AN ADAPTABLE IOT RULE ENGINE FRAMEWORK FOR DATAFLOW MONITORING AND CONTROL STRATEGIES

Ken Chen
School of Software and Microelectronics
Peking University
aken5930@outlook.com

ABSTRACT

The management of data generated by high-volume devices in Internet of Things (IoT) systems has been a topic of ongoing discussion. Several studies have explored the use of generic rule engine, primarily based on the RETE algorithm, for monitoring the flow of device data. Some studies have also proposed improved RETE algorithm in IoT filed. However, implementing modifications to the general rule engine remains challenges in practical applications. The open-source platform, Thingsboard, introduces an IoT-specific rule engine that does not rely on the RETE algorithm. Its interactive mode attracted attention from developers and researchers. However, the close integration between its rule module and the platform, as well as the difficulty in formulating rules for multiple devices, limits its flexibility. This paper presents an adaptable and user-friendly rule engine framework for monitoring and control IoT device data streams. The framework is easily extensible and allows for the formulation of rules contain multiple devices. We designed a Domain-Specific Language (DSL) for user interaction. A prototype system of this framework was implemented to verify the validity of theoretical method. Analysis suggests that this framework has potential to be adaptable to a wide range of IoT scenarios and is especially effective in where real-time control demands are not as strict.

Keywords IoT, Rule Engine, Dataflow Monitoring and Control

1 Introduction

The extensive volume of data generated by Internet of Things (IoT) devices, along with the diverse needs of users, highlights the necessity for customized monitoring of device data streams. In the past few years, researchers predominantly relied on general rule engines, primarily based on the RETE[1] algorithm, to address these challenges[2–5]. Rule engine is the software system that separates rule definitions from their implementation. Typically, rule consists of IF-THEN statements, where the action in the THEN statement is executed if the condition in the IF statement is met. Rule engine interprets these rule statements and integrates them into the software system. The RETE algorithm, which provides an efficient method for matching rules with data, is currently the most widely used approach for constructing rule engines.

The RETE algorithm optimizes the calculation of matching results by caching intermediate operation results, thereby reducing the time complexity of rule matching. This allows for the reuse of previous calculation results if a certain input attribute remains unchanged in the next batch of data. However, in IoT scenarios, device messages change rapidly, resulting in a low reuse rate of intermediate results. Thus, the space-for-time strategy often proves inadequate. To address this issue, researchers have proposed improved versions of the RETE algorithm to enhance its applicability in IoT scenarios[6–8]. Nevertheless, replacing the RETE algorithm is challenging in practice due to the complex internal implementation of general rule engines, such as Drools, which are designed to adapt to various application scenarios.

General rule engines are not the only solution for customized data stream monitoring. The open-source IoT data platform, Thingsboard[9], deviates from the RETE algorithm in its rule engine implementation and proposes an alternative rule model for streaming data processing. The Thingsboard rule engine offers a more user-friendly interactive model compared to general rule engines and has found wider application in various IoT scenarios. However, the strong

coupling between the components of Thingsboard makes it challenging to apply its rule engine individually in other systems. Additionally, the Thingsboard rule engine does not support the monitoring of data flows from multiple devices within a single rule.

Both the aforementioned primarily applied solutions have potential limitations. General rule engines based on the RETE algorithm are not well-suited for handling continuously changing data sets. The Thingsboard rule engine, while feature-rich, suffers from poor portability.

To address these issues, this paper introduces an adaptable and easily extendable IoT-specific rule engine framework. The framework supports multi-device monitoring. And we defined a basic Domain-Specific Language (DSL) for rule description. This framework can be seamlessly integrated into almost any structured IoT system.It is primarily used to modify device data flow monitoring rules during the run-time of the IoT system.

We implemented a prototype system based on proposed framework, conducted experiments using the REST API, and validated the theoretical approach through a series of test cases. This paper discusses the framework's effectiveness, scalability, limitations, and application scenarios.

This paper begins by presenting the outcomes and limitations of general rule engines and Thingsboard rule engine, subsequently establishing the research objectives.

Subsequent chapters propose a novel IoT rule engine framework and the DSL for rule description. The design of rule DSL is presented, followed by an outline of the flow of the rule matching function in accordance with its syntax. The paper ultimately provides the algorithm for parsing rule text and the methodology for rule scheduling.

The latter part of the paper details the implementation of the prototype system, test cases, and results.

At last, a comprehensive discussion and summary of the proposed framework conclude the paper.

2 Related Works

The RETE algorithm is widely used in general rule engines for pattern matching. Pattern matching means matching an object, referred to as a fact, with a rule. This algorithm optimizes the computational load by compiling user-defined rules into a network structure known as the RETE network.

The RETE network incorporates the design of Alpha Memory and Beta Memory, which store intermediate results of each pattern matching. However, in scenarios with rapidly changing datasets, such as IoT, the reuse rate of intermediate nodes in RETE networks is low, leading to low cost performance in generating complex RETE networks. To address this limitation, scholars have made improvements to the RETE algorithm in the IoT field[6–8].

Due to the complexity of the RETE algorithm, IoT system developers often choose to use a general rule engine based on RETE, such as Drools, rather than implementing it from scratch. Drools is widely used in IoT and offers a comprehensive implementation of RETE. However, general rule engines like Drools have high structural complexity, making it challenging to replace the embedded RETE algorithm.

An alternative option is the open-source platform Thingsboard, which provides a dedicated rule engine for IoT device data flow processing. Thingsboard offers a modular rule editing interface with higher usability compared to Drools. Some researchers and developers have chosen to consolidate data visualization and data flow rule management into Thingsboard instead of using a general rule engine[13, 14]. The rule engine of Thingsboard is implemented as a chain structure, which is known as rule chain.

A rule chain primarily consists of three types of rule nodes:

- Filter node for filtering and routing message attributes.
- Enrichment node for updating the metadata of incoming messages.
- Action node for executing various actions based on the results of incoming message matching.

These three categories of nodes can be compared to the Rules, Facts, and Actions of the RETE algorithm. Thingsboard decomposes complete rules into rule nodes and combines these nodes into a rule chain to enable flexible rule configuration.

However, despite offering a comprehensive solution for IoT data analysis and visualization, Thingsboard rule engine has certain limitations. Firstly, it is tightly integrated with other modules of the platform, making it challenging to migrate to alternative systems. Secondly, rules are defined directly on individual devices, preventing users from creating rules that contains multiple devices.

