ASTD Patterns for Integrated Continuous Anomaly Detection In Data Logs

Chaymae El Jabri[1][0009−0002−7933−8874], Marc Frappier[1][0000−0002−4402−2514], and Pierre-Martin Tardif[1][0000−0002−7413−6897]

Université de Sherbrooke, Sherbrooke, Canada

Abstract. This paper investigates the use of the ASTD language for ensemble anomaly detection in a data logs. It uses a sliding window technique for continuous learning in data streams, coupled with updating learning models upon the completion of each window to maintain accurate detection and align with current data trends. It proposes ASTD patterns for combining learning models, especially in the context of unsupervised learning, which is commonly used for data streams. To facilitate this, a new ASTD operator is proposed, the Quantified Flow, which enables the seamless combination of learning models while ensuring that the specification remains concise. Our contribution is a specification pattern, highlighting the capacity of ASTDs to abstract and modularize anomaly detection systems. The ASTD language provides a unique approach to develop data flow anomaly detection systems, grounded in the combination of processes through the graphical representation of the language operators. This simplifies the design task for developers, who can focus primarily on defining the functional operations that constitute the system.

Keywords: ASTD · Anomaly detection · Continuous learning.

1 Introduction

In today's digital age, protecting IT infrastructure from cyberattacks and security breaches is critical for organizations to ensure daily operations, store sensitive data and manage customer information. Anomaly detection techniques can help organizations identify unusual patterns and behaviors in their systems so they can respond quickly and prevent potential security incidents. Anomaly detection techniques are instrumental in diverse areas, including fraud detection, network security, and intrusion detection within business applications [1].

Recognizing the pivotal role of anomaly detection systems in ensuring the security and reliability of various applications, from cybersecurity to industrial monitoring, it is crucial to acknowledge the challenges associated with their development [2]. Effectively addressing these challenges becomes imperative for successfully deploying robust and adaptive detection systems.

In the realm of anomaly detection systems, a formidable challenge arises from the dynamic nature of data patterns. To maintain the system's accuracy over

time, periodic model re-training becomes imperative. Nils Baumann et al. [3] underscore the critical importance of automating the re-training process to adapt to evolving data patterns seamlessly. This challenge necessitates implementing robust mechanisms that detect anomalies and autonomously refine their understanding of normal and abnormal behaviors in the ever-changing data landscape.

The intricacy of learning systems poses yet another significant challenge, encompassing multifaceted phases such as data pre-processing and model training. Benjamin Benni et al. [4] delve into a comprehensive analysis of this complexity, shedding light on the intricate processes that form the backbone of effective anomaly detection. Addressing this challenge requires the development of streamlined strategies to simplify the various phases, ensuring that the learning system can efficiently navigate the intricacies of data preprocessing and model training. Overcoming this hurdle is crucial for enhancing anomaly detection systems' overall effectiveness and efficiency.

As detection systems scale up to handle vast amounts of data, a distinct challenge emerges in maintaining modularity to ensure scalability and ease of maintenance. The development of large-scale detection systems demands a careful balance to prevent unwieldy complexity. Baldwin and Clark [5] stress the significance of modularity in such systems, emphasizing its pivotal role in facilitating scalability and simplifying maintenance efforts. Successfully addressing this challenge involves designing detection systems with modular architectures that can seamlessly adapt to the increasing demands of data volume and computational resources, ensuring both scalability and ease of long-term maintenance.

This article introduces a method for developing anomaly detection systems using a specification language called Algebraic State Transition Diagram (ASTD) [6]. It investigates the extent to which this language reduces the complexity of the detection system by adding an abstraction layer. Additionally, it examines how the graphical representation of the language's operators contributes to easing development efforts by managing the scheduling of various processes within the detection system. ASTD is a graphical and executable notation for composing state machines, offering modularity and flexibility in system development [7]. The paper's contributions include (1) The extension of the ASTD language by the Quantified Flow operator to allow the combination of an arbitrary number of models while keeping the specification compact, and (2) The definition of an ASTD specification that represents a pattern on which to base the development of more complex systems; this specification has the following features: - Automated re-training of learning models, - Composition of three learning models to detect anomalies in data logs, - Combination of the decisions of each model for each event. The intent is to provide an illustrative example of specifications of anomaly detection systems that can be easily adapted for other contexts or learning methods.

The paper is divided into six sections. In Section 2, we emphasize the importance of automating the renewal of the learning model in the context of dynamic data, the role of abstraction in reducing system complexity, and modularity, which facilitates maintenance and extension without introducing errors.

Section 3 introduces the Quantified Flow operator as an extension of the ASTD language to easily combine an arbitrary number of learning models. In Section 4, we present a case study on the detection of unexpected events, implementing the following essential features for unsupervised anomaly detection: - Automation of retraining of learning models using the Sliding Window technique. - Model composition using the Quantified Flow operator. - Decision combination of models through Majority Voting. In section 5, we assess the performance of the specification in detecting unexpected events during a day of activity, while highlighting the effect of training data renewal and the combination of unsupervised models. Finally, in Sections 6 and 7, we summarize our findings and conclude.

2 Related Work

The MLOps [18] approach presents a set of principles aimed at standardizing the deployment, management, and monitoring processes of machine learning models in production environments. This approach integrates best practices and tools to optimize the model lifecycle, ensuring their effectiveness and robustness throughout their usage. In this work, we propose a development framework for an unsupervised anomaly detection pipeline, based on statistical learning models. This framework aims to incorporate features that align with certain technical aspects of MLOps, such as:

1. **Periodic Learning**: Regular updating of data using a sliding window approach [8,9]. Gamma [8] suggests that in most cases, we are primarily concerned with computing statistics for the recent past rather than the entire history. The sliding window method is useful in this regard as it allows us to focus on the relevant data. There are various window models, including the sequence-based model where the window size is determined by the number of observations, and the timestamp-based model where the window size is determined by duration.
2. **Metadata Storage**: Storage of intermediate results associated with the learning model.
3. **Entity-Based Data Processing**: Separation and processing of data based on specific entities, such as users or machines, to customize analyses and anomaly detection.

