system"

To calculate a final score, we follow a three-step procedure:

1) Add up the final score for all odd-numbered questions, then subtract 5 from the total to get final-odd.
2) Add up the final score for all even-numbered questions, then subtract 25 from the total to get final-even.
3) Add final-odd and final-even, then multiply the result by 2.5.

The final score is in the range of 0-100, which determines a tool's usability, shown in Fig. 13 for Matcher Cells.[2]

Fig. 14 shows, in ascending order, the score of the usability evaluation using SUS for each participant. At first glance, we observe that most participants (52.2%) find "Acceptable" the use of Matcher Cells. A percentage of 39.1% of students find its use "Good" or "Excellent", and 8.7% (equivalent to two students) find its use "Poor". The average score is 66.19, meaning that the usability to use Matcher Cells is "Acceptable", close to "Good".

e: CONCLUSION

Considering the results of our four empirical evaluations, we can observe that even if this pattern matching algorithm requires to configure a set of rules before using, developers are able to use it without a steep learning curve. The trade-off between a pre-configuration of the rule composition and the flexibility to express matching semantics can affect the preference of Matcher Cells. Nevertheless, if developers can use an external library for rule compositions, it might give preference towards Matcher Cells.

B. PERFORMANCE

The main goal of this study focuses on defining a self-replication algorithm to flexibly express matching semantics; hence, we have yet to sacrifice any potentially valuable feature based on its expected cost. Nonetheless, any pattern matching algorithm must exhibit a performance

[2]Subtle variations in the ranges can be found on the Web.

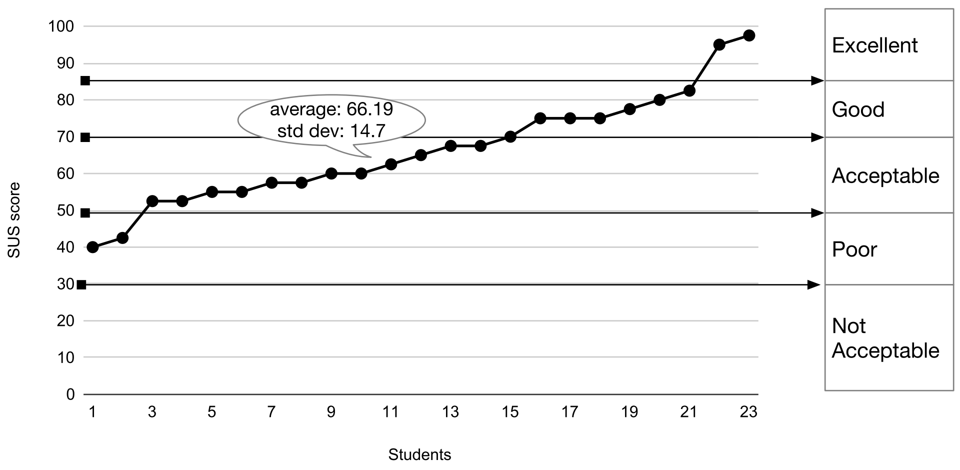


FIGURE 14. In ascending order, the SUS score for 23 students.

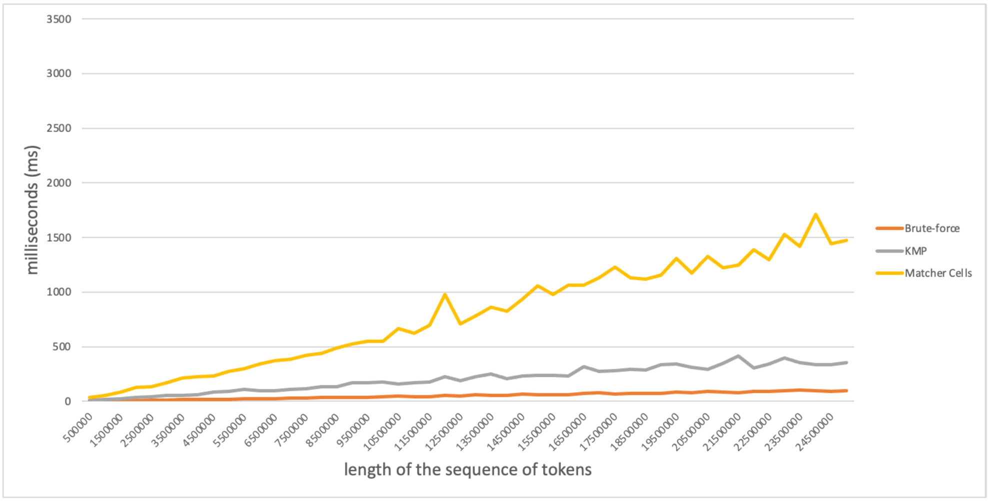


FIGURE 15. Scenario 1: Sequence of tokens $a^n x$ and pattern x.