This paper proposes a framework that incorporates the outstanding design patterns from both the Drools rule engine and the Thingsboard rule engine, while addressing their shortcomings in the context of IoT data flow monitoring and control.

Drools stores data that will be processed in a unified working memory and delegates all rule actions to a single executor. These designs are well-suited for the scenario discussed in this paper. Therefore, the framework presented in this paper intends to adopt these approaches by consolidating all data for rule monitoring into a shared memory and executing all actions uniformly using a separate executor thread.

However, the RETE algorithm, which significantly impacts the performance issues of Drools in IoT data monitoring, will not be utilized in the framework proposed in this paper.

Thingsboard's practice of assigning rules directly to specific devices, which binds a particular data stream from the input platform to a specific process, poses challenges in defining rules for monitoring multiple devices. To address this, the paper suggests importing all monitored data flows into a public cache, allowing unrestricted access to all rules.

Furthermore, all rule nodes in Thingsboard inherit from a common superclass. This inheritance structure leads to a certain level of interdependence between action nodes and condition nodes. To enhance the flexibility of our framework, we propose a complete separation of the processes of evaluating conditions and executing actions, with only the necessary parameters for action execution being transferred between them.

Thingsboard's rule model implements a corresponding rule node class for each condition and action, which is intuitive and user-friendly. The proposed framework inherits this design pattern.

3 IoT Rule Engine Framework

The paper proposes a framework of IoT rule engine that allows users to customize data flow monitoring and control rules in run-time of the IoT system. This framework enables users to upload new monitoring rules through REST API interfaces and control the activation state of these rules. Each rule specifies the data source it applies to, the condition it evaluates, and the action to be executed when the rule is matched. The active rule will be converted into a match function that is periodically called in the system.

To establish a common understanding, the paper introduces two notations: M and E. The notation M{K|V} represents a map with a key K and its corresponding value V. Similarly, E{V1,V2} defines a structure with data member V1 and V2.

3.1 DSL for Rule Description

In order to enable the execution of user-customizable monitoring and control strategies, we defined the syntax of the rules in the following manner.

- Datasource: {name{device_id, device_type, attribute}}
- Condition: {expression of name} or {condition_type: params}
- Action:{action_type:string contains symbol $name}

The Datasource field specifies the name and attributes of the datasource. Datasource represents real-time data obtained from a data flow. 'Device_id' serves as a unique identifier for the device within the system. 'Device_type' records the type of the device. 'Attribute' refers to a field in the message uploaded by the device, indicating the specific data that needs to be monitored. Multiple datasources can be listed in this field, separated by a semicolon.

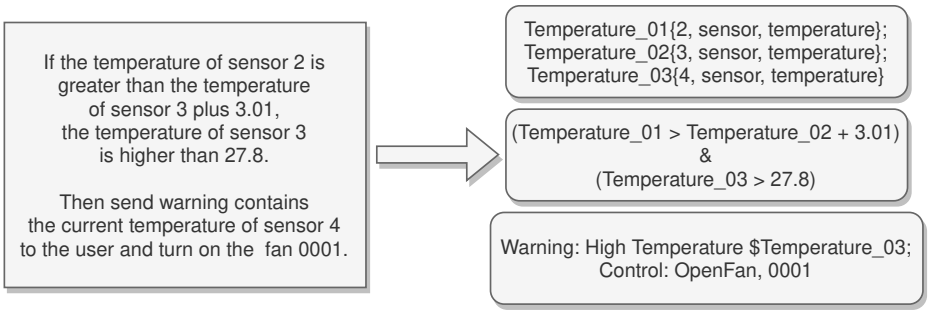
The Condition field defines the match condition for the data source. There are two types of conditions: expression condition and functional condition. Functional condition specifies condition type with symbols.

The Action field determines the action that will be executed if the rule matches successfully. The $ symbol can be used to refer to a data source defined within the same rule.

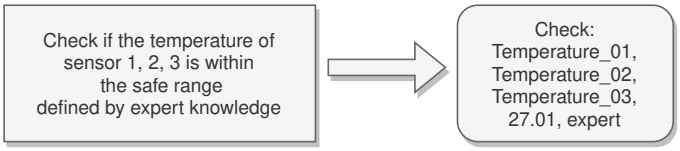
Figure 1 provides an example of rule DSL. Figure 1(a) demonstrates a usage of expression condition. Figure 1(b) illustrates a usage of functional condition.

3.2 Process of Match Function

First and foremost, it is important to comprehend the functioning of the match function (MF) that is produced by the rule parsing function and embodies our methodology for achieving rule matching during runtime. In corresponding



(a) Usage of Rule with an Expression Condition



(b) Usage of Functional Condition

Figure 1: Example of Rule DSL

with the three rule statements, the operation of MF can also be conceptualized as consisting of three components: obtaining the most recent data, performing conditional evaluation, and executing the necessary actions.

3.2.1 Maintaining and Retrieving datasources

Figure 2 outlines the logic for maintaining and retrieving updating data.

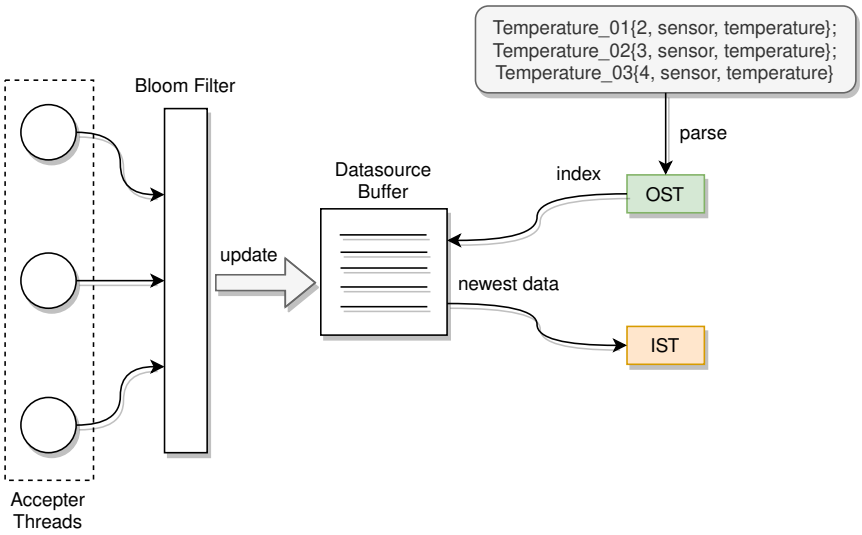


Figure 2: Maintaining and Retrieving Current Data

The rules store the up-to-date values of their data sources in a global map called the data source cache(DSB). When the MF is executed, it retrieves the necessary data for the rule from DSB. The following is the definition of DSB.

$$DSB = M\{Index|E\{Reference, Data\}\}$$

Where $Index$ is defined as follows.

$$Index = E\{device\_id, device\_type, attribute\}$$

"Reference" refers to the quantity of rules that mentions a particular datasource, while "Data" refers to the up-to-date value of datasource.

