A FLEXIBLE IOT RULE ENGINE FRAMEWORK FOR CUSTOMIZING MONITORING AND CONTROL STRATEGIES

A PREPRINT

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

September 28, 2023

ABSTRACT

Handling data generated by high volume devices in IoT systems has been a longstanding topic. Many of these research works have explored how to apply a generic rule engine centered on the RETE algorithm to device data flow monitoring, and proposed improvement works for the RETE algorithm. However, it is not easy to modify the generalized rule engine in practical development. The open-source platform Thingsboard had proposed an IoT-specific rule engine that does not use the RETE algorithm, and its good interaction mode attracted many developers and researchers, but the coupling between the rule module and the platform and the limitation that it is difficult to formulate the rules for multi-device monitoring lead to the lack of flexibility in its application mode. In this paper, for IoT device data stream monitoring, we propose a rule engine framework that is flexible and easy to use, easy to extend, and allows the formulation of multi-device monitoring rules, and design a DSL for user interaction. a prototype system applying the framework is realized, and the correctness of the theoretical approach is verified through experiments. Finally, through analyzing and exploring, it is concluded that the framework can be flexibly applied to most IoT scenarios and is more effective in scenarios with less stringent control real-time requirements.

Keywords IoT · Rule Engine · Dataflow Monitoring and Control

1 Introduction

The huge data magnitude of IoT devices and the high diversity of user requirements together determine the importance of customized monitoring for device data streams. For a long time, researchers have favored the use of generic rule engines based on the RETE[1] algorithm to solve such problems. A rule engine is a software technique that decouples the rule definition from the software implementation. A rule is usually a set of statement pairs in the form of IF-THEN, which means that if the condition described in the IF statement is satisfied, the action described in the THEN statement is performed. A rule engine parses the rule statements and actually deploys the rules into the software system. RETE[1] algorithm is currently the most widely used rule engine construction method, which defines an efficient algorithm for performing rule-data matching.

The RETE algorithm saves the intermediate operation results in the process of calculating the matching results in order to reduce the time complexity of rule matching, so that the results of the last operation can be reused if a certain input attribute in the next batch of data does not change. However, in IoT scenarios, device messages change quickly and the reuse rate of intermediate results is low. This leads to little success of the space-for-time strategy. To address this problem, researchers have proposed improved RETE algorithms as a way to improve its effectiveness in IoT application scenarios. However, in order to adapt to various application scenarios, the internal implementation of the generalized rule engine represented by Drools is usually complicated, so the operation of replacing the RETE algorithm is difficult from a practical point of view.

However generic rule engines are not the only solution to the problem of customized data stream monitoring. The open-source IoT data platform Thingsboard abandons the RETE algorithm in its rule engine implementation and proposes a rule model for streaming data processing as an alternative. The Thingsboard rule engine has a more user-friendly interaction than the general-purpose rule engine, and has been more widely used in various IoT scenarios. This suggests that a relatively concise rule model may be more favorable to IoT users than a generic rule engine with a comprehensive syntax. However, the strong coupling between the components of the Thingsboard platform makes it difficult to apply its rule engine to other systems individually. There is also another issue that the Thingsboard rule engine does not support managing data flows from multiple devices in a single rule.

In the problem of customizing IoT data stream monitoring, both of the above mentioned solutions that have been mainly applied have some potential limitations. RETE-based generic rule engines are not suitable for handling continuously changing data sets, as well as the poor portability and feature completeness of the Thingsboard rule engine. Therefore, this paper proposes a flexible and easy-to-extend IoT-specific rule engine framework that supports multi-device monitoring, defining the most basic DSL for describing rules. The framework can be easily embedded into almost any structured IoT system, and will be mainly used to change the monitoring rules for device data flow at runtime of the IoT system. In this paper, we have implemented a prototype system based on the framework, conducted experiments on the prototype system using the REST API, and verified the correctness of the theoretical approach through a set of test cases. The effectiveness, scalability and limitations of the framework are explored at the end of the paper, and possible application scenarios are discussed on this basis.

The paper firstly introduces the results and shortcomings achieved by the Generalized Rule Engine and the Thingsboard Rule Engine, and puts forward the research objectives of this paper. In the subsequent chapters, a new IoT rule engine framework and a DSL for describing rules are proposed.To facilitate the readers' understanding, the paper starts from the practical requirements and firstly gives the design of the rule DSL, then gives the flow of the rule matching function according to its syntax, and finally gives the algorithm for parsing the rule text and the method for scheduling the rules. The remaining part of the paper illustrates the implementation of the prototype system and the test cases and test results, and finally the proposed framework is discussed and summarized more comprehensively.

2 Related Works

The RETE algorithm, which is widely used in general-purpose rule engines, is essentially an efficient pattern matching algorithm. the RETE algorithm refers to the object to be matched as a fact, and pattern matching refers to the process of matching the fact with the rule. the RETE algorithm compiles the user-defined rules into a network structure that is more conducive to parallel computation and can significantly reduce the amount of computation, which is called the RETE network. the RETE network adopts the design of Alpha Memory and Beta Memory, and the two types of Memory act in different phases of pattern matching. network adopts the design of Alpha Memory and Beta Memory, two kinds of Memory respectively act in different stages of pattern matching, save the intermediate results of each pattern matching, when the next batch of data arrives, if the data of a certain to-be-matched object has not been changed, then the intermediate operation results of the last time can be used directly. Environmental data collected by IoT devices usually changes rapidly and is uploaded frequently. In the case of rapidly changing data sets, the intermediate node reuse rate of RETE networks is low, which leads to the cost-effectiveness of generating complex RETE networks. To address the limitations of RETE algorithm in the field of IoT, some scholars have already made improvement work for RETE[2–4].

From a practical point of view, considering the complexity of the RETE algorithm, developers of IoT systems usually choose to directly use a generic rule engine based on the RETE algorithm instead of implementing the RETE algorithm from scratch. Among many generic rule engines, Drools[5] implements all the details of RETE and is the most widely used generic rule engine in IoT, which can be seen in many related researches[6, 7]. However, the generalized rule engine represented by Drools has high structural complexity, and it is difficult to replace the RETE algorithm in it.

Thingsboard is an open source IoT data processing and visualization platform, which proposes a rule engine dedicated to IoT device data stream processing, while providing a modular rule editing interface, compared with the Drools rule engine based on the Java language to define the DSL, the use of the Thingsboard rule engine has a lower threshold. After the emergence of Thingsboard[8] open source IoT platform, some researchers and developers chose to unify the functions of data visualization and data stream rule management to Thingsboard, instead of using the general rule engine[9, 10]. The rule engine of Thingsboard is implemented in a chain structure, and the complete rules configured for a device are called rule chains. The rule chain mainly consists of three kinds of rule nodes, namely Filter node for filtering and routing message attributes, Enrichment node for updating the metadata of incoming messages, and Action node for executing various actions based on the results of incoming message matching. The three types of nodes are analogous to the Rules, Facts, and Actions of the RETE algorithm. Thingsboard splits complete rules into

rule nodes and combines these nodes into a rule chain to realize flexible rule configuration and deployment schemes. Although Thingsboard provides a more complete IoT data analysis and visualization solution, its rule engine has potential limitations. Firstly the rules engine is strongly coupled with other modules of the platform, making it difficult to migrate to other systems. Secondly rