Automated Discovery of CEP Applications with Evolutionary Computing

Automated Discovery of CEP Applications with Evolutionary Computing

Giulio Appetito
Department of Civil Engineering and Computer Science Engineering (DICII)
University of Rome Tor Vergata
Roma, Italy
giulio.appetito@alumni.uniroma2.eu

Eric Medvet
Department of Engineering and Architecture
University of Trieste
Trieste, Italy
emedvet@units.it

Vincenzo Gulisano
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Abstract

Complex event processing (CEP) is key for detecting patterns in digital systems (e.g., smart grids and vehicular networks) through platforms like Apache Flink CEP that decouple application logic from distributed execution in cloud-to-edge infrastructures. Yet, a barrier remains: system experts can identify relevant patterns but often lack programming skills to implement CEP applications, limiting effective use.

We present a preliminary study on using evolutionary computation to automate CEP application discovery from data. Experts provide examples of relevant event sequences for an evolutionary algorithm to evolve applications to detect similar patterns. Initial results are promising and highlight CEP-related challenges that open new research directions.

CCS Concepts

• **Information systems** → **Data management systems**; • **Theory of computation** → **Evolutionary algorithms**; **Streaming models**.

Keywords

Complex Event Processing; Evolutionary Computing

ACM Reference Format:
Giulio Appetito, Eric Medvet, and Vincenzo Gulisano. 2025. Automated Discovery of CEP Applications with Evolutionary Computing. In *The 19th ACM International Conference on Distributed and Event-based Systems (DEBS '25), June 10–13, 2025, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3701717.3730548

DEBS '25, Gothenburg, Sweden

ACM ISBN 979-8-4007-1332-3/25/06
https://doi.org/10.1145/3701717.3730548

1 Introduction

Complex event processing (CEP) is key for monitoring/detecting service status and performance patterns in cloud-to-edge architectures and digital systems that rely on event-producing sensors (e.g., smart grids and vehicular networks). Its growing adoption is driven by the ability to define rich and complex event patterns through expressive application programming interfaces (APIs) and decouple application semantics from deployment and execution, allowing CEP applications to scale up and out in cloud-to-edge continua.

As CEP systems gain traction, their usage expands from cloud programmers to system experts with deep domain knowledge. The latter can identify meaningful patterns, but often lack skills to program CEP applications in modern frameworks, a gap limiting broad adoption of CEP systems.

To address this challenge, we study the *automated discovery of CEP applications*, which has received limited attention despite the success of automatic program discovery in other domains [19]. On the one hand, automated discovery can help find patterns like those marked by an expert. On the other hand, a pattern found through automated discovery can support inference through a concise description of the matching sequences if an underlying common pattern exists.

Specifically, we investigate using *evolutionary computation (EC)* to generate applications from sample event sequences of desired patterns. EC iteratively evolves a population of candidate solutions through mechanisms inspired by natural selection (e.g., mutation, crossover, and selection) [7]. In this context, candidate solutions are CEP applications whose fitness depends on how accurately they detect the expected patterns in input event streams. For representing applications in a way that can be evolved with an evolutionary algorithm (EA), we rely on context-free grammar GP (CFG-GP), an EA where candidate solutions are strings of a language defined by a user-provided grammar [20]. A key advantage of CFG-GP is its inherent explainability. Since

CEP applications are explicitly defined and interpretable, system experts can understand/refine the generated solutions, ensuring alignment with domain-specific requirements.

Our initial study on user access tracking shows this approach is promising. Key research challenges remain, though: (1) reducing the vast search space introduced by CEP system APIs and (2) ensuring scalable fitness evaluations.

Organization: we introduce preliminary concepts about CEP and EC in Section 2, we cover our problem statement in Section 3, discuss our initial use case and result in Section 4, present related work in Section 5, and conclude in Section 6.

2 Preliminaries

2.1 Complex event processing (CEP)

CEP is a stream processing paradigm used to identify key patterns from streams of *events* carrying a timestamp and application-specific attributes. CEP systems like Apache Flink CEP [8] offer an API which allows users to define patterns declaratively, specifying which events to detect, in what order, the time constraints between consecutive or all events, and event grouping using key-by partitioning.

Key features are *contiguity constraints* and *skip strategies* for overlapping matches. For the former, strict contiguity requires exact sequence matches; relaxed ignores intervening events but restarts after each match; non-deterministic relaxed allows overlaps and new matches to start at any qualifying event. For the latter, different strategies control the number of matches each event can contribute to. We focus on the *skipToNext* strategy, discarding any partial match starting with an event for which a complete match is found. Contiguity/skip strategy choices greatly affect performance:

Example 2.1. Suppose a stream carries the number of daily failed login events from a set of servers. A pattern might detect servers whose failed logins grow above a threshold T_1, keep growing for 1 to 5 events, and eventually grow over threshold T_2 during a week, indicating a security threat.

```
1 Pattern.begin("start").where(event.
    failed_log > T1)
2  .followedBy("middle").times(1,5)
3  .where(current.failed_log > prev.
    failed_log)
4  .followedBy("final").where(event.
    failed_log > T2)
5  .within(Time.weeks(1)).keyby(event.
    server_id);
```

Strict contiguity matches only non-decreasing sequences. Relaxed contiguity allows sequences with occasional drops. Non-deterministic relaxed contiguity enables overlapping matches, potentially causing a combinatorial explosion and high computational cost.

2.2 Evolutionary computation (EC)

EC [7] encompasses optimization methods inspired by natural evolution; EAs implement these via iterative, population-based search. Given a search space S and a fitness function $f : S \to \mathbb{R}$ (we assume, without loss of generality, to be maximized), an EA: (1) generates a population of many candidate solutions (i.e., elements of S, also called *individuals*) with some non-deterministic initialization procedure, (2) repeats the following steps until some termination criterion is met: (i) it builds an offspring of many new individuals starting from the current population (called parents) by repeatedly selecting good individuals and modifying or combining them (this step is called *reproduction*), (ii) it merges the offspring and the parents to form a larger population, and (iii) trims to the size of the initial population by selecting survival individuals. Individual selection for reproduction and survival is specific to each EA, but is generally independent of the search space S. In both cases, selection is non-deterministic but based on the comparison of the fitness of pairs of individuals: i.e., given an $s_1, s_2 | f(s_1) > f(s_2)$, s_1 has a greater probability of reproduction or surviving than s_2.

The iterative procedure ends after n_{evals} evaluations of f. The mechanisms according to which individuals are modified or recombined are S-dependent and called *genetic operators*. Typical operators include mutation $o_{\text{mut}} : S \to \mathcal{P}(S)$ and crossover $o_{\text{xover}} : S \times S \to \mathcal{P}(S)$, where $\mathcal{P}(S)$ is the set of probability distributions over S. In practice, o_{mut} can be seen as a non-deterministic operator over S and o_{xover} as a non-deterministic bi-operator. The population initialization step can be formalized as a probability distribution over S.

The genetic operators $o_{\text{mut}}, o_{\text{xover}}$ (and the initialization step) are key in making an EA capable of solving a problem, establishing how the population moves (and where it starts from) in S, hopefully towards a point $s^\star$ which maximizes f. Ideally, they should have a good locality, i.e., make similar solutions have similar fitness, while granting the closure property, i.e., ensuring that each operator application produces a solution in S. Meeting these requirements for an arbitrary S is hard. We use an EA, CFG-GP, which meets this challenge with an indirect representation of solutions.

CFG-GP [20] is a form of genetic programming (GP) [11] (an EA initially adopted to evolve computer programs) where solutions are strings of a language defined by a context-free grammar (CFG). Internally, CFG-GP manipulates solutions through their genotypes, which are production trees.