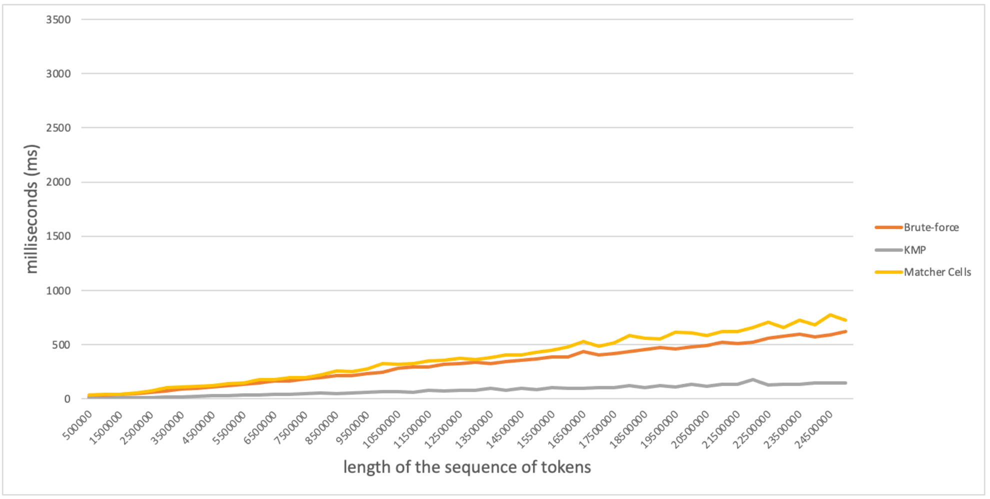


FIGURE 16. Scenario 2: Sequence of tokens $(ax)^n$ and pattern ax.

evaluation against large amounts of data. Hence, we carry out a preliminary performance evaluation using the TypeScript implementation of Matcher Cells.

In our proposal, we evaluated Matcher Cells with three scenarios with different effects, where each one increases the number of cells or rules to manage. The first scenario

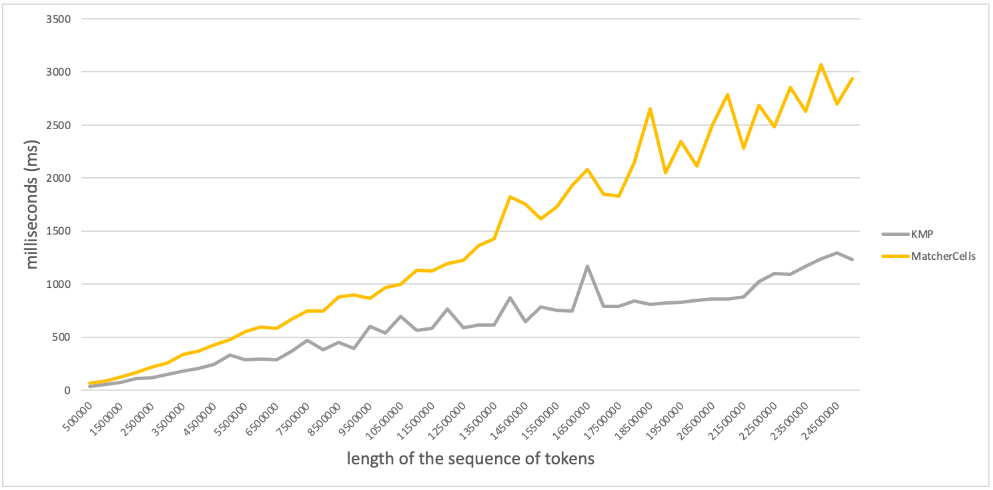


FIGURE 17. **Scenario 3: Sequence of tokens $a^n x$ and pattern a^n. Note the brute-force algorithm is not in the chart because its performance evaluation is out of the chart range, *i.e., too slow*.**

only manages *one cell* in the solution. The second scenario manages *a limited number (n) of cells* in the solution. The last one creates cells for every new token that appears in the sequence, meaning that Matcher Cells has to manage *a massive number of cells* simultaneously in the solution. To execute these three scenarios, we used NodeJS (v16) [16] on a Macbook Pro (2020) with a 2 GHz Quad-Core Intel Core i5 and 16GB of RAM running macOS Big Sur, and Matcher Cells' GitHub revision e4c556d (April 7, 2021). Fig. 15, Fig. 16, and Fig. 17 show these three scenarios of the performance evaluation comparing three algorithms (TABLE 1): *brute-force* (baseline), KMP [1] (efficient algorithm), and Matcher Cells.

For KMP, we use an implementation available on the NPM repository [49]. Each scenario's length sequence goes from 50,000 tokens to 2,500,000 tokens.

Fig. 15 shows the evaluation of the simplest scenario of a pattern consisting only of one character (x), which appears at the end of the sequence. For example, if the sequence length is 50,000, the sequence is defined by $a^{49,999}x$, and the pattern is x. Given that this scenario is simple, we can observe the brute-force algorithm has the best performance, and our proposal has the worst performance. This is because Matcher Cells and KMP require overload to work; for example, our proposal must keep cells and execute a set of rules that are composed.

Fig. 16 shows the multiple matches of the pattern ax in a sequence defined by the regex $(ax)^n$. For example, if $n = 3$, the sequence is $axaxax$, and the pattern is ax. In this case, we can see that KMP has the best performance, while our proposal has a similar performance with the brute-force algorithm. Compared to the previous Matcher Cells evaluation, the performance in this case is 3 times better for large sequences. This is because every time there is a match, all cells except the seed are removed from the solution.