MF use the Datasource field as a reference to retrieve the relevant data from the DSB. In order to facilitate this process, a symbol table is introduces. Since the Datasource field is initially in string format when it is passed into the platform, it cannot be directly used as an index for querying. As a result, during the parsing of MF, the Datasource in string format is converted into another symbol table. This new symbol table uses symbols as indexes and the corresponding $Index$es in DSB as values. The symbol table that stores the up-to-date data $Index$ associated with each symbol is referred to as the outer symbol table (OST).

$$OST = M\{Symbol|Index\}$$

The inner symbol table (IST) stores the most up-to-date information about symbols.

$$IST = M\{Symbol|Data\}$$

Therefore, the OST is used to retrieve the up-to-date data from DSB, while the IST is used to receive and store these data.

It is crucial to emphasize that the DSB should consistently hold the latest data received throughout the system. The datasources within DSB are updated by a set of threads known as the accepter threads (AT), which are responsible for receiving external data. For each attribute in received messages, AT queries DSB to determine if a datasource exists. If exists, it is updated accordingly.

In IoT systems, there exists a notable disparity between the number of message attributes that users are concerned with and the total number of attributes contained within the message. The attributes found in a message typically include upload time, session number, device ID number, device type, communication protocol, environmental data, and other fundamental information. The majority of these attributes pertain to control information, which ensures communication reliability and message legibility, with only a small portion being environmental data of interest to users. As the DSB exists as a global data structure, it necessitates certain synchronization measures to ensure safe progress of concurrent reads, resulting in concurrency overhead during the query process. To address these considerations, a Bloom Filter[15] is used to reduce the frequency of AT accessing DSB.

3.2.2 Calculation of Specified Condition with datasource

Figure 3 depicts the logic for computing the results of rule matching.

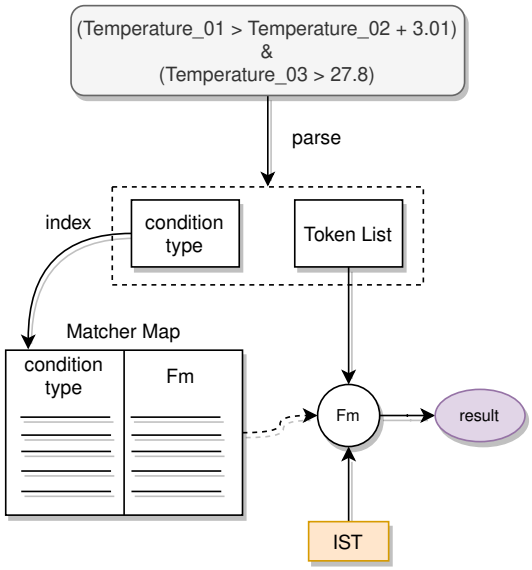


Figure 3: Calculating the Rule Matching Results

After retrieving the latest data and creating the IST, we use data contains in it to perform the computation as specified in the rule. Subsequently, we analyze the Condition field to determine the type identifier of the condition and the list of $Token$. The term $Token$ is defined as follows.

$$Token = E\{type, value, real\_num\}$$

The term 'type' means the classification of the $Token$, which can encompass variable types, bracket types, operator types, numeric types, or string types. The term 'value' denotes the string representation of $Token$. The term 'real_num' signifies the present value of the string. It is noteworthy that solely variable type and numeric type $Token$ will possess a populated 'real_num' field.

Subsequently, we use the condition type identifier to index Matcher Map to obtain the pattern matching function (F_m) that corresponds to this condition type.

$$MatcherMap = M\{condition\_type|F_m\}$$

We proceed by inputting the list of ISTs and $Token$ into the function F_m to obtain the execution result.

For functional conditions, the input parameter list of F_m contains only two types of $Token$s. One of them is variable Token, which can be used to query for a value of the IST. Another type is string $Token$, which will be parsed by F_m. In order to unify the handling of the two conditions, a condition type symbol will be auto assigned as index during parsing expression condition. The computation process for F_m of expression condition involves replacing the variable $Token$ in the list with the latest data, and then using a stack-based expression evaluation algorithm to calculate its value. This entails sequentially sending the $Token$ list of the succeeding expression order to a stack, where numerical $Token$s are directly placed into it. Operator $Token$s will extract the top two elements from the stack, perform the corresponding operation, and push the result back into the stack until the final result is achieved.

3.2.3 Execution of Specified Actions

Figure 4 illustrates the logic for executing actions within the rule.