Formally, a CFG G is a tuple (T, N, R, n_0) where T is a set of terminal symbols, i.e., the alphabet including all the symbols the strings of the language $\mathcal{L}(G)$ are made of, N is a set of non-terminal symbols (disjoint from T), $n_0 \in T \cup N$ is the starting symbol, and R is a set of production rules. Each production rule is composed of a left-hand-side non-terminal

symbol and a right-hand-side sequence of terminal or non-terminal symbols. A common way to concisely represent a grammar is through the Backus-Naur form (BNF). In this work, we designed a grammar for CEP applications: we include symbols and rules for representing fixed constructs (as, e.g., the where and followedBy clauses) and others for representing variable parts depending on the problem at hand (as, e.g., the event . failed_log property name).

In CFG-GP, genotypes are production trees: ordered trees where nodes are labeled with symbols in $N \cup T$, the root with n_0, terminal nodes with symbols in T, and non-terminal nodes with symbols in N. Each non-terminal node labeled with $n \in N$ has children matching (in order and labels) one of the production rules having n as left-hand side. By traversing terminal nodes from the first to the last, one can obtain a string of the language $\mathcal{L}(G)$ defined by G.

CFG-GP's operators act directly on production trees, randomly replacing a subtree with a newly generated one compatible with G and the replaced subtree's root (mutation), and randomly exchanging two compatible subtrees of the two parents (crossover). The initialization procedure produces random production trees with depth in a given interval.

Concerning the EA components not depending on S, in this work we use a standard version of GP backing CFG-GP. Namely, we use tournament selection with size n_{tour} in reproduction, truncation for survival selection, a population of n_{pop} individuals, and an offspring with the same size n_{pop}. When doing reproduction, we apply the crossover operator with probability p_{xover}, otherwise, we apply mutation.

3 Evolution of complex event processing applications

Figure 1 overviews the automated discovery of CEP applications we envision: (1) *Data sampling:* A sample event dataset is created from the live input stream. (2) *Identification of sample patterns:* An expert marks relevant event sequences. (3) *Automated customization of the grammar:* The expert possibly selects some attributes of interest or imposes bounds to thresholds or durations, which are then reflected in the problem-dependent parts of the grammar. (4) *Evolutionary optimization:* CFG-GP runs an evolution with the customized grammar and a fitness function working on the sample dataset through the CEP engine. (5) *Deployment of CEP applications:* The best evolved CEP application is deployed on live data. (6) *Continuous drift monitoring:* The expert monitors the CEP application performance, assessing whether new applications are needed or if some are obsolete.

Our preliminary study focuses on steps (2)–(4). We simulate (2) by defining a *target* CEP application and marking all matching events as those identified by an expert. Then, (3)–(4) run the evolutionary process to generate CEP

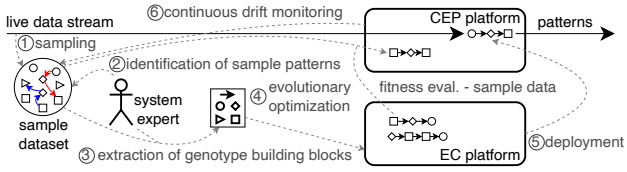

Figure 1: CEP application discovery process (overview).

applications. We analyze these steps to evaluate the applicability of EC (i.e., the quality of candidate applications and the computational cost to assess their fitness) and identify the challenges arising from the joint use of a CEP and an EA.

We assess quality using the F_β *score*, a common effectiveness measure for information retrieval and binary classification. F_β is a weighted harmonic mean of *precision* (Prec) and *recall* (Rec), defined as $F_\beta = (1+\beta^2)\frac{\text{Prec Rec}}{\beta^2\text{Prec}+\text{Rec}}$, where β determines the relative precision vs. recall importance. In our context, the precision is the rate of sequences the application matches that are actually to be matched; the recall is the rate of sequences to be matched that are matched by the application. We set $\beta = 1$ and hence operate with the F_1 score as fitness since both precision and recall are equally critical for complex events detection.

The computational cost for each candidate application is defined as the time required to deploy said application at the CEP engine, execute it, collect the matched sequences, and compare them with the ones marked by the system expert.

Concerning the grammar G, we design it to contain the following constructs from the API of Flink CEP:

Event pattern An individual event within a sequence, the fundamental building block of a CEP application.

Simple conditions Conditions applied to an event, determining whether it should be included in the pattern based solely on its attributes. Multiple conditions can be combined using boolean operators (`and`, `or`).

Quantifiers The expected number of occurrences of an event in a pattern, including:

- oneOrMore(), for one or more occurrences of an event;
- optional(), to possibly match but not require an event;
- times(n), to match exactly n occurrences of an event;
- times(a, b) to specify that the event occurs between a and b times (inclusive).
- timesOrMore(n), to match an event n or more times.

Contiguity conditions Conditions on how events are matched with one another, including:

- next(), for strict contiguity, i.e., consecutive events;
- followedBy(), for relaxed contiguity;
- followedByAny(), for non-deterministic relaxed contiguity, allowing multiple possibly overlapping sequences.

Time constraints Constructs like within(time), imposing a maximum time window for a sequence to match a pattern or to be discarded otherwise.

Keying mechanism Constructs like key_by(key), enabling the partitioning of event streams by a set of attributes.

Our code is found at https://zenodo.org/records/15299657.

4 Use case and Preliminary Results

4.1 Experimental setup

Our use cases focus on user access tracking on a server based on a live stream of login attempts. The sample dataset consists of ≈ 600 login events, each a tuple carrying the UNIX time of the attempted login, the IP address attempting the login, and a Boolean indicating if the login was successful.

For the first use case, we aim to identify sequences of failed logins followed by a successful login for each unique IP address in the sample dataset (≈ 90 distinct IPs). 100 sequences are matched, with lengths ranging from 2 to 57 tuples. For the second use case, we aim to identify 5 to 10 failed logins (from any IP) within 10 s. The number of matched sequences is 674, with sequence lengths ranging from 5 to 10 tuples.

For both experiments, we choose the *skipToNext* strategy (see Section 2.1). This choice reduces the computational cost required by looser skip strategies in which, e.g., every possible match is potentially emitted. All experiments run on a server equipped with an Intel(R) Core(TM) i7-4790 CPU running at 3.60 GHz. The system features 4 physical cores and 8 logical threads. The machine has 8 GB of memory and runs Java 21. We utilize JGEA [16] for the CFG-GP implementation and Apache Flink CEP [8] for the CEP engine. JGEA evaluates concurrently the fitness of candidate individuals in the same iteration of the EA: we set it to use all the 8 logical threads. We set a limit of $n_{\text{evals}} = 5\,000$ fitness evaluations as the stopping criterion for CFG-GP and we set $n_{\text{pop}} = 100$, $n_{\text{tour}} = 5$, and $p_{\text{xover}} = 0.8$ (widely adopted settings for GP). To avoid handling too large trees, we enforce a maximum depth $d_{\text{max}} = 16$ for the production trees built in the process.

We deploy Apache Flink CEP using Docker Compose, with separate Docker containers for the JobManager and TaskManager. Each TaskManager is configured with 16 Task Slots, ensuring efficient parallel processing of Flink CEP jobs (EA individuals). We allocated 2 GB of process memory for the JobManager and 5 GB for the TaskManager.

To minimize computational costs in the experiments, we capped each individual evaluation at 60 s. We determined this time limit based on the observed average execution time of Flink CEP jobs when evaluating different individuals, ensuring that only a negligible number of cases were excluded, thereby minimally impacting the EA behavior. The presence of CEP patterns exceeding the 60 s threshold is because, upon reaching the timeout, the Flink CEP client signals the Flink CEP cluster to cancel the job, but additional time is required to complete the cancellation once the request is issued.

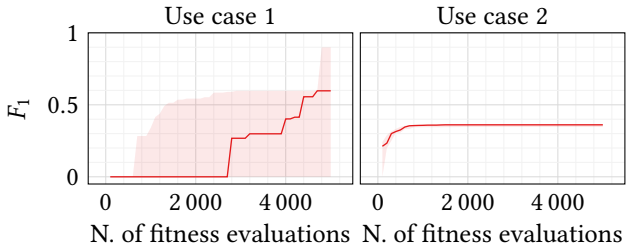

Figure 2: Fitness F_1 of the best individual in the population during the evolution for the two use cases: the line shows the median across 10 repetitions of the evolutionary optimization; the shaded area shows the first-third quartile range.

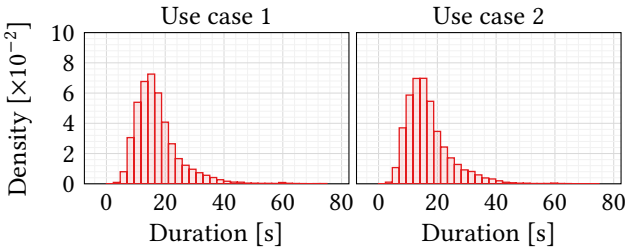

Figure 3: Distribution of the evaluation time for all the CEP applications generated during the 10 evolutionary runs for the two considered use cases.

4.2 Results and discussion

We run CFG-GP 10 times for each use case by varying EA's random seed. Figure 2 shows the fitness (i.e., F_1) of the best individual in the population (median/interquartile across repetitions) during the evolution. Figure 3 shows the distribution of the evaluation time for all individuals (i.e., applications). Results show that EC can be used to automate the discovery of CEP applications, though its current effectiveness is limited—some of the repetitions do not find any application with an F_1 greater than zero. Several challenges highlight open research directions in the EC for CEP context, though.

Long evaluation times for fitness assessment. As Figure 3 shows, evaluating the fitness of candidate applications is computationally expensive. We believe that this is due to (a) our non-trivial practical settings, where JGEA submits applications to a Flink CEP cluster not running in the same JVM, and (b) the potential high complexity of candidate applications. While the former cause might be mitigated by running the CEP engine closer to JGEA, this might only change the trade-off between deployment time (shorter, because network communication is faster) and running time (potentially longer, since the engine would run on a single machine). For the latter cause, several techniques could reduce the complexity of generated artifacts by, e.g., simplifying them during

or after the evolution [6], imposing a secondary objective other than just fitness [5], and enforcing simplicity directly in the search space [4]. Further research can thus study the benefits of these approaches in this context.

A key observation is that, for a candidate application, low fitness does not necessarily imply low computational cost. In early generations, individuals with poor fitness may define patterns that match most of the input data most of the time, leading to substantial computational overhead. It is thus important to implement early termination mechanisms. If certain patterns are not detected at specific positions within a stream, their evaluation could be, e.g., halted early. Also, initial fitness evaluations could run on portions of the sample dataset, expanding only when individuals exceed a predefined fitness value. Another strategy is threshold-based early termination to stop evaluations if the number of detected patterns exceeds a limit over a portion of the stream.

Efficient exploration of large search spaces. The search space for candidate applications is inherently large due to the extensive APIs provided by CEP systems, composable in multiple ways. In our setup, the search space is narrowed by automating the extraction of information from sample patterns. System experts can be prompted to specify constraints like: (a) Should all attributes be used, or can some be ignored? (b) Are constant-value attributes within a pattern possible key-by attributes? (c) Do overlapping sequences represent instances of the same pattern? An open research question remains: how can an optimal API subset be automatically selected (at least initially) to reduce the search space and boost evolutionary approaches' efficiency for CEP applications?

Another observation stems from the difference between the results of use case 1 and 2. While for the former the fitness continuously increases (in median) to good values, it soon stagnates for the latter to a rather low value for all the repetitions. We believe this might be due to the population falling in a local optimum, i.e., an application that is relatively good but far from the optimal one. Previous works exist where this problem has been addressed successfully: for example, in the context of the evolutionary inference of regular expressions, [2] proposes to split the problem of learning a single regex for many kinds of desired extractions into several simpler problems, each with more homogeneous extractions. We leave to future work the investigation about the use of this approach in the context of CEP applications.