Fig. 17 shows the performance evaluation of a complex pattern, a^n, in a sequence of $a^n x$. For example, if $n = 3$, the sequence is $aaax$ and the pattern is aaa. First, note that the figure does not display the brute-force algorithm. This is because its performance is orders of magnitude slower than the performance of the other algorithms. Therefore, we excluded it to be able to observe the difference between Matcher Cells and KMP. Second, although KMP is better than Matcher Cells, both algorithms have a similar trend, indicating that our proposal might be scalable to more complex sequences and patterns.

Conclusion: With this preliminary evaluation, we can observe that our proposal is not as efficient as existing pattern matching algorithms. Indeed, if we carry out a preliminary time complexity of the current version of Matcher Cells, we might estimate:

- ***Worst case.*** If no rule removes cells, the time complexity is $O(c^r * n)$, where c is the number of cells, r is number of rules, and n is the length of the input. Although this result is clearly much slower than the existing algorithms, we can observe that the use of Matcher Cells with regular expressions is better than that of the brute-force algorithm (Fig. 17). This case shows that our algorithm is exponentially time-consuming, meaning that this algorithm can be extremely slow.
- ***Best case.*** If a rule like kill creator is used, the time complexity might improve to $O(1^r * n) \rightarrow O(n)$ because only one cell is alive during the matching process. This time complexity means that Matcher Cells is linear, making it works fast. However, this efficient result only happens when a programmer wants to match the first match in an input, and not all possible matches.

Although this paper explores how we might use self-replication algorithms to flexibly match patterns, we think, as a future step, that is possible to explore efficient ways to process units like cells. For example, as our current implementation evaluates all cells within a solution for every new token, we might index or classify cells to prevent evaluations when specific tokens do not affect some cells.

C. PROGRAMMING ABSTRACTIONS AND EXPRESSIVENESS

A distinguishing feature of our proposal is the composition of simple abstractions, *i.e.,* rules, to flexibly express matching semantics. By *simple*, we mean a rule that only targets one concern in an isolated manner, where compositions of these rules are able to express advanced semantics. The use of simple abstractions boosts *modularity* [50], meaning that the reuse of abstractions (components) by allowing separate concerns. However, the effect of *tyranny of the dominant decomposition* [51] raises the following issue: a concern that does not fit into the initial view of a system ends up being tangled and scattered with other concerns, implying that this concern cannot be defined in an isolated manner. This issue appears in the rule compositions of our proposal as well.

Consider as an example a mobile context-aware system scenario where the system must match malicious patterns *if and only if* the Internet connection context is *unsafe*. For this scenario, an intuitive composition of the per-solution rules is to use the add-seed and then the kill-all-on-safe rules, as presented in the code snippet below. In this composition, add-seed creates new cells to match malicious patterns, while the kill-all-on-safe rule kills all cells, that match malicious patterns, when the Internet connection is *safe* because these patterns are only relevant in an *unsafe* context. Unfortunately, note that the composition will not match all desired malicious patterns. In particular, any pattern that starts in a *safe* context and ends the matching process in an *unsafe* context will not be matched. This is because the rule kill-all-on-safe kills any cell in a safe context, involving the cell *seeds* added by the add-seed rule. The erroneous behavior arises due to the composition of two isolated rules whose impacts affect each other, *e.g.,* kill-all-on-safe impacts on add-seed.

```
(define one—match—at—a—time—on—unsafe
   (kill—all—on—safe (add—seed identity)))
```

Regarding Matcher Cells' expressiveness, we claim the composition of simple rules in our proposal allow for an expressive definition of the matching semantics. To affirm that our proposal can express and execute any matching semantics that a Turing-complete language can express, we only need to simulate one of these languages with our rules. The λ-calculus [52] is a Turing-complete and functional programming language whose abstractions consist of functions that take one function as parameter and return a function as a result. Like the λ-calculus, Matcher Cells' rules are also functions that take functions (cells) as parameters and return functions (cells) as a result. Using the *currying* design pattern [53] to remove the need of a second parameter in rules, we can say our proposal simulates the λ-calculus; therefore, Matcher Cells is a Turing-complete expressive.

Conclusion: We can affirm that Matcher Cells users will have to understand how to compose rules; indeed, we used a 40-minute session to teach developers how to compose rules before evaluating Matcher Cells. Likewise, if functions can use all the power of a Turing-complete language, we can also affirm that these rules are expressive enough. However, the use of high-level programming abstractions and expressiveness present the following trade-off:

- ***Programming abstractions.*** Using simple abstractions like rules, developers can enhance the *flexibility to express* different matching semantics in one algorithm; being careful in the composition of rules. However, if a rule has to tangle concerns, modularity and reuse for the composition of rules might be affected.
- ***Expressiveness.*** To define rules, developers can use a Turing-complete language. However, the use of the *full power* of a language can break the spirit to be simple enough only to implement one concern of a particular matching semantics.