By implementing these features, we aim to create a flexible pipeline that optimizes the development lifecycle of anomaly detection models.

The development of the Anomaly Detection System using ASTD language follows the Model-Driven Engineering (MDE) paradigm, which provides a potential solution to reduce complexities through abstraction [10]. MDE advocates for using software models at different levels of abstraction to (semi-) automatically construct software systems. Models serve as abstractions of complex entities; they conceal unwanted details so that modelers can easily focus on their areas of interest. The ASTD language ensures, through the graphical representation

of its operators, the combination and scheduling of processes, enabling a focus on the core operations of the detection system.

Modularity in software design is a crucial principle that significantly enhances maintainability and extensibility. By breaking down a complex system into smaller, independent modules or components, developers can isolate specific functionalities and dependencies within discrete units. This isolation simplifies the debugging process and facilitates maintenance, as changes or updates can be made to individual modules without affecting the entire system. Moreover, modular design fosters code reuse, allowing developers to replicate and adapt modules across different projects, saving time and effort. This approach is supported by studies such as [11] and [12], which emphasize the importance of modularity for building scalable and maintainable software systems. Overall, the use of modular design principles is a well-established strategy that enhances the robustness and longevity of software systems, making them more adaptable to evolving requirements and reducing the likelihood of introducing errors during maintenance or extension. The ASTD language is intrinsically modular. An ASTD specification consists of algebraic operators organized in a tree structure, where each branch defines a specific functionality of the system being modeled or developed.

3 Extended ASTD Formalism

ASTD is a specification language combining state-transition diagrams with process algebra operators. The syntax of ASTDs is defined using a type hierarchy. Characteristics shared by all ASTD types are defined in the abstract ASTD type $\mathsf{ASTD} \triangleq \langle n, P, V, A_{astd} \rangle$ where $n \in \mathsf{Name}$ is the name of the ASTD, P is a list of parameters, V is a list of attributes, $A_{astd} \in \mathrm{A}$ is an action. Parameters P are used to receive values passed by a calling ASTD; they can be read-only or read-write. Attributes V are state variables that can be modified by actions within the scope of the ASTD. Actions can also modify attributes received as parameters of ASTD. Type ASTD is extended by one recursive subtype for each ASTD operator.

Automaton is one of the ASTD types and it is very similar to the traditional automaton and hierarchical state machine, except its states can be of any ASTD type, thus enabling powerful combinations of behavior descriptions. The initial state of an automaton is denoted by ⟜○, and the final state by ◎. Each state has actions that are executed when entering the state, staying in the state, and exiting the state. An empty action is denoted by the symbol skip. Transitions are labeled by $\mathsf{e}(\boldsymbol{w})[\phi]/A_{tr}$, where $\mathsf{e}(\boldsymbol{w})$ is an event pattern with pattern expression list $\boldsymbol{w}$, ϕ is a guard that must hold for the transition to trigger and A_{tr} is an action. To trigger a transition, an event $e(v_1, \ldots, v_n)$ must match the event pattern $e(w_1, \ldots, w_n)$. The event pattern is mandatory on a transition while the guard and the action are optional and defaulted to true and skip respectively. A pattern expression w_i can be one of the following:

- The identifier of an ASTD parameter, attribute, or *quantification* variable associated with a quantified ASTD
- a wild card "_" that can match any value of v_i.
- $?x : T$: this declares a variable x of type T whose scope is only the transition.

This is a brief presentation of the ASTD language and its basic operator Automaton. One can find in [13,14] the definitions and semantics of all operators of the language: sequence, choice, Kleene closure, parameterized synchronization, flow, quantified choice, quantified synchronization, guard, and call. For the sake of concision, operators used will be illustrated by examples and informally described in the sequel of the paper.

The objective of this section is to present the extension of the ASTD language with the addition of the Quantified Flow operator $\langle \mathsf{Qflow} \rangle$, as well as the motivations behind this addition. The Quantified Flow was introduced to enable the language to support the combination of learning models, which consist of a training phase and a detection phase. To allow the invocation of methods for each model independently for these two phases, the addition of this operator was deemed necessary.

The existing synchronization operators are not suitable for this application. Quantified synchronization, for instance, allows for the parallel execution of its sub-instances and synchronizes their actions based on a set of events called Δ. The events in Δ are executed only once all sub-instances can perform them. Quantified interleaving, on the other hand, is a special case of quantified synchronization where the Δ set is empty ($\Delta = \varnothing$), allowing completely independent execution of the sub-instances.

The Quantified Flow, however, allows for an arbitrary number of instances to execute an event as soon as they can do so. The syntactic and semantic definitions of the Quantified Flow are as follows:

Syntax

The quantified flow ASTD subtype has the following structure:

$$\mathsf{Qflow} \triangleq \langle \Psi :, x, T, b \rangle$$

where $x \in \mathsf{Var}$ a quantified variable that can be accessed in read-only mode, T the type of x, and $b \in \mathsf{ASTD}$ the body of the flow. The state of a quantified flow is of type $\langle \Psi :_\circ, E, f \rangle$ where $\Psi :_\circ$ is the constructor, E the values of attributes and $f \in T \to \mathsf{State}$ is a function which associates a state of b to each element x of T, each state of b is an instance of the quantified flow. Initial and final states are defined as follows. Let a be a quantified flow ASTD.

$$\begin{aligned} \mathit{init}(a) &\triangleq (\Psi :_\circ, a.E_{\mathit{init}}, T \times \{\mathit{init}(a.b)\}) \\ \mathit{final}(a, (\Psi :_\circ, E, f)) &\triangleq \forall\, c : T \cdot \mathit{final}(a.b, f(c)) \end{aligned}$$

Semantics

Rule $\Psi :_1$ describes the execution of an event.

$$\Theta \triangleq \left(E_g = E_e \lhd\!\!\!- E \quad a.A_{astd}(E''_g, E'_g) \quad E'_e = E_e \lhd\!\!\!- (V \lhd E'_g) \quad E' = V \lhd E'_g\right)$$

$$\Psi :_1 \frac{\Omega_{qflow} \qquad \Theta}{(\Psi :_\circ, E, f) \xrightarrow{\sigma, E_e, E'_e}_a (\Psi :_\circ, E', f')}$$

where environments E_e, E'_e denote the before and after values of variables in the ASTDs enclosing ASTD a. The ASTD action A_{astd} defines the computation of E'_g from E''_g. E'_e and E' are extracted by partitioning E'_g using V. Premiss Θ is is used when a sub-ASTD transition is involved.

We use the following abbreviation to denote that an ASTD cannot execute a transition from a state s and global attributes E_g.

$$s \xrightarrow{\sigma, E_g}\!\!\!\!\!\!/\;\;_a \triangleq \neg \exists\, E'_g, s' \cdot s \xrightarrow{\sigma, E_g, E'_g}_a s'$$

Premiss Ω_{qflow} non-deterministically picks a permutation p of T (noted $p \in \pi(T)$) and Es, a sequence of environments storing the intermediate results of the computation of E''_g from E_g by iterating over the elements $p(i)$ of p, and executing the instances of the quantified flow. Thus, the execution order of the instances is non-deterministically chosen. If the specifier wants to have a deterministic result for the value of attributes, then he/she must ensure that the actions of the instances are commutative. Let $k = |T|$.

$$\Omega_{qflow} \triangleq \left(\begin{array}{l} p \in \pi(T) \land Es \in 0..k \to \mathsf{Env} \land Es(0) = E_g \land Es(k) = E''_g \land \\ \forall\, i \in 1..k \cdot \left(\begin{array}{l} f(p(i)) \xrightarrow{\sigma, Es(i-1) \lhd\!\!\!- \{x \mapsto p(i)\}, Es(i)}_{a.b} f'(p(i)) \\ \lor \\ Es(i) = Es(i-1) \land f(p(i)) \xrightarrow{\sigma, Es(i)}\!\!\!\!\!\!/\;\;_{a.b} \end{array} \right) \end{array} \right)$$

The Quantified Flow is a generalization of the flow operator, which is an operator similar to the AND state in statecharts. In [15], the flow operator is used for combining the processes of data pre-processing, training, and detection during the development of anomaly detection systems by ASTDs. This is possible thanks to the fact that a single input event can be processed differently in each of the sub-ASTDs of the flow operator according to two distinct actions. This is represented in Figure 1:When event $\mathsf{e()}$ is received, $\mathsf{act_1}$ is executed, followed by $\mathsf{act_2}$.

Anomaly detection models offer methods responsible for training and predicting scores. This allows us to define an abstract structure that represents their general functions. In object-oriented programming, this is achieved by defining the skeleton of the detection models in an abstract class and implementing its methods in specific ways at the level of sub-classes (see Fig. 2).

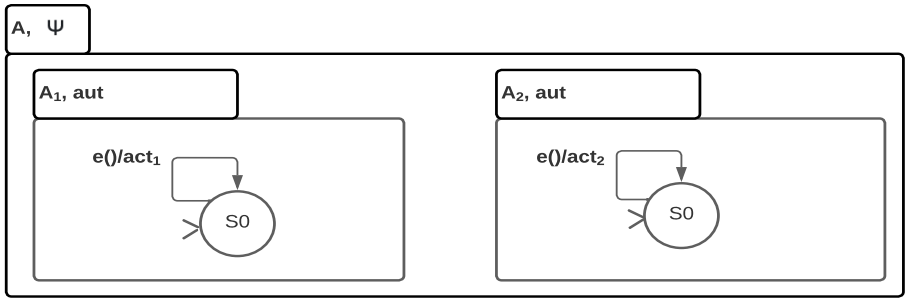

Fig. 1. Flow operator

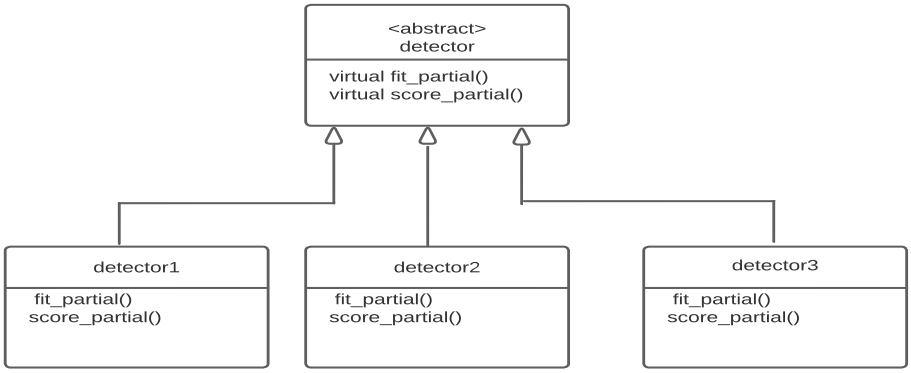

Fig. 2. Class hierarchy for anomaly detection using multiple techniques

In order to combine a set of heterogeneous detectors while maintaining the abstraction of their functioning in an ASTD, we define the quantified flow operator, a generalization of the flow operator over a quantification set. We use the abstract class *detector*, as shown in Figure 2, which contains two methods: fit_partial(), which trains the model sequentially by adding each received instance, and score_partial(), which returns the score of the input instance according to the reference model.

In Figure 3, which defines the specification for combining detectors, we define an attribute associated with the quantified flow A named *detectors* of type map<string, detector*>, which associates each detector name with an instance declaration of its class: '$detector_1$': new detector1(), '$detector_2$': new detector2(), '$detector_3$': new detector3().

Upon receiving an event e() the quantification variable d will traverse the quantification set {$detector_1$, $detector_2$, $detector_3$} and execute the fit_partial and score_partial actions of each model. Consequently, we combine the different detectors while ensuring the abstraction of their functions. This example illustrates that the hierarchical nature of ASTDs integrates with class hierarchies.

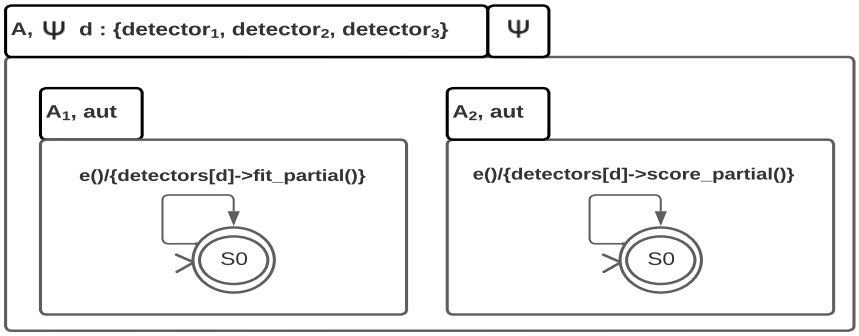

Fig. 3. An ASTD pattern for combining three anomaly detection models using the quantified flow

The abstract class detector is used in the declaration of the map detectors in the quantified ASTD A, and each instance of A can use its own specific type of detector.

To generalize to any set of detectors, we must first define classes that inherit from the *detector* interface, implementing the fit_partial() and score_partial() methods. Next, pass a JSON configuration file as a parameter to the specification, containing the list of detectors names and the constructors initializing each detector. This will enable dynamic loading of the set of detectors.

ASTDs are supported by the tools eASTD and cASTD [16]. eASTD is a graphical editor of ASTD specifications. cASTD is a compiler that translates ASTD specifications into executable code. It first generates an implementation in an abstract, intermediate, imperative language that can be translated into an equivalent executable imperative language like C++, Java, Python. Currently, C++ is the sole translation implemented. The generated code can read the data continuously from a data source, and apply the operations contained in the specification in the order that has been defined thanks to the process algebra operators.

4 ASTD Specification For Combining Anomaly Detection Models

In this section, we present a generic ASTD specification for combining a set of heterogeneous detectors. For this purpose, we introduce a real application case, in which we will determine all the components and elements of the specification. The complete specification is found in [17]. The main goal of this application is to identify unusual or unexpected events within user activities. These "unexpected events" typically manifest as activities occurring at times when a user is not usually active. Our example is based on the time of occurrence of an event,

for illustrative purposes and the sake of simplicity. Other criteria, or more general techniques for identifying anomalies, could easily be used with our ASTD pattern. For our example, we select three attributes from those available in the log files which are:

- Id: uniquely identifies each event, designated in the specification by eventId.
- CreationTime: determines the date and time in Coordinated Universal Time (UTC) when the user performed the activity, designated in the specification by eventDate.
- UserId: the user who performed the action

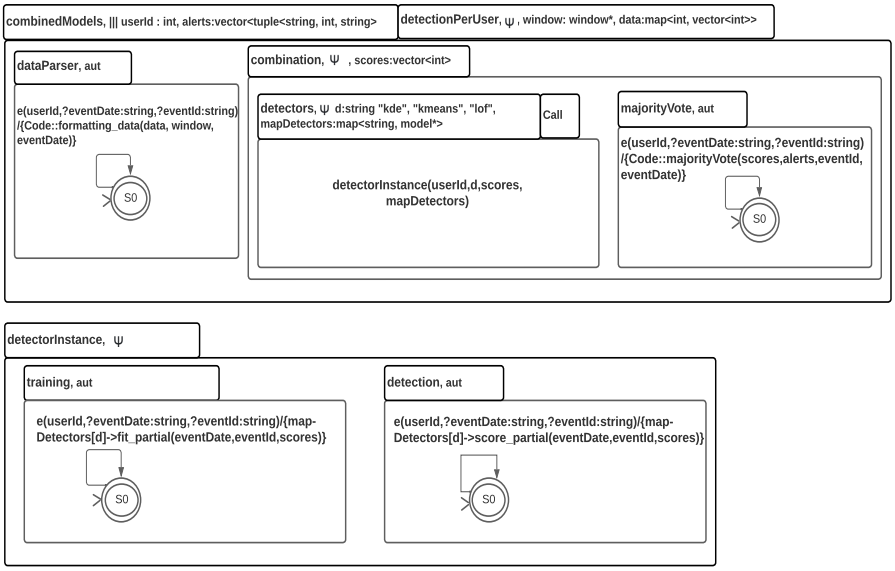

Fig. 4. Specification pattern

We utilize three heterogeneous learning models, which are:

- K-means: a clustering algorithm for batch learning adapted to circular data. The number of clusters used is optimized using the silhouette coefficient. The distance used for clustering refers to the time interval between two events occurring at different hours, denoted as a and b. The formula to compute this distance is as follows:

$$\text{distance}(a,b) = \begin{cases} \min(b-a, a-b+24), & \text{if } a < b \\ \min(a-b, b-a+24), & \text{otherwise} \end{cases} \quad (1)$$

- KDE: is a non-parametric statistical technique for estimating the probability density of a random variable.