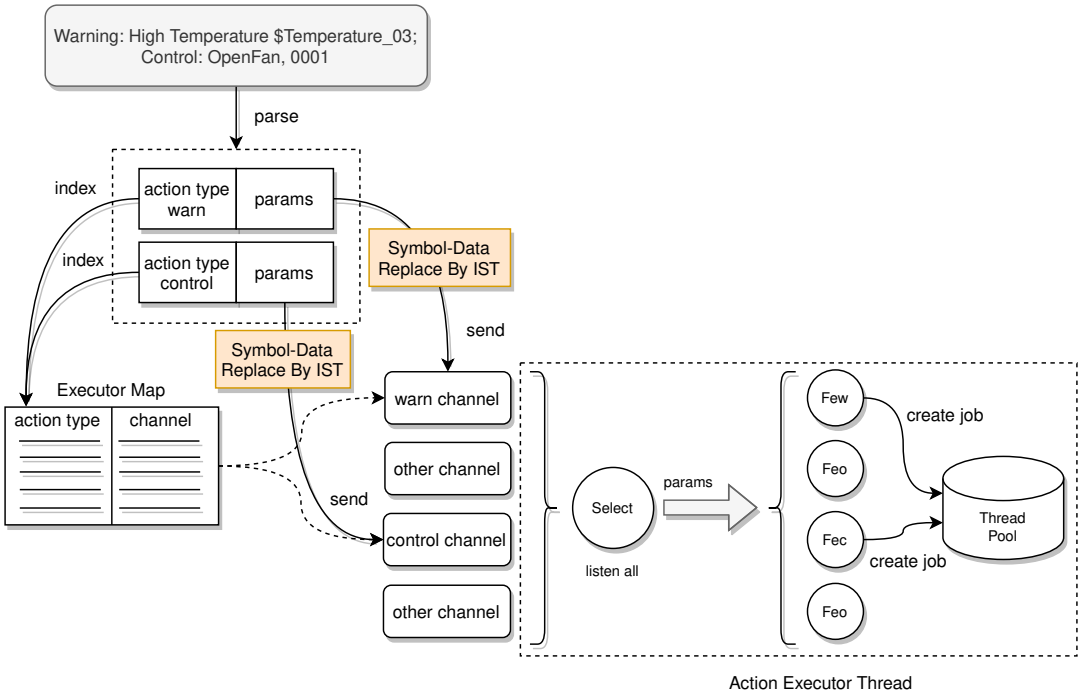


Figure 4: Executing Actions

After applying the conditional type to index F_m, the list of tokens is processed with the IST. This process means matching the most recent data with the rules. The actions specified in the Action field are executed when a match is found.

First, it is assumed that each action is executed by a corresponding execution function (F_e). This assumption is valid because even complex action logic can be encapsulated into a callable function. The function takes all the necessary parameters for action execution, which enhances code organization and maintainability, facilitating future development.

Under this assumption, the calling mode of F_e is further analyzed. Directly calling F_e by MF (main function) is not advisable as it binds the execution results of multiple F_es. If an error occurs during execution, all subsequent F_es will be affected. Additionally, the execution order of multiple F_es at the same level may vary depending on the order in which they are called by the main function.

One possible solution is to create a thread for each F_e to handle the execution. However, this approach has two drawbacks. Firstly, creating numerous threads can disrupt normal system operation, especially when there are a large number of rules. Secondly, there is no clear separation between condition matching and action execution, which hinders system modularization and distribution.

To address these issues, each F_e is assigned a channel for transferring the parameter list. A channel can be seen as a shared memory area implemented using the consumer-producer model. In the Executor Map, channels are organized and categorized based on their respective action types.

$$ExecutorMap = M\{action\_type|channel\}$$

In the proposed approach, the parameter list is sent to the channel in string format by the MF after a successful match. It is important to note that any datasources indicated by the $ symbols in the string are replaced with numerical values before being sent. The channels are all listened to by a single thread, referred to as the action executor (AE). The AE then forwards the received parameter lists to their corresponding F_es. These F_es are executed by a public threadpool. This design aims at limit the system resources that could be occupied by action execution, reduce the overhead associated with creating and destroying multiple threads, and effectively separate the processes of condition matching and action execution into two independently running modules.

3.3 Translation Process from Rule Text to Runtime Structure

Once the process of executing MF is comprehended, the subsequent inquiry concerns the production of MF from the rules. Specifically, it entails understanding how the rule parse function modifies the current system while interpreting the rule text and how it constructs the corresponding MF.

3.3.1 Parsing Datasource and Creating OST

The first step of creating a MF involves parsing the Datasource field and constructing the OST.

3.3.2 Condition Parsing, Categorization of Matching Conditions, and Token List Generation

After the initial step, the Condition field is evaluated based on its category. Depending on the category, different parse functions are called to generate the $Token$ list. For conditions of the expression type, the string expression will be initially parsed into a mid-order $Token$ list. This mid-order $Token$ list is then converted into a post-order $Token$ list, which allows the expression evaluation algorithm, F_m, based on the $Token$ Stack, to perform calculations. For conditions of the functional type, they will be directly parsed into a $Token$ list.

3.3.3 Action Parsing and Action List Generation

Parsing the Action field is a straightforward process. Each action type identifier and its corresponding parameters will be parsed as a structure named Action.

$$Action = E\{action\_type, params\}$$

These structures collectively constitute the Action List (AL).

3.3.4 Creating a Matching Function and Passing the Parsed Object as a Closure-Free Variable to the Function Block

Matcher generator function returns the MF associated with the rule by using the OST, condition type, $Token$ list, and AL as input arguments. The flow of the matcher generator function is to create and return a function that does not require any inputs or outputs. The generated function runs as follows.

1. The OST is used to query DSB to obtain the latest data of the symbols and construct the IST from it.
2. Subsequently, condition type is used to index and call the F_m, with IST and $Token$ list as input parameters. The return value of this call serves as a condition for determining whether the pattern matching is successful or not.
3. If the match is found, the AL is iterated, an IST-based symbolic-value substitution of the datasource is performed on the parameter list, and it is forwarded to the corresponding channel after substitution.

3.4 Rule Scheduling

The corresponding MF is specifically designed for the execution of rules. However, the framework also provides the capability to include rules that do not need to be executed immediately, allowing users to register them in the system. In order to facilitate scheduling, the framework defines four states for these rules: undefined, inactive, scheduled, and active. Additionally, the framework outlines the transition relationship between these states.

Scheduling logic of rules will be implemented with the assistance of an active rule map, a scheduled rule map, and a Rule Database (RDB).

Figure 5 elucidates the connection between rule state transitions and scheduling interfaces.

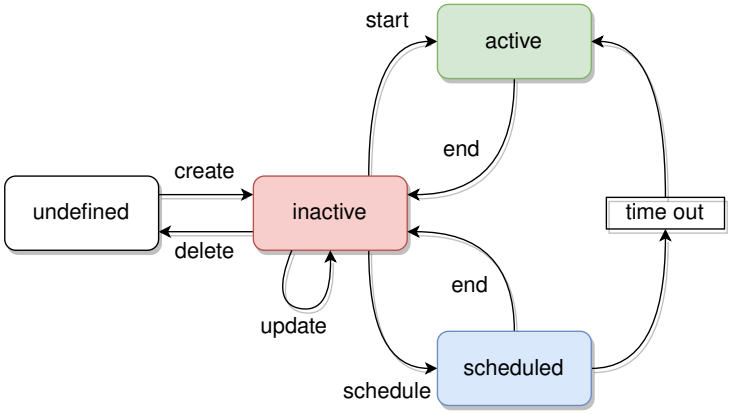


Figure 5: Scheduling Logic of Rules

3.4.1 Create Rules Interface

The rule is created and be saved to the RDB with the rule state initialize as 'inactive'.

3.4.2 Start Rules Interface

In order to match up-to-date data with rules in active state, a process known as the matcher thread (MT) runs continuously to periodically invoke MF. We found that the Cron[16] timed task framework is suitable for the requirements of our application. Therefore, we create a global Cron object and use it to execute all MF. The implementation is straightforward. We simply provide the execution period and the generated MF to the Cron object.

To ensure the latest data will be accessed and updated by AT, rules stored in the RDB are extracted and the Datasource is preparsed. Corresponding entries are then added to DSB, which increases the reference count of the target datasource or creates the datasource if it does not already exist. After this step, the rule text is parsed into MF. MF is then added to the Cron. Corresponding Cron entry ID is saved in the active rule map, with the Rule Table ID (RID) serving as the index. Finally, the rule status is updated to 'active'.

Figure 6 illustrates the process of activating a rule.

3.4.3 Schedule Rules Interface

A new timer, denoted as 'Timer,' is added into the scheduled rule map, with indexing based on the RID. Upon reaching its designated time, 'Timer' will call the 'Start Rules' interface and subsequently eliminate itself from the scheduled

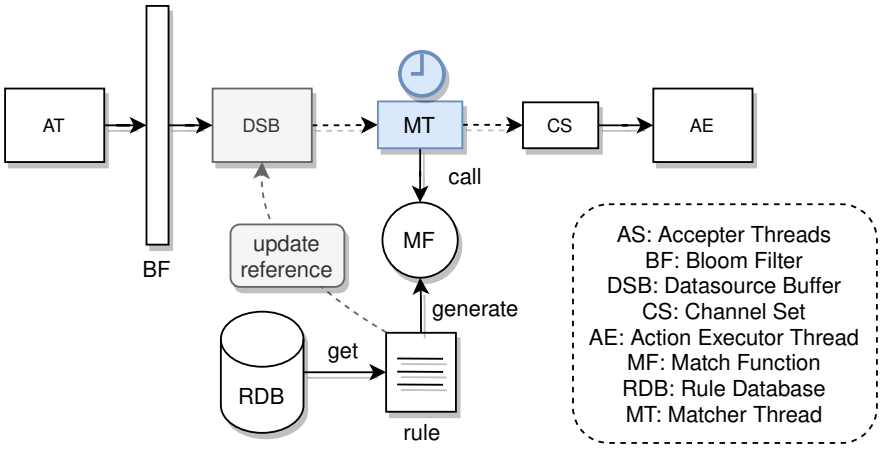


Figure 6: The Process of Activating a Rule

rule map. In the mean time, rule status will change to 'scheduled'. After a while, the rule status will undergo a transition from 'scheduled' to 'active' when timeout.

3.4.4 Update Rules Interface

Rules in RDB with a status of 'inactive' are updated. The rule status does not need to be updated.

3.4.5 End Rules Interface

The process of ending rules involves removing the corresponding entry in DSB, which decreases the reference count of the target datasource. If this reduction results in a count of zero, the entire datasource is deleted.

If the target rule has a status of 'scheduled' in the RDB, its RID is used to retrieve and delete entries from the scheduled rule map. Additionally, the corresponding Timer is halted.

On the other hand, if the target rule has a status of 'active' in the RDB, its RID is utilized as an index to fetch and delete entries from the active rule map. The fetched object entry is then used to remove the corresponding task from the Cron.

Ultimately, the status of the rule is modified to 'inactive'.

3.4.6 Delete Rules Interface

Rule with a status of 'inactive' in the RDB will be deleted.

4 Evaluation

A prototype system has been developed to evaluate the efficacy of the framework proposed in this paper.

4.1 Prototype System with Proposed Rule Engine Framework

We commence our analysis by examining the theoretical application scenarios of IoT rule engine framework. Subsequently, we proceed to develop a prototype system for the purpose of testing framework.

4.1.1 Theoretical Application Model

The data generated by devices is received and processed through AT. The data is then stored in MongoDB for long-term storage and historical data analysis. Additionally, the DSB is continuously updated with real-time data. To send control commands to the devices, the MQTT Broker is utilized, where the devices only need to subscribe to a specific MQTT topic for this purpose. Furthermore, our primary service database, MySQL, serves as RDB in this scenario.

Figure 7 presents the general application model.

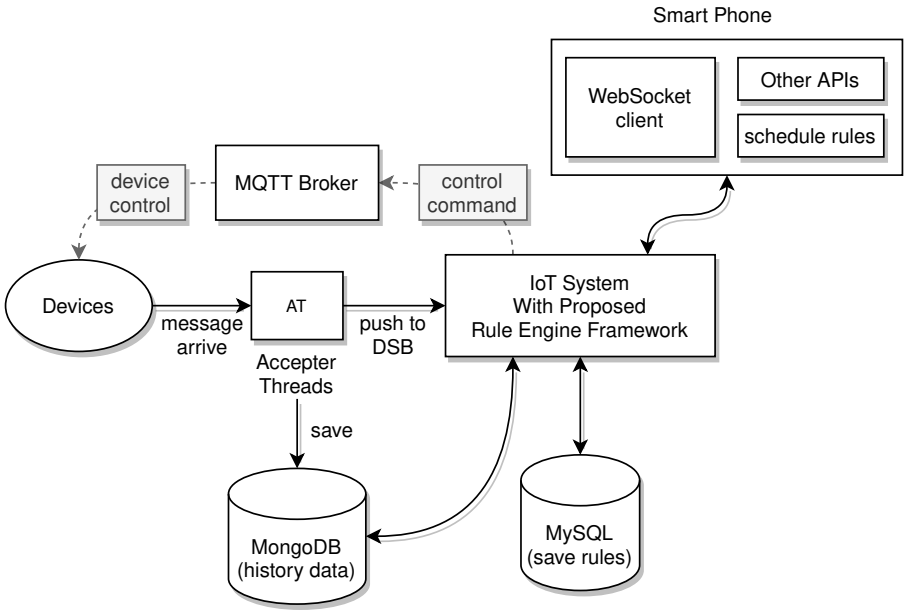


Figure 7: Theoretical Application Model

4.1.2 Prototype System

The prototype system was constructed with the intention of fulfilling the following objectives.

- To simulate data flow send by IoT devices, we use a Python script contains a TCP client, which transmits data to the server. Additionally, we implemented an AT , which contains TCP server, to receive data.
- To simulate the process of device command reception, we use a MQTT client to subscribe to a specific topic.
- To simulate the process of a user-side smartphone receiving information pushed from the server, we use a WebSocket client to establish a connection with the prototype system.
- For testing the functionality of our framework and simulating user actions, we utilized the Swagger UI web page to call the REST API interface provided by the system.