A potential solution is to set a boundary between abstractions and expressiveness. This boundary can be addressed by the use of a *domain-specific language* [54] to define language-level rules that enforce the simple spirit of matching rules that are expressive enough.

VI. RELATED WORK

String pattern matching has been a subject of study since the late 1970s and remains a vibrant research field due to its diverse applications that encompass a broad spectrum of domains, including intrusion detection systems, bioinformatics, web search engines, spam filters, natural language processing, and web scraping. According to [6], string matching algorithms fall into two main categories: exact string matching algorithms and approximate string matching algorithms. The former aims to identify a complete match, whereas the latter is designed to locate a substring that closely resembles a specified pattern string.

Exact string matching algorithms can be further categorized into single-pattern and multiple-pattern exact matching approaches. In single-pattern matching algorithms, the algorithm is designed to work with a single pattern as input, directing its efforts to locating that specific pattern within the target database (*i.e.,* sequence of tokens). In contrast, multiple-pattern matching algorithms are equipped to handle a single input, tasked with searching for multiple instances of that input throughout the target database. Moreover, software based exact matching algorithms can be divided into character comparison, hashing, bit-parallel, and hybrid approaches [6], [55].

Unlike exact string matching algorithms, approximate algorithms can be classified as filtration-based algorithms and backtracking-based algorithms. The first one follows a two-stage process. In the initial stage, these algorithms pinpoint potential occurrences of patterns within the text. In the subsequent stage, all of these identified locations undergo comprehensive verification. On the other hand, approximate algorithms build upon the foundations of exact string matching algorithms but precise string matching algorithms are adapted to facilitate approximate searching through edit distance operations [56], [57], [58].

For instance, the Levenshtein distance [59], also known as edit distance, is a measure of the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into another. For example, let's consider two words: "kitten" and "sitting". In total, we needed seven operations to transform "kitten" into "sitting". Thus, the levenshtein distance between these two words is 7. The smaller the levenshtein distance, the more similar the words are in terms of their spelling or structure. Such approaches often promote the utilization of compact data and data structures based on suffix indexing [60], [61].

To evaluate Matcher Cells against the most relevant approaches in terms of flexibility, we categorize them into three sets: classical general-purpose algorithms, domain-specific algorithms, and nature-inspired models.

A. CLASSICAL GENERAL-PURPOSE ALGORITHMS

Classical general-purpose algorithms for string pattern matching are foundational techniques designed to efficiently locate occurrences of a specific pattern within a given text or a sequence of tokens. These algorithms encompass character-based, hashing, suffix automata, bit-parallel, and hybrid approaches, as categorized by [62]. A concise overview of each follows.

The character-based approach is a classical method that addresses string matching problems by directly comparing individual characters. This method does not entail any preprocessing and typically involve two essential stages: the searching phase and the shift phase. Previous research has sought enhancements for both stages. Significantly, among various character-based approaches, the BM algorithm [25] stands as a benchmark and standard method in the field.

The suffix automaton is a composite structure involving two interconnected yet separate automaton constructors: the deterministic acyclic finite state automaton, which serves as a data structure representing a finite set of strings, and the suffix automaton, a finite automaton functioning as a suffix index for matching purposes [63]. This strategy is suitable for long-length patterns and performs very well because as it gives a pre-generated prefix table, so the procedure allows skipping certain comparisons during matching.

KMP [1] and BM [25] are examples of algorithms that uses the concept of automata, mainly focus on performance. The Matcher Cells algorithm, on the contrary, focuses on the runtime flexibilities that enables developers to customize matching semantics, inspired by the self-replicating behavior of cells. Therefore, we should not directly compare Matcher Cells with existing proposals in terms of performance; rather we should compare them regarding the flexibility to match patterns in different ways.

In hashing-based strategies, characters are represented by hash values rather than being compared individually, significantly reducing computational overhead through the comparison of integer values instead of characters [64]. For instance, the Karp-Rabin algorithm [24] employs this method to address string matching challenges, conducting comparisons from left to right. However, the approach is constrained by hash collisions, where two distinct strings may map to the same numerical value. While these methods accelerate pattern matching, they ultimately rely on character-based comparisons and lack the runtime flexibilities offered by Matcher Cells.

Other classical algorithms are bit-parallel and hybrid approaches. The first one relies on the principles of parallel computing, reducing the number of operations within the algorithm to match the number of bits in a computer word [65]. This algorithm demonstrates speed and efficiency, particularly when the length of the provided pattern p is shorter than the word length [64]. The second one combines the advantages of different algorithms and is performs better than individual algorithms [66]. These approaches can combine one or more character-based methods, one or more methods from automata-based and character- and automata-based methods [6]. In terms of flexibility, each of them lacks the option for semantic customization.

B. DOMAIN-SPECIFIC ALGORITHMS

Domain-specific algorithms for pattern matching are tailored methods designed to address specific application domains or types of data. Unlike general-purpose algorithms that aim for versatility across various scenarios, domain-specific algorithms are optimized to excel in particular contexts. While Matcher Cells can operate in the same domains as classical algorithms, it could be especially advantageous in domains where temporal information is crucial for pattern detection, particularly in highly dynamic environments.