- LOF: is an unsupervised anomaly detection method that calculates the local density deviation of a given data point from its neighbors. It considers as outliers the samples whose density is significantly lower than that of their neighbors.

The data used during the training of the different models consists of the hours of events performed by a user within a day, which means the data is circular in an interval of [0,24[.

In Figure 4, we present the top-level ASTD named combinedModels of type quantified interleave, denoted by ||| in the upper left tab. It declares a quantification variable userId of type int with an $UnboundedDomain$, which allows the processing of all the users received without the need to identify them a priori. The quantified interleave allows each user to be treated independently by associating an instance of the flow sub-ASTD for each user. The flow combines two sub-ASTDs dataParser, and combination

It has the following attributes:

- window of type window* initialized by new window(window_parameters).
- data of type $map\langle int, vector\langle int\rangle\rangle$. In cases where the period type is either 'week' or 'day', the keys of the map represent the period number. However, if the type of window is 'instance', the map contains a single key with a value of '0'. The values in the map are the minutes of the occurrence of the events, stored in a vector.
- alerts, which is of type $vector\langle tuple\langle string, int, string\rangle\rangle$. The alerts attribute stores information about abnormal events, including the event identifier, the number of models that flagged the event, and the date of its occurrence.

Additionally, the following parameters are defined:

- window_parameters of type json that respects the following structure $\{window\_size : int, sliding\_size : int, type \in \{'day','week','instance'\}\}$
- kde_parameter: double; the value of the k-percentile that will determine the threshold of probability densities below which an event is considered abnormal. It takes values in [0.5, 5].
- kmeans_parameter: double; the threshold to which the absolute value of the events cluster's z-score is compared. It takes values in [1.5, 2.5].
- lof_parameter: double; the value of the k-percentile that will determine the threshold of LOF scores for the training data, above which any score from the test data is considered abnormal. It takes values in [75, 95].

The sub-ASTD named detectionPerUser is of type flow denoted by Ψ :, which is a binary operator. It allows the same event to be treated by its two sub-ASTDs, and the combination of the latter two by sharing inherited variables. The right sub-ASTD is DataParser of automaton type, consisting of a single initial and final state (S0) having a loop transition labeled with the event pattern e(userId, ?eventDate: string, ?eventId: string) and the action formatting_data(data, window, eventDate), which adds each received event to the training set and defines the data belonging to the current window according to the type of period chosen

as shown in the algorithm 1, The methods *add_instance* and *add_period* of the window class can be found in Appendix A.

Algorithm 1 formatting_data

```
1: Input: data, window, eventDate
2: Output: data, window updated
3: int hour = get_hour(CreationDate)
4: string type = window → getType()
5: if type == "day" or type == "week" then
6:     period = Compute_period(CreationDate, type)
7:     add minute to data[period]
8:     std::vector⟨int⟩periodsToDelete = window → add_period(period)
9:     delete the periods in periodsToDelete from the map data
10: if type == "instance" then
11:     add minute to data[0]
12:     bool sliding_on = window → add_instance(minute)
13:     int sliding_size = window → getSliding_size()
14:     if sliding_on then
15:         data[0] ← delete elements from data[0] from start to sliding_size
```

The left sub-ASTD is named Combination it is a flow with the parameter: scores of type $vector\langle int\rangle$. It stores the scores of value 0 or 1 of an input data for each detection model. At the level of the left ASTD referred to as detectors, we establish an attribute called $mapDetectors$, which is of type $map\langle string, model*\rangle$. Here, the term "model" represents an abstract class from which three distinct learning models inherit: namely, k-means, kernel density estimation (KDE), and the local outlier factor (LOF). mapDetectors is initialized using the function $init\_map(kmeans\_parameters, kde\_parameters,$
$lof\_parameters)$ (Algorithm 2).

Algorithm 2 Initialize Map of Models

```
1: function INIT_MAP(kmeans_parameters, kde_pa − rameters, lo − f_parameters)
2:     Map⟨String, Model*⟩ map_classes
3:     map_classes["kde"] ← new kde(kde_parameters)
4:     map_classes["kmeans"] ← new kmeans(kmeans_parameters)
5:     map_classes["lof"] ← new lof(lof_parameters)
6:     return map_classes
```

ASTD detectors respects the structure presented in Section 3, except that in order to modularise the specification we use the operator Call, which calls the ASTD DetectorInstance containing the actions allowing the training and the detection by each model. It has two sub-ASTDs training and detection which are of type automaton having both a single state which is initial and final with a

loop transition labeled by the same event e(userId, ?eventDate: string, ?eventId: string) and with the following actions :

- $mapDetectors[d] \rightarrow fit\_partial(data)$ at the training ASTD which launches the computation of the three learning models each time there is enough data in the current window.
- $mapDetectors[d] \rightarrow score\_partial(eventDate, event_Id, scores)$ in the detection ASTD , which populates the scores vector with predictions from each model, while adhering to the discrimination criteria set for each of the models.

After having obtained the score of each model, we perform a Majority Voting in the majorityVote ASTD by the action Code::majorityVote(scores, alerts, eventId,eventDate), which scans scores and in the case that more than 50% are positive (of value 1), it adds the event data in alerts, as shown in the algorithm 3.