5 Related work

To the best of our knowledge, no other studies investigate CEP applications discovery from data using EC and tested on CEP engines widely used in production systems. There are, however, other works for (partially) automatic discovery of applications and other usages of EC for inferring applications, or other artifacts (as, e.g., regular expressions) from data.

CEP applications discovery alternatives. While we rely on EC to evolve applications from sample event sequences, other studies explored, e.g., rule-based learning, machine learning classification, and deep learning-based pattern recognition.

Margara et al. [14] propose *iCEP*, a framework for automating application generation from historical event traces. *iCEP* refines constraints iteratively to extract event patterns using a modular approach to define time windows, event attributes, and sequence constraints. Unlike traditional machine learning, it models event traces as sets of constraints.

Mehdiyev et al. [17] explore machine learning techniques for automatic application extraction. They apply rule-based classifiers to identify event patterns and match complex event rules, leveraging historical data to improve accuracy and reduce manual effort in defining applications.

Liu et al. [13] present a framework that integrates a two-layer LSTM with an attention mechanism and a decision tree-based data mining algorithm, enabling the automatic extraction of meaningful applications from IoT data streams, with a specific application to air pollution forecasting.

Unlike our work, such approaches as well as others such as [9, 10, 12], while proposing relevant techniques, are not evaluated with an engine like Apache Flink CEP, which is widely used in production systems. Evaluating with production-level engines is crucial not only to ensure the applicability and usefulness of the results in practical settings but also to realistically assess the computational costs associated with the fitness evaluation of candidate solutions, which impacts the feasibility and scalability of such approaches in real-world deployments. The only exception is [13] which, however, only refers to the final deployment of a CEP application on Flink CEP without assessing its computational costs or detailing which Flink CEP APIs are being used.

EC for other applications generation. For their ability to optimize over non-trivial search spaces, EAs have been widely used to infer artifacts acting as rules, binary classifiers, or extractors from data, including regular expressions [3], signal temporal logic formulae [18], and access control policies [15].

In most cases, the space of solutions can be defined as a regular language, hence through a CFG. Also, the quality of candidate solutions is often assessed by applying them to some data and measuring how close their outcome is to a target outcome. For example, in [3], where the goal is to synthesize a regular expression that matches all and only the text snippets marked by the user, the quality of candidate solutions is assessed using the precision at the level of matches and the character level.

Our work shares several aspects of other studies applying EAs for inferring rules from data: the definition of the search

space through grammar and the use of understandable information retrieval indexes for driving the search. However, we remark that our search space is particularly expressive due to the large APIs of CEP systems. Also, as applying a CEP application may be computationally expensive, as it may match a large number of events, special care has to be taken when exploring the space of CEP applications with an EA. From this point of view, our scenario resembles that of signal temporal logic [1]. Indeed, the similarity of this framework to stream processing has been noted [4].

6 Conclusions and future work

Our initial study aims at simplifying the design of CEP applications for system experts with deep domain knowledge but lacking programming skills. We thus explore how EC can automate the discovery of CEP applications, with experts providing sample patterns from a dataset and automatically evolving candidate applications that detect similar patterns.

To the best of our knowledge, ours is the first study that investigates the combined use of EC and CEP with an empirical assessment based on Apache Flink CEP, a widely adopted production-level framework. Our initial results are promising and can stimulate research (1) towards improving feedback mechanisms based on active learning, or integrating human-in-the-loop feedback through grammar customization (pre-optimization) and direct marking of example sequences, and (2) towards further enhancing the scalability of fitness evaluation for candidate applications when jointly using EC and CEP frameworks in evolutionary processes.

Acknowledgments

Work supported by the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement number 101072456, Chalmers AoA Energy projects DEEP and INDEED, the Swedish Energy Agency (SESBC) project TANDEM, the Wallenberg AI, Autonomous Systems and Software Program and Wallenberg Initiative Materials for Sustainability project STRATIFIER.