In the realm of information security, specifically concerning spambots, algorithms play a crucial role in protecting digital systems and user data from the actions of automated programs engineered for spam distribution [67]. Spambots, also known as spam robots, are automated scripts or software applications designed to create and propagate unsolicited and potentially harmful content, including unwanted advertisements, phishing schemes, and malware [68]. According to [69], there are four types of spam detection techniques: content-based, link-based, machine learning, and string pattern matching-based. Subsequently, we compare Matcher Cells with techniques based on string pattern matching in the domain of spambot detection.

Alamro et al. [69] propose an algorithm that can detect one or more sequences of indeterminate actions in text T in linear time. The algorithm takes into account temporal information, because it considers time-annotated sequences and requires a match to occur within a time window t. Authors state that some spambots might attempt to disguise their actions by varying certain actions. For example, a spambot takes the actions `ABCDEF`, then `ACCDEF`, then `ABDDEF`, etc. Thus, the sequence can be described as `A[BC][CD]DEF`. Spambots try to deceive by changing the second and third action so actions `[BC]` and `[CD]` are variations of the same sequence. Additionally, spambots can execute these variations across different time frames, adding complexity to their detection.

In reference to the research conducted by Alamro et al. [69], Matcher Cells exhibits comparable functionality by amalgamating multiple rules to identify variations in actions. Employing the *One Match at a Time* and *Match per Time Frame* semantic rules, Matcher Cells seamlessly incorporates temporal information, enabling it to effectively detect spambots that disguise their actions. Furthermore, a pivotal feature distinguishing Matcher Cells could be the ability to identify multiple spambots through the application of the *Always Start a Match* rule. This rule facilitates a concurrent process of matching sequences, leading to the simultaneous identification of multiple spambots.

To detect the spambot sequences, the algorithm requires as input sequences temporally annotated actions from user logs. Specifically, each request in a user log is mapped to a predefined index key in the sequence and the date timestamp for the request in the user log is mapped to a time point in the sequence. Then, by using Manber and Myers algorithm [29] and bit masking operation, the algorithm can detect one or more indeterminate sequences in a Web user log.

Ghanaei et al. [70] present a technique for identifying Web spambots, addressing spam-related issues on the Web. This method relies on analyzing Web usage behavior, extracting discriminative features known as action strings from user logs to distinguish between spambot and human actions. An action is defined as a set of user efforts aimed at achieving specific purposes, while action strings represent sequences of actions for a particular user in a transaction. To implement a real-time, on-the-fly classification method, the authors construct a trie data structure based on action strings. Within this structure, each trie edge includes an action key index, and each node incorporates the probability of a given action string being associated with either human or spambot behavior. Consequently, new actions can be classified into two categories: Match and NotMatched.

Hayati et al. [68] introduce a method for detecting web spambots. The authors propose a rule-based approach that analyzes web usage behavior action strings using Trie data structures. These action strings are indicative of spambot activity. The system is designed for on-the-fly classification, meaning it can quickly and effectively identify web spambots in real-time.

In light of the research conducted by Ghanaei et al. [70] and Hayati et al. [68], it is anticipated that Matcher Cells would exhibit superior performance in spambot detection. This expectation arises from the observation that these studies do not incorporate temporal information. Therefore, in the context of evaluating the most pertinent works on string-based approaches to spambot detection, Matcher Cells is expected to showcase enhanced flexibility in rule composition, thereby boosting its effectiveness in identifying spambots.

C. NATURE-INSPIRED MODELS

Nature-inspired models are computational or mathematical models that draw inspiration from natural processes, phenomena, or systems observed in the natural world. We have identified two models for comparison with Matcher Cells: cellular automaton and chemical abstract machine. We elaborate on these comparisons below.

A cellular automaton [71] is a collection of cells that evolves through a number of discrete time steps according to a set of simple rules based on the states of neighboring cells. In contrast to its simplicity, cellular automata can model complex behavior in various areas such as physics, engineering, and theoretical biology. For example, it is known that pores of leaves in plants can be represented using a cellular automaton [72]. Similarly, in Matcher Cells, a programmer can use a composition of simple rules to define their own matching semantics. Although cellular automata can be Turing-complete, so it can be applied to pattern matching, to the best of our knowledge there is no research proposing them as concrete interfaces for this subject. In contrast, our proposal provides a concrete interface of rules that are composed and applied for pattern matching.

A chemical abstract machine [73] is a model for asynchronous concurrent computations. This solution borrows an idea from a chemical solution in which floating molecules can interact with each other according to reaction rules, allowing for contact among molecules. This model gives expressive power to proposals such as Petri Nets [74] in concurrent programming. Indeed, it is possible to represent the full Calculus of Communicating Systems (CSS) [75] using elements such as *agents*, *molecules* and *rules* defined in chemical abstract machines. Matcher Cells also adopt a set of reaction rules to represent expressiveness for defining programmers' own semantics. It also might be possible to improve the performance of Matcher Cells, introducing the concept of a membrane that encapsulates molecule evolutions locally [73].