Algorithm 3 majorityVote: Majority Vote Algorithm

```
1: procedure MajorityVote(scores, alerts, eventId, eventDate)
2:     if scores.size() ≠ 0 then
3:         count ← 0
4:         for i ← 0 to scores.size() − 1 do
5:             if scores[i] = 1 then
6:                 count ← count + 1
7:         if count > ⌊labels.size()/2⌋ then
8:             alerts.push_back(⟨eventId, count, eventDate⟩)
9:             print eventId is malicious
10:        scores.clear()
```

To apply these models to data streams they are integrated in a sliding window. We have defined three distinct types of windows, to capture relevant information for anomaly detection in various applications, each with varying window sizes. Specifically, we have timestamp-based windows that are categorized based on the number of days or weeks, where each event is associated with a unique day or week number defined by YYYYDDD or YYYYWW, respectively; where YYYY denotes the year, DDD denotes the day's number, and WW denotes the week's number. We refer to these values as *periods*. Additionally, we have a sequence-based window type whose size is determined by the number of events. In all cases, the window's initialization involves the following three parameters:

- *window_size*: the number of days, weeks, or events the window covers.
- *sliding_size*: the number of days, weeks, or events the window moves.
- *type*: 'day', 'week', or 'instance'.

Window sliding is shown in figure 5, and depends on two parameters: *window_size* (ws) and *sliding_size* (ss). *Window_size* consists of three units, representing the window size, and *sliding_size* consists of one unit, which determines the

number of units by which the window moves; data associated with old units is deleted. The window moves when we obtain the necessary data to complete the *sliding_size*, at which point we update the window and delete the data from the previous window's old units

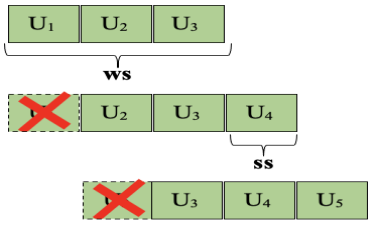

Fig. 5. Sliding window

The training and detection by each model occurs as follows:

- K-means: Throughout the training process, our objective is to identify the clusters within the data of the current time window. We optimize the number of clusters, denoted as 'k', by evaluating the silhouette coefficient, a measure of cluster quality. In addition to identifying the clusters, we also compute and store the standard deviation and mean values for each cluster. During the detection phase, our system identifies anomalies by a two-step process. First, we determine the cluster that is closest to the input minute, and then we calculate the z-score. If the computed z-score exceeds a threshold defined by 'kmeans_parameter', we classify it as an anomaly.
- KDE: The training involves modeling the probability density of a user's activity over the 24 hours of the day based on the data contained in the current window. The percentile of the probability densities of the training data is calculated according to kde_parameter, which represents the detection threshold. Then, when a new event is received, the time of its occurrence is calculated. If the probability density associated with this time is below the threshold, the event is assigned a value of 1, indicating that the event is considered an anomaly.
- LOF: We use the algorithm from the sklearn library, choosing cosine as the metric. Before providing the data to the training model, we convert it into Cartesian space to ensure compatibility with the chosen metric. The percentile of the LOF scores of the training data is calculated according to lof_parameter, which represents the detection threshold. When a new event is received, we compare its score with the threshold. If the score exceeds the threshold, the data is considered abnormal and is assigned a value of 1.

5 Experiment

The initial goal of this case study was to detect user activities occurring at unusual times within the activity logs of various Microsoft Office 365 services [15]. However, since there is no ground truth available for this type of application, we will apply the case study to a dataset from CERT Insider Threat version 4.2 [19], which simulates the activity of 1,000 employees, 70 of whom are malicious according to three malicious scenarios. The dataset that will be used is logon.csv, which contains user IDs, logon and logoff dates, and the PC on which the activity was performed. We will focus on the anomalies associated with the first scenario, which identifies users who logged in after working hours to upload data to wikileaks.org.

Although we will concentrate on a subset of the available information, our main interest in this application lies in the detection rate. We will also examine the effect of combining models through majority voting and the impact of the data renewal method.

First, we convert each line of the logon.csv file into an event in the form of e(userId, date, eventId), and then we provide these as input to the executable C++ code that translates the ASTD specification.

The evaluation is performed using the ROC (Receiver Operating Characteristic) curve and AUC (Area Under the Curve) metrics. The ROC curve represents the relationship between the detection rate (DR) and the false positive rate (FPR), while the AUC summarizes the ROC curve into a single numerical value, allowing for model comparison.

We also calculate the detection rate (DR) for different models. The detection rate (DR) is defined as follows:

$$DR = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

The table 1, highlights the performance metrics of different anomaly detection configurations, revealing notable improvements when combining KDE, LOF, and KMeans models using a majority voting method. Across all scenarios, the combined models show enhanced detection rates (DR) and AUROC, indicating that leveraging the strengths of multiple models results in more robust and accurate anomaly detection. This ensemble approach reduces the likelihood of missing true anomalies while maintaining a high overall performance.

Comparing the different window configurations, we observe that the first case (10,5,week), with a window size of 10 weeks, a sliding size of 5 weeks, performs the best. This setup achieves the highest DR and AUROC values across all models, demonstrating its effectiveness in detecting anomalies. The large number of alerts generated in this configuration indicates the model's high sensitivity. In contrast, the second case (10,0,week), which lacks a sliding window, shows significantly lower performance metrics. The absence of overlapping windows reduces the model's ability to renew data, leading to decreased detection rates and AUROC values, although it generates more alerts, potentially increasing false positives. The third case (100,50,instance) uses an instance-based sliding

Thresholds (kde, lof, kmeans)		(1.5, 0.5, 95)			
window (window_size, sliding_size, type)		kde	lof	kmeans	combined models
(10,5,week)	DR	0.89	0.89	0.94	0.92
	AUROC	0.83	0.89	0.83	0.89
	Number of alerts	109060			
(10,0,week)	DR	0.73	0.19	0.54	0.48
	AUROC	0.74	0.51	0.58	0.63
	Number of alerts	175995			
(100,50,instance)	DR	0.60	0.56	0.84	0.76
	AUROC	0.73	0.71	0.8	0.81
	Number of alerts	100948			

Table 1. Performance metrics for an anomaly detection configuration using KDE, LOF, and KMeans models with different window settings.

window, resulting in moderate performance. While the combined models still outperform individual ones, the overall DR and AUROC are lower than in the first case, and the number of alerts is the lowest, suggesting fewer false positives but a risk of missing true anomalies.