In our prototype system[1], we have implemented F_e for pushing MQTT messages and F_e for delivering messages to WebSocket clients. Furthermore, we have developed a collection of other interfaces, which are also essential for the system's functionality. These interfaces include WebSocket connection services.

Figure 8 illustrates the prototype system used for testing purposes.

4.2 Test Cases and Results

The following section presents the test cases used in the prototype system testing. We deployed the prototype system on the Ubuntu 22.04 operating system. Each of these test cases yielded the anticipated outcomes, thereby affirmed the effective validation of our proposed framework.

Test-1: Validation of Single-Device Rule with Expression Condition and Multiple Actions

Rule 1:

- Datasource: tem{1, Portable, temperature}
- Condition: tem > 22.1
- Action: WebSocket: 1,rule Matched, temperature is $tem!;Mqtt: localhost, 1883, admin, emqx@123456, test, control temperature

[1]https://github.com/ChenKen9869/DCA-IOT-system

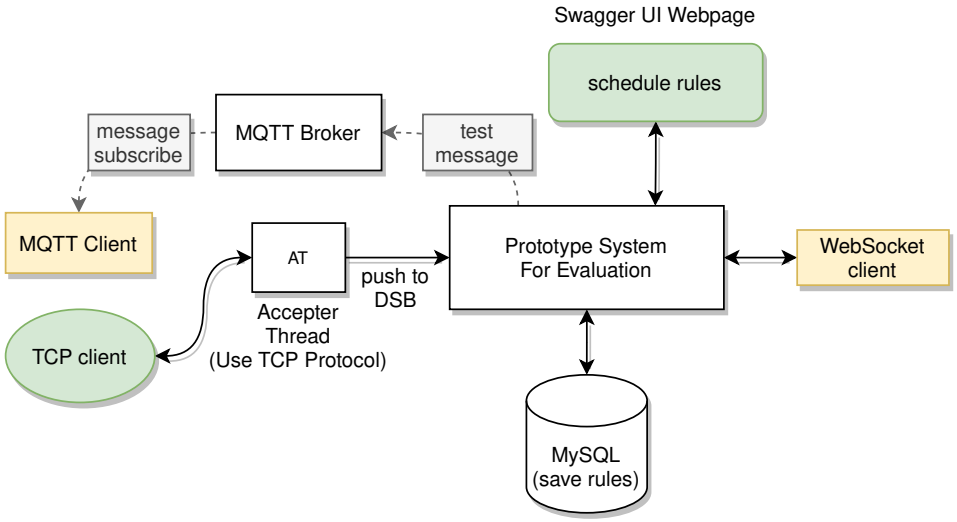


Figure 8: Prototype System for Testing

In order to evaluate the effectiveness of the rule, we performed an experiment using a Python script to transmit a range of messages to the server. It was observed that only a portion of these messages which satisfied the specified conditions were processed. These results obtained from this experiment aligned with our initial expectations.

1. MQTT client received expected message only if temperature attribute of message is greater than 22.1.
2. WebSocket client received expected message only if temperature attribute of message is greater than 22.1.

The testing methodologies for the following rules are similar. Both of them involve modifying the message body in response to specific conditions and then verifying if the MQTT client and WebSocket client receive the expected messages.

Test-2: Validation of Multi-Device Linkage Rules with Expression Condition and Multiple Actions

Rule 2:

- Datasource: tem_1{1, Portable, temperature}; tem_2{1, Fixed, temperature}
- Condition: (tem_2 > 25.3) & (tenm_1 > tem_2 +3)
- Action: WebSocket: 1,rule Matched, temperature is $tem_2 and $tem_1!;Mqtt: localhost, 1883, admin, emqx@123456, command, open fan

Test-3: Validation of Functional Condition through Multiple Actions

Rule 3:

- Datasource: longitude{3, Portable, longitude}; latitude{3, Portable, latitude}
- Condition: PointSurface: longitude, latitude, 1xx.40xxx2, 3x.92xx55, xx6.4xx70x, xx.89xx55, 1xx.40xx92, x9.8xx353, xxx.38xx46, xx.89x365
- Action: WebSocket: 1,rule Matched, position is $longitude $longitude!;Mqtt: localhost, 1883, admin, emqx@123456, command, find device

The PointSurface functional condition of Rule 3 is used to verify the inclusion of a point within a designated surface. This rule is utilized to establish whether the target device is situated within a predetermined range. From the third parameter onwards, the longitude and latitude coordinates of the points that define the polygonal boundary of the specified range are populated in a sequential manner. In order to safeguard privacy, certain numerical values in the test case have been obscured, with the use of 'x' as a replacement.

Test-4: Simultaneous Execution of Multiple Verification Rules

To ensure the successful attainment of desired outcomes when executing multiple rules concurrently, we commenced testing by starting Rule 1, Rule 2, and Rule 3 concurrently. And also using REST API interfaces to schedule them. During this phase of concurrent testing, the predetermined relationships and conditions for state transitions operated as expected. Subsequently, we extended our testing to encompass additional rules, and despite the higher complexity, we consistently obtained the expected outcomes.

Test-5: Testing of the average memory consumption of a single rule

We tested the average memory usage of a single rule by running 100,000 rules concurrently in the system. All testing rules have the same definition with Rule3. All rules are executed at a 5-second interval. Figure 9 illustrates the relationship between system uptime and memory usage.