VII. CONCLUSION

The field of pattern matching algorithms has been vastly studied, with solid contributions from the research community. Most of the contributions in the field are related to efficiency and performance, leaving the flexibility to express different matching semantics aside. As a consequence, developers of these algorithms need to learn many algorithmic techniques, tweak them in contortive ways, or create new specialized techniques altogether if their specific needs are not supported off-the-shelf. This paper explores the use of self-replication algorithms to express different matching semantics flexibly. As a result of this exploration, we propose Matcher Cells, an algorithm inspired by the self-replication behavior of cells that allows developers to match patterns flexibly. The matching semantics of Matcher Cells is expressed by the composition of simple match rules. We provide a functional description of our proposal to implement it in any Turing-complete language that provides functions like abstractions. Additionally, we provide a concrete implementation for TypeScript used to evaluate our proposal by means of two applications for streaming data sequences. Additionally, we evaluate the performance of our approach with an empirical evaluation to assess the usability aspects of Matcher Cells.

Considering this paper as a first step to propose self-replication algorithms to match patterns, there are still some open issues to address. For example, although performance is beyond the scope of our evaluation, we are aware that the current implementation needs to improve its performance. We plan to explore and evaluate index strategies in cells and rules to solve this issue.

ACKNOWLEDGMENT

The authors want to thank Marcelo Lazo (marcelo.lazo@alumnos.ucn.cl), an undergraduate student from Universidad Católica del Norte, Chile, who implemented the TypeScript version of Matcher Cells. Additionally, they thank Éric Tanter (etanter@dcc.uchile.cl) for providing initial ideas.

REFERENCES

[1] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, Jun. 1977.

[2] J. Sakarovitch, *Elements of Automata Theory*. Cambridge, U.K.: Cambridge Univ. Press, Oct. 2009.

[3] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, Jun. 1968.

[4] L. Chen, S. Lu, and J. Ram, "Compressed pattern matching in DNA sequences," in *Proc. IEEE Comput. Syst. Bioinf. Conf.*, Standford, CA, USA, Aug. 2004, pp. 62–68.

[5] B. A. Hamed, O. A. S. Ibrahim, and T. Abd El-Hafeez, "A survey on improving pattern matching algorithms for biological sequences," *Concurrency Comput., Pract. Exper.*, vol. 34, no. 26, p. e7292, Nov. 2022.

[6] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran, "Exact string matching algorithms: Survey, issues, and future research directions," *IEEE Access*, vol. 7, pp. 69614–69637, 2019.

[7] G. Barbera, L. Araujo, and S. Fernandes, "The value of web data scraping: An application to TripAdvisor," *Big Data Cognit. Comput.*, vol. 7, no. 3, p. 121, Jun. 2023.

[8] M. Khder, "Web scraping or web crawling: State of art, techniques, approaches and application," *Int. J. Adv. Soft Comput. Appl.*, vol. 13, no. 3, pp. 145–168, Dec. 2021.

[9] P. Gao, M. Saeki, J. Guo, and H. Han, "Stable web scraping: An approach based on neighbour zone and path similarity of page elements," *Int. J. Web Eng. Technol.*, vol. 13, no. 4, pp. 301–333, 2018.

[10] A. K. Kar, "Bio-inspired computing: A review of algorithms and scope of applications," *Exp. Syst. Appl.*, vol. 59, pp. 20–32, Oct. 2016.

[11] J. V. Neumann, *Theory of Self-Reproducing Automata*. Champaign, IL, USA: Univ. Illinois Press, 1966.

[12] P. Leger and É. Tanter, "A self-replication algorithm to flexibly match execution traces," in *Proc. 11th Workshop Found. Aspect-Oriented Lang.*, Potsdam, Germany, Mar. 2012, pp. 27–32.

[13] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," in *Special Issues in Object-Oriented Programming*. Berlin, Germany: Springer, 1996.

[14] R. A. Kelsey and J. A. Rees, "A tractable scheme implementation," *LISP Symbolic Comput.*, vol. 7, no. 4, pp. 315–335, 1994.

[15] TypeScript. (Oct. 1, 2023). *JavaScript With Syntax for Types*. [Online]. Available: https://www.typescriptlang.org

[16] The OpenJS Foundation. (Oct. 1, 2023). *NodeJS: A Javascript Runtime Built for the Server Side*. [Online]. Available: https://nodejs.org

[17] Twitter. (Oct. 1, 2023). *A Microblogging and Social Networking Service*. [Online]. Available: http://twitter.com

[18] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Pers. Commun.*, vol. 8, no. 4, pp. 10–17, Aug. 2001.

[19] P. Leger and M. Lazo. (Oct. 1, 2023). *Case Studies of Matcher Cells*. [Online]. Available: http://pragmaticslaboratory.github.io/matcher-cells-study-cases

[20] J. Brooke, *Usability Evaluation in Industry*. Boca Raton, FL, USA: CRC Press, 1996.

[21] A. Apostolico and Z. Galil, *Pattern Matching Algorithms*. Oxford, U.K.: Oxford Univ. Press, 1997.