The improvement in performance with a sliding window underscores the importance of data renewal. Overlapping windows ensure that the model continuously updates its data, enhancing its capacity to detect anomalies accurately. This continuous data renewal is crucial for maintaining high detection rates and AUROC values, as it allows the model to adapt to new patterns and anomalies in the data. Furthermore, the ASTD specification ensures uniform data size across all users, whether by the number of instances or duration during training. This uniformity guarantees consistent performance across different users, preventing bias towards any particular user's data size and resulting in fairer and more reliable anomaly detection.

Finally, the choice of parameters such as window size, sliding size, and type plays a significant role in the number of detected alerts. Balancing between true anomaly detection and minimizing false alerts is critical to optimizing the model's performance. Selecting optimal parameters reduces errors and enhances the effectiveness of the anomaly detection system, ensuring that true anomalies are identified while minimizing the occurrence of false positives. This careful parameter selection is essential for achieving accurate and reliable anomaly detection in various scenarios.

6 Discussion

The ASTD specification in Section 4 uses the quantified interleave operator, which provides processing independence for each user and allows separation of

the variables common to all users from those associated with each user. The common variables are defined at the level of the quantified interleave, while the user-specific variables are declared below the quantified interleave in the specification hierarchy. These variables can be accessed by their names in the specification, and the cASTD compiler forwards them to the associated userId instance. Data renewal is performed using a sliding window approach, which requires three specified parameters: *window_size*, *sliding_size*, and *type*. The management of data and launching of recomputation of learning models for each user are dependent on these parameters.

The ASTD specification in Section 4 illustrates the utility of the Quantified Flow operator, which enables the execution of the three learning models while maintaining the operation of each abstract. The combination of learning models is achieved through Majority Voting, but other combination techniques can be employed by modifying the action at the left sub-ASTD of the ASTD combination.

The ASTD language provides a framework for better structuring the code by its operators. It enables us to determine the different components of the system, in our case: user, learning models, and window, as well as their interrelationships. This promotes the adoption of a robust development approach. However, the C++ language, which is employed at the level of actions in an ASTD specification, does not currently provide a comprehensive set of machine-learning libraries. This limitation could be addressed by integrating Python code that handles these tasks.

It's worth noting that the ASTD specification presented here is not limited to the specific anomaly detection methods described. Instead, it can be easily adapted to accommodate various other anomaly detection techniques by simply modifying the ASTD components specific to the chosen method. This flexibility underscores the generative power of the ASTD language.

Additionally, the object-oriented architecture of the classes representing the learning models plays a pivotal role in abstracting the specific behavior of each model. This architectural choice harmonizes seamlessly with the ASTD framework within a Quantified Flow. As such, our contribution extends beyond a practical implementation and serves as a specification pattern, emphasizing the language's capacity to abstract and modularize complex systems.

The ASTD language, through its visual approach, provides a detailed view of the various stages of the pipeline, thus allowing for a better understanding and maintenance of the system. This becomes more apparent when using the eASTD editor of the language, where for each component of the specification, one can see its various properties and also assign comments describing its function in the overall system.

One of the major properties of the ASTD language lies in the modularity it brings to the development of detection systems. This modular approach not only facilitates the initial development of the system but also its subsequent evolution. A designer can make targeted modifications without compromising the overall integrity of the system.

Another important aspect of the ASTD language lies in the scheduling of the features of the detection system. The language's operators play a central role in this task, enabling smooth and efficient process management. By entrusting scheduling to these operators, the ASTD language significantly reduces development effort. Designers can focus on business logic, leaving operators to handle the coordination of different stages of the system.

The drawback of this method lies in the fact that it requires an understanding of the functioning of each of the ASTD language operators. Indeed, although the clear visualization and modularity offered by the language facilitate system design and maintenance, dependence on operators can pose a challenge for developers less familiar with them.

Note that the purpose of this experiment is not to evaluate the accuracy of the produced detection model. This is a separate issue that is orthogonal to the objective of this paper, which is to streamline the construction of models. In unsupervised learning of anomalies, evaluation requires the intervention of security experts who evaluate each alert individually and requires the setting of an appropriate threshold that returns a «manageable» number of alerts given the human analysis resources available. Ground truth is expensive to obtain in this domain, and realistic data sets are hard to obtain.

7 Conclusion

In this study, we use the ASTD language for anomaly detection in data logs. Our focus centered on the sliding window technique for continuous learning in data streams, coupled with updating learning models upon the completion of each window to maintain accurate detection and align with current data trends. Additionally, we emphasized the significance of employing methods for combining learning models, especially in the context of unsupervised learning, which is commonly used for data streams. To facilitate this, we extended the ASTD language with a new operator, the Quantified Flow, which enables the seamless combination of learning models while preserving the functioning of each of them in an abstract manner. Therefore, our contribution extends beyond a mere implementation and serves as a specification pattern, highlighting the language's capacity to abstract and modularize anomaly detection systems. In conclusion, the ASTD language provides a unique approach to developing data flow anomaly detection systems, grounded in the combination of processes through the graphical representation of the language operators. This simplifies the design task for developers, who can focus primarily on defining the functional operations that constitute the system.