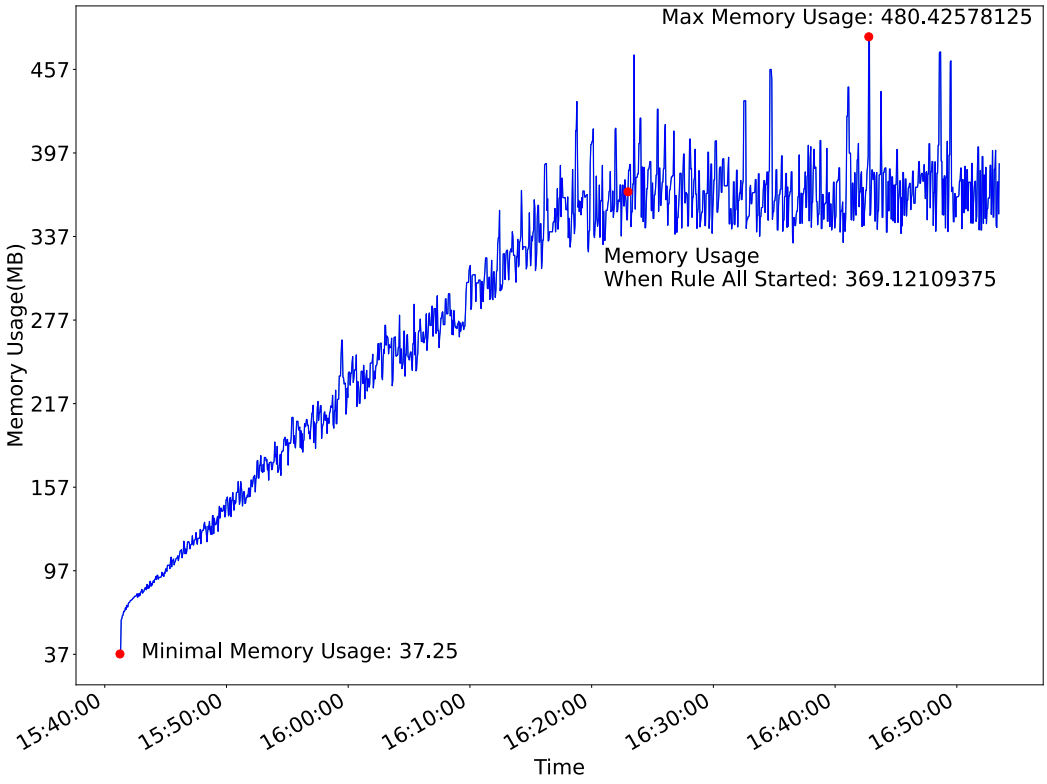


Figure 9: Memory Usage of Prototype System

Before we started the concurrent testing, the prototype system had a memory usage of 37.25MB. Then we started to create and run the rules with 20 threads, each responsible for 5000 rules. When the time reached 16:22:59 and all the rules were created, the system memory usage was about 369.12MB. We didn't create any new rules for nearly 30 minutes, and the system's memory footprint fluctuated around 369.12MB, peaking at around 480.43MB.

We used the peak memory usage of 480.43MB when all rules had started, minus the memory usage of 37.25MB when no rules were started, and then divided by the total number of rules 100,000 to get an average memory usage of about 4.54KB per rule. This result is well lower than the minimum average memory usage of about 3.4MB for the RETE algorithm and the RETE improved algorithm family[8].

5 Discussion

After verified the theoretical correctness of our framework, we will now engage in a comprehensive examination of its validity, challenges, potential methods for addressing limitations, and the usage scenarios.

5.1 Effectiveness

In light of the research background and research objectives previously discussed, we assess the effectiveness of the proposed framework by considering both theoretical perspectives and empirical findings. Through our experimental verification, we have confirmed the validity of the suggested rule engine. Our subsequent discussion primarily focus on the applicability, ease of use, and potential for expansion of our framework.

5.1.1 Easily Integrated with almost any IoT system

The proposed framework can be easily integrated into the existing mainstream structures of IoT system.

- **Resource Shortage System**: The suggested framework does not impose substantial memory overhead, in contrast to the RETE algorithm, rendering it appropriate for systems with limited resources. Furthermore, the comprehensive implementation of the framework can be accomplished as long as the deployed system has multithreading capabilities. In situations where the underlying hardware operates exclusively in a single-threaded mode, it is feasible to integrate the framework within the higher-level software of the system.
- **Distributed Systems**: Maintaining a low level of coupling between individual components is of utmost importance in order to ensure the smooth functioning of a distributed system. The necessary modifications to accommodate a distributed system are not significant. For example, in the case of the DSB, it is adequate to encapsulate it within a separate microservice and establish external interfaces for updating, deleting, and querying data sources. Likewise, for components such as the AE that require substantial resources, we can manage them within separate microservices and provide dedicated interfaces for receiving lists of action parameters.
- **Systems dependent on third-party services**: In an IoT system that relies on a third-party service, certain devices within the system do not directly send messages to the main server. Instead, they send messages to a server managed by the manufacturer of the device. This server then responds to HTTP data requests from users of the device. In this particular situation, our main objective is to create an AT that regularly queries the third-party service to obtain the data and subsequently transfers it to DSB.

5.1.2 User-friendly

The rule syntax of Drools is based on the Java language, which may pose a degree of complexity for users. In contrast, Thingsboard rule engine utilizes a modular user interface and JavaScript language for rule creation, rendering it relatively more accessible to users compared to Drools. The framework proposed in this paper adopts a specifically designed DSL for rule deployment. This DSL consists of only three statements with a simple syntax that can be readily comprehended and mastered by users without extensive computer expertise. Moreover, scheduling rules through REST APIs within the framework is also user-friendly.

5.1.3 Effortlessly Scalability

The framework discussed in this paper have the room for improvement of providing users with the ability to define their own data processing procedures using a general-purpose programming language. Thingsboard incorporates a JavaScript interpreter service to enable customizing rule nodes. Our framework can also support customizing Condition and Action by wrapping interpreters for script languages, such as JavaScript or Python, as F_m or F_e.

Primary components, MF, AE, and AT, demonstrate low coupling. This is evident as there are no direct calls between these components. Instead, their dependencies are established through indexing or intermediate caching mechanisms. As a result, extending F_m, F_e, and AT is a straightforward process.

The extensibility of the rule DSL in the framework is enhanced by incorporating design principles from general-purpose programming languages. For example, the rule DSL is able to expand to allow for the definition of constants. These constants could be parsed and stored in a constant table along with their symbols and data values. The constant table can then be queried for future use. Additionally, rule chaining can be implemented in the framework as well. Rule chaining means automatically triggering the next rule upon the successful matching of the current rule. If the preceding rule fails, the subsequent rule will not be triggered. This is achieved by introducing a new F_e, which modifies a specific

cell of DSB to a predefined value. Then the preceding rule uses such F_e to update DSB, while the subsequent rule take corresponding DSB cell as its datasource.

5.2 Challenges and Feasible Countermeasures

In the current framework, pattern matching is triggered periodically by the Cron. The trigger condition is based on the timing of Cron's execution interval. When triggered, data corresponding to the rule is retrieved from DSB. However, the framework lacks the ability to check if the retrieved data has been previously validated or to detect message omissions. This could lead to the processing of duplicate messages if the datasource in the DSB has not been updated when pattern matching occurs. Additionally, if multiple datasources are updated quickly within the pattern matching execution interval, only the latest one will be used for pattern matching.