[22] P. Leger, É. Tanter, and H. Fukuda, "An expressive stateful aspect language," *Sci. Comput. Program.*, vol. 102, pp. 108–141, May 2015.

[23] S. Kumar and E. H. Spafford, "A pattern matching model for misuse intrusion detection," in *Proc. 17th Nat. Comput. Secur. Conf.*, Oct. 1994, pp. 11–21.

[24] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, Mar. 1987.

[25] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[26] G. Navarro and M. Raffinot, "A bit-parallel approach to suffix automata: Fast extended string matching," in *Proc. Annu. Symp. Combinat. Pattern Matching*. Cham, Switzerland: Springer, 1998, pp. 14–33.

[27] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching," in *Proc. Conf. Current Trends Theory Pract. Informat.* Cham, Switzerland: Springer, Nov. 1999, pp. 295–310.

[28] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th Annu. Symp. Switching Automata Theory*, Oct. 1973, pp. 1–11.

[29] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, Oct. 1993.

[30] M. Crochemore and D. Perrin, "Two-way string-matching," *J. ACM*, vol. 38, no. 3, pp. 650–674, Jul. 1991.

[31] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.

[32] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.

[33] M. Rubinchik and A. M. Shur, "EERTREE: An efficient data structure for processing palindromes in strings," *Eur. J. Combinatorics*, vol. 68, pp. 249–265, Feb. 2018.

[34] B. Meyer, "Incremental string matching," *Inf. Process. Lett.*, vol. 21, no. 5, pp. 219–227, Nov. 1985.

[35] Amazon. (Oct. 1, 2023). *AWS WAF and AWS Shield Documentation*. [Online]. Available: https://aws.amazon.com/documentation/waf

[36] G. Springer and D. P. Friedman, *Scheme and the Art of Programming*. Cambridge, MA, USA: MIT Press, 1989.

[37] H. R. Maturana and F. J. Varela, *Autopoiesis and Cognition: The Realization of the Living*, vol. 42. Berlin, Germany: Springer, 2012.

[38] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *Proc. 20th ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang. Appl.*, San Diego, CA, USA, Oct. 2005, pp. 365–383.

[39] T. Shoji, M. Takimoto, and Y. Kambayashi, "Capture of multi intruders by cooperative multiple robots using mobile agents," in *Proc. 12th Int. Conf. Agents Artif. Intell.*, Valletta, Malta, 2020, pp. 370–377.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Professional Computing Series). Reading, MA, USA: Addison-Wesley, Oct. 1994.

[41] W3 Techs. (Oct. 1, 2023). *Usage of Client-side Programming Languages*. [Online]. Available: https://w3techs.com/technologies/history_overview/client_side_language/all

[42] Stackover Flow. (Sep. 1, 2022). *Developer Survey Results*. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[43] C. W. Brown and E. A. Hardisty, "RegeXeX: An interactive system providing regular expression exercises," in *Proc. 38th SIGCSE Tech. Symp. Comput. Sci. Educ.* New York, NY, USA: Association for Computing Machinery, Mar. 2007, pp. 445–449.

[44] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 415–426.

[45] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," *Int. J. Hum.-Comput. Interact.*, vol. 24, no. 6, pp. 574–594, Jul. 2008.

[46] D. Derisma, "The usability analysis online learning site for supporting computer programming course using system usability scale (SUS) in a university," Int. Assoc. Online Eng., Austria, Tech. Rep., Jun. 2020.

[47] P. Vlachogianni and N. Tselios, "Perceived usability evaluation of educational technology using the system usability scale (SUS): A systematic review," *J. Res. Technol. Educ.*, vol. 54, no. 3, pp. 392–409, May 2022.