References

[1] Ezio Bartocci, Cristinel Mateis, Eleonora Nesterini, and Dejan Nickovic. 2022. Survey on mining signal temporal logic specifications. *Information and Computation* 289 (2022), 104957.

[2] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2015. Learning text patterns using separate-and-conquer genetic programming. In *Genetic Programming: 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings 18*. Springer, 16–27.

[3] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1217–1230.

[4] Luca Bortolussi, Vincenzo Gulisano, Eric Medvet, and Dimitrios Palyvos-Giannas. 2019. Automatic translation of spatio-temporal logics to streaming-based monitoring applications for IoT-equipped autonomous agents. In *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things*. 7–12.

[5] Karina Brotto Rebuli, Mario Giacobini, Sara Silva, and Leonardo Vanneschi. 2023. A comparison of structural complexity metrics for explainable genetic programming. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*. 539–542.

[6] Lulu Cao, Zimo Zheng, Chenwen Ding, Jinkai Cai, and Min Jiang. 2023. Genetic programming symbolic regression with simplification-pruning operator for solving differential equations. In *International Conference on Neural Information Processing*. Springer, 287–298.

[7] Kenneth De Jong. 2017. Evolutionary computation: a unified approach. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 373–388.

[8] flinkcep [n.d.]. Apache Flink CEP. https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/libs/cep/ Accessed:2024-02-28.

[9] Lars George, Bruno Cadonna, and Matthias Weidlich. 2016. Il-miner: Instance-level discovery of complex event patterns. *Proceedings of the VLDB Endowment* 10, 1 (2016), 25–36.

[10] Sarah Kleest-Meißner, Rebecca Sattler, Markus L Schmid, Nicole Schweikardt, and Matthias Weidlich. 2023. Discovering Multi-Dimensional Subsequence Queries from Traces–From Theory to Practice. In *BTW 2023*. Gesellschaft für Informatik eV, 511–533.

[11] John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4 (1994), 87–112.

[12] Yan Li and Tingjian Ge. 2021. Imminence monitoring of critical events: A representation learning approach. In *Proceedings of the 2021 International Conference on Management of Data*. 1103–1115.

[13] Yuan Liu, Wangyang Yu, Cong Gao, and Minsi Chen. 2022. An Auto-Extraction Framework for CEP Rules Based on the Two-layer LSTM Attention Mechanism: A Case Study on City Air Pollution Forecasting. *Energies* 15, 5892 (2022). https://doi.org/10.3390/en15165892

[14] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. 2014. Learning From the Past: Automated Rule Generation for Complex Event Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 47–58. https://doi.org/10.1145/2611286.2611289

[15] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. 2015. Evolutionary inference of attribute-based access control policies. In *Evolutionary Multi-Criterion Optimization: 8th International Conference, EMO 2015, Guimarães, Portugal, March 29–April 1, 2015. Proceedings, Part I 8*. Springer, 351–365.

[16] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. 2022. JGEA: a modular java framework for experimenting with evolutionary computation. In *Proceedings of the genetic and evolutionary computation conference companion*. 2009–2018.

[17] Nijat Mehdiyev, Julian Krumeich, David Enke, Dirk Werth, and Peter Loos. 2015. Determination of Rule Patterns in Complex Event Processing Using Machine Learning Techniques. *Procedia Computer Science* 61 (2015), 395–401. https://doi.org/10.1016/j.procs.2015.09.168

[18] Federico Pigozzi, Laura Nenzi, and Eric Medvet. 2024. BUSTLE: a Versatile Tool for the Evolutionary Learning of STL Specifications from Data. *Evolutionary Computation* (2024), 1–24.

[19] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2022. A comprehensive survey on program synthesis with evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 27, 1 (2022), 82–97.

[20] Peter A Whigham. 1995. Inductive bias and genetic programming. (1995).