References

1. Ahmed M, Mahmood AN, Islam MR. A survey of anomaly detection techniques in financial domain. Future Generation Computer Systems. 2016 Feb 1;55:278-88.

2. Yao D, Shu X, Cheng L, Stolfo SJ, Bertino E, Sandhu R. Anomaly detection as a service: challenges, advances, and opportunities. Morgan & Claypool; 2018.
3. Baumann N, Kusmenko E, Ritz J, Rumpe B, Weber MB. Dynamic data management for continuous retraining. InProceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings 2022 Oct 23 (pp. 359-366).
4. Benni B, Blay-Fornarino M, Mosser S, Precisio F, Jungbluth G. When DevOps meets meta-learning: A portfolio to rule them all. In2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) 2019 Sep 15 (pp. 605-612). IEEE.
5. Baldwin CY, Clark KB. Design rules: The power of modularity. MIT press; 2000.
6. Frappier M, Gervais F, Laleau R, Fraikin B, St-Denis R. Extending statecharts with process algebra operators. Innovations in Systems and Software Engineering. 2008 Oct;4:285-92.
7. Tidjon LN, Frappier M, Mammar A. Intrusion detection using ASTDs. InAdvanced Information Networking and Applications: Proceedings of the 34th International Conference on Advanced Information Networking and Applications (AINA-2020) 2020 (pp. 1397-1411). Springer International Publishing.
8. Gama J. Knowledge discovery from data streams. CRC Press; 2010 May 25.
9. Jankov D, Sikdar S, Mukherjee R, Teymourian K, Jermaine C. Real-time high performance anomaly detection over data streams: Grand challenge. InProceedings of the 11th ACM international conference on distributed and event-based systems 2017 Jun 8 (pp. 292-297).
10. Moin A, Challenger M, Badii A, Günnemann S. A model-driven approach to machine learning and software modeling for the IoT: Generating full source code for smart Internet of Things (IoT) services and cyber-physical systems (CPS). Software and Systems Modeling. 2022 Jun;21(3):987-1014.
11. Van Vliet H, Van Vliet H, Van Vliet JC. Software engineering: principles and practice. Hoboken, NJ: John Wiley & Sons; 2008 Jul 31.
12. Martin RC. Clean Craftsmanship: Disciplines, Standards, and Ethics. Addison-Wesley Professional; 2021 Sep 16.
13. Frappier, Marc, et al. "Extending statecharts with process algebra operators." Innovations in Systems and Software Engineering 4 (2008): 285-292.
14. Tidjon, Lionel Nganyewou, et al. "Extended algebraic state-transition diagrams." 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE, 2018.
15. Chaymae, El Jabri, et al. "Development of monitoring systems for anomaly detection using ASTD specifications." International Symposium on Theoretical Aspects of Software Engineering. Cham: Springer International Publishing, 2022.
16. GRIF. 2023. ASTD Tools. https://github.com/eljabrichaymae/ASTD-tools.git
17. Chaymae El Jabri. 2023. ASTD Patterns for Integrated Continuous Detection, Corre- lation and Response of Cybersecurity Incidents. `https://github.com/eljabrichaymae/Case_Study-ASTD_Patterns-.git`
18. Kreuzberger, Dominik, Niklas Kühl, and Sebastian Hirschl. "Machine learning operations (mlops): Overview, definition, and architecture." IEEE access 11 (2023): 31866-31879.
19. CERT and ExactData, LLC. Insider Threat Test Dataset. Accessed: Jul. 8, 2024. [Online]. Available: `https://kilthub.cmu.edu/articles/dataset/Insider_Threat_Test_Dataset/12841247`

A Algorithms of the specification

Algorithm 4 add_period

Input: *period*
Output: The periods to be deleted in case the window moved, *seen_periods*, *ref_periods*, and *is_training_on* updated
if $this \rightarrow is\_trainig\_on$ **then**
    $this \rightarrow num\_training++$
    $this \rightarrow is\_training\_on = false$
$vector\langle int\rangle$ *periods_to_delete*
if *period* not in $this \rightarrow seen\_periods$ and *period* not in $this \rightarrow ref\_periods$ **then**
    $this \rightarrow seen\_periods.push\_back(period)$
    if ($this \rightarrow seen\_periods.size()$ - $this \rightarrow window\_size$ -1) % $sliding\_size == 0$ **then**
        int index = $\max(0, this \rightarrow seen\_periods.size() - 1 - this \rightarrow window\_size)$
        $int\ index_1 = max(0, (int)index - this \rightarrow sliding\_size)$
        $periods\_to\_delete$ = vector⟨int⟩($this \rightarrow seen\_periods$.begin() + $index_1, this \rightarrow seen\_periods$.begin() + index)
        $this \rightarrow ref\_periods = vector\langle int\rangle\ (this \rightarrow seen\_periods.begin() + index, this \rightarrow seen\_periods.end() - 1)$
        if $this \rightarrow ref\_periods.size() == this \rightarrow window\_size$ **then**
            $this \rightarrow is\_training\_on = true$
return periods_to_delete

Algorithm 5 add_instance

```
1: Input: minute
2: Output: boolean renew_data indicates whether the training data needs to be
   renewed
3: if this → is_trainig_on then
4:     this → num_training + +
5:     this → is_training_on = false
6: this → seen_instances + +
7: bool renew_data = false
8: if this → seen_instances <= this → window_size then
9:     if this → seen_instances == this → window_size then
10:        this → is_training_on = true
11:        renew_data = false
12: else
13:    if (this → seen_periods.size() - this → window_size ) % sliding_size then
14:        this → is_training_on = true
15:        renew_data = true
16: return renew_data
```