[48] G. Albaum, "The Likert scale revisited," *Market Res. Soc. J.*, vol. 39, no. 2, pp. 1–21, Mar. 1997.
[49] M. Mota. (2022). *A Concrete Implementation of KMP Available on the NPM Repository*. Accessed: Oct. 1, 2023. [Online]. Available: https://www.npmjs.com/package/kmp
[50] D. Parnas, "On the criteria for decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[51] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, "N degrees of separation: Multi-dimensional separation of concerns," in *Proc. Int. Conf. Softw. Eng.*, Los Angeles, CA, USA, May 1999, pp. 107–119.
[52] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
[53] H. B. Curry, "Some philosophical aspects of combinatory logic," in *Proc. The Kleene Symp.*, vol. 101, J. Barwise, H. J. Keisler, and K. Kunen, Eds. 1980, pp. 85–101.
[54] A. Van Deursen and P. Klint, "Domain-specific language design requires deature descriptions," *J. Comput. Inf. Technol.*, vol. 10, no. 1, pp. 1–17, 2002.
[55] K. Al-Khamaiseh and S. Al Shagarin, "A survey of string matching algorithms," *Int. J. Eng. Res. Appl.*, vol. 4, pp. 144–156, Aug. 2014.
[56] M. Farach-Colton, G. M. Landau, S. C. Sahinalp, and D. Tsur, "Optimal spaced seeds for faster approximate string matching," *J. Comput. Syst. Sci.*, vol. 73, no. 7, pp. 1035–1044, Nov. 2007.
[57] J. Kärkkäinen and J. C. Na, "Faster filters for approximate string matching," in *Proc. Meeting Algorithm Eng. Experiments*, 2007, pp. 84–90.
[58] G. Kucherov, L. Noé, and M. Roytberg, "Multi-seed lossless filtration," in *Combinatorial Pattern Matching*, S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, Eds. Berlin, Germany: Springer, 2004, pp. 297–310.
[59] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau? Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009.
[60] G. Kucherov, K. Salikhov, and D. Tsur, "Approximate string matching using a bidirectional index," *Theor. Comput. Sci.*, vol. 638, pp. 145–158, Jul. 2016.
[61] G. Navarro and R. Baeza-Yates, "A hybrid indexing method for approximate string matching," *J. Discrete Algorithms*, vol. 1, pp. 205–239, Jan. 2001.
[62] A. N. M. E. Rafiq, M. W. El-Kharashi, and F. Gebali, "A fast string search algorithm for deep packet classification," *Comput. Commun.*, vol. 27, no. 15, pp. 1524–1538, Sep. 2004.
[63] W. Yang, "Mealy machines are a better model of lexical analyzers," *Comput. Lang.*, vol. 22, no. 1, pp. 27–38, Apr. 1996.
[64] A. Akram Abdulrazzaq, N. Abdul Rashid, A. Hasan, and M. Abu-Hashem, "The exact string matching algorithms efficiency review," *Global J. Technol.*, vol. 4, pp. 576–589, Jan. 2013.
[65] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 1–42, Mar. 2013.
[66] F. Franek, C. G. Jennings, and W. F. Smyth, "A simple fast hybrid pattern-matching algorithm," *J. Discrete Algorithms*, vol. 5, no. 4, pp. 682–695, Dec. 2007.
[67] P. Heymann, G. Koutrika, and H. Garcia-Molina, "Fighting spam on social web sites: A survey of approaches and future challenges," *IEEE Internet Comput.*, vol. 11, no. 6, pp. 36–45, Nov. 2007.
[68] P. Hayati, V. Potdar, A. Talevski, and W. Smyth, "Rule-based on-the-fly web spambot detection using action strings," in *Proc. Annu. Collaboration, Electron. Messaging, Anti-Abuse Spam Conf.*, 2010, pp. 1–7.
[69] H. Alamro, C. S. Iliopoulos, and G. Loukides, "Efficiently detecting web spambots in a temporally annotated sequence," in *Advanced Information Networking and Applications*. Cham, Switzerland: Springer, 2020, pp. 1007–1019.
[70] V. Ghanaei, C. S. Iliopoulos, and S. P. Pissis, "Detection of web spambot in the presence of decoy actions," in *Proc. IEEE 4th Int. Conf. Big Data Cloud Comput.*, Dec. 2014, pp. 277–279.
[71] P. Sarkar, "A brief history of cellular automata," *ACM Comput. Surv.*, vol. 32, no. 1, pp. 80–107, Mar. 2000.
[72] D. Peak, J. D. West, S. M. Messinger, and K. A. Mott, "Evidence for complex, collective dynamics and emergent, distributed computation in plants," *Proc. Nat. Acad. Sci. USA*, vol. 101, no. 4, pp. 918–922, Jan. 2004.
[73] G. Berry and G. Boudol, "The chemical abstract machine," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 217–248, Apr. 1992.
[74] W. Reisig, *Petri Nets: An Introduction*, vol. 4. Berlin, Germany: Springer, 2012.
[75] R. Milner, *A Calculus of Communicating Systems*. Berlin, Germany: Springer, 1982.

PAUL LEGER (Member, IEEE) received the Ph.D. degree in computer science from the University of Chile. He is currently an Associate Professor with Universidad Católica del Norte, Chile. His research interests include issues related to programming languages, software engineering, and different programming approaches.

HIROAKI FUKUDA received the Ph.D. degree in computer science from Keio University. He is currently an Associate Professor with the Shibaura Institute of Technology, Japan. His research interests include software engineering and distributed programming.

NICOLÁS CARDOZO received the joint Doctoral Diploma degree from Université Catholique de Louvain and Vrije Universiteit Brussel, Belgium. He was a Postdoctoral Fellow with Trinity College Dublin and Vrije Universiteit Brussel. He is currently an Associate Professor with Universidad de los Andes, Colombia. His research interest includes the design and implementation of programming languages for distributed adaptive software systems. He has worked in the implementation of dynamic distributed adaptations in the smart cities domain from different perspectives, such as automated personalized assistants and evolutionary models for dynamic adaptations.

DANIEL SAN MARTÍN received the B.S. degree in engineering science and computer engineering from Universidad Católica del Norte, Coquimbo, Chile, and the M.Sc. and Ph.D. degrees in computer science from Universidade Federal de São Carlos, Brazil. He is currently an Assistant Professor with the School of Engineering, Universidad Católica del Norte. Also, he has held pivotal positions as a Chief Information Security Officer (CISO), the Project Manager (PM), and the Information Analyst in both public and private sectors. His current research interests include software engineering, software architecture, programming languages, and models.