One possible solution is to keep a record of message upload frequencies for each device type and align the pattern matching execution period with this frequency. However, network congestion or downtime of IoT devices can still cause delays in updating DSB, potentially resulting in message duplicates. Network congestion can also lead to delayed message arrivals, causing consecutive messages arrive and message omissions within a single execution cycle. While setting an appropriate pattern matching execution cycle can mitigate message omissions and duplicates, alternative trigger conditions may be necessary in certain special scenarios.

To prevent duplicate data processing, a session field can be introduced in both DSB and OST to track duplicate data. Before each matching execution, the session of the data source in the OST is compared to the session in the data source cache. If the OST session is smaller, the session in the OST is updated, and matching proceeds. This approach helps avoid message duplication and detects message omissions, but it does not guarantee against message omissions.

For scenarios where both preventing duplicate data processing and absolute avoidance of missing data are crucial, a push trigger condition can be used. In this approach, DSB proactively pushes new data to the MT by a channel. Instead of storing the actual data values in DSB, a map is maintained in DSB with the RID as the key and the channel as the value. When updating a specific datasource, DSB sends the same message to all channels in the map. Additionally, a continuous loop is used to invoke MF indefinitely, rather than depending on the Cron. MF listens to each datasource channel using the rule ID and the OST. When data arrives in a channel, it populates IST and initiates pattern matching as soon as IST is filled. In this setup, reference counting in the DSB is no longer necessary as the map capacity represents the number of rules requesting data from the source.

When implementing push-style trigger conditions as described above, it is advisable to avoid creating rules for multi-device processing, especially when the message arrival frequencies vary significantly between devices. This is because building the IST can become slower, especially if data from one symbol arrives significantly earlier than data from another symbol. If several batches of new messages arrive during this time, the threads responsible for sending data to the channel in DSB may become blocked for an extended period, impacting the update process for other data sources. In systems using such push-type trigger conditions, controlling the execution period may not be suitable as it could block the DSB update thread if data is not fetched within the execution interval.

5.3 Usage Scenarios

Based on the aforementioned analysis of its effectiveness and limitations, it is apparent that the framework possesses a broad range of potential applications. However, it is necessary to mention that the framework may not be suitable for situations that necessitate strict real-time requirements. This is due to the fact that device control relies on periodic pattern matching executions rather than continuous data flow monitoring, which could potentially result in delays in issuing control instructions.

We contend that the rule engine framework proposed in this paper is well-suited for IoT scenarios such as home automation, agricultural applications, and personal wearable devices. However, it may not exhibit optimal performance in scenarios that require precise real-time control, such as chemical production processes.

6 Conclusion

This paper presents a novel rule engine framework that is specifically designed for the IoT domain. The framework contains a user-friendly DSL and enables the seamless modification of data flow monitoring policies without causing disruptions to IoT system operations. The framework is characterized by its high scalability and versatility, making it suitable for a wide range of scenarios, particularly those that do not require strict real-time device control.

Proposed framework stands out in addressing challenges related to monitoring IoT data flows by minimizing additional memory consumption compared to rule engines based on the RETE algorithm. Additionally, it offers improved flexibility and user-friendliness compared to Thingsboard rule engine.

In future researches, our study will focus on enhancing the robustness of the proposed rule engine framework. This may involve allowing users to assign priorities to pattern matching or action execution in rules, or figuring out methodologies for detecting semantic discrepancies that may occur when multiple rules are concurrently utilized.

References

[1] Forgy C L. Rete: A fast algorithm for the many pattern/many object pattern match problem[J]. Artificial Intelligence, 1982, 19(1): 17-37.

[2] Kaed C E, Khan I, Van Den Berg A, etc. SRE: Semantic Rules Engine for the Industrial Internet-Of-Things Gateways[J]. IEEE Transactions on Industrial Informatics, 2018, 14(2): 715–724.

[3] Kiran M P R S, Rajalakshmi P, Bharadwaj K, etc. Adaptive rule engine based IoT enabled remote health care data acquisition and smart transmission system[A]. 2014 IEEE World Forum on Internet of Things (WF-IoT)[C]. 2014: 253–258.

[4] Mainetti L, Mighali V, Patrono L, etc. A novel rule-based semantic architecture for IoT building automation systems[A]. 2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)[C]. 2015: 124–131.

[5] Mazon-Olivo B, Hernández-Rojas D, Maza-Salinas J, etc. Rules engine and complex event processor in the context of internet of things for precision agriculture[J]. Computers and Electronics in Agriculture, 2018, 154: 347–360.

[6] Gao T, Qiu X, He L. Improved rete algorithm in context reasoning for web of things environments[C]//2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing. IEEE, 2013: 1044-1049.

[7] Li Q, Jin Y, He T, et al. Smart home services based on event matching[C]//2013 10th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD). IEEE, 2013: 762-766.

[8] Chen J, Li Y. Improved rule engine based on RETE algorithm for smart-home environment[C]//2019 IEEE 19th International Conference on Communication Technology (ICCT). IEEE, 2019: 344-349.

[9] ThingsBoard - Open-source IoT Platform[EB/OL]. /2023-09-28. https://thingsboard.io/.

[10] Drools - Drools - Business Rules Management System (Java TM , Open Source)[EB/OL]. /2023-09-28. https://www.drools.org/.

[11] Luo X, Fu Y, Yin L, et al. A scalable rule engine system for trigger-action application in large-scale IoT environment[J]. Computer Communications, 2021, 177: 220-229.

[12] Cambra C, Sendra S, Lloret J, et al. An IoT service-oriented system for agriculture monitoring[C]//2017 IEEE International Conference on Communications (ICC). IEEE, 2017: 1-6.

[13] De Paolis L T, De Luca V, Paiano R. Sensor data collection and analytics with thingsboard and spark streaming[A]. 2018 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)[C]. 2018: 1-6.

[14] Kadarina T M, Priambodo R. Monitoring heart rate and SpO2 using Thingsboard IoT platform for mother and child preventive healthcare[J]. IOP Conference Series: Materials Science and Engineering, IOP Publishing, 2018, 453(1): 012028.

[15] Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.

[16] cron package[Source Code]. /2023-09-28. https://pkg.go.dev/github.com/robfig/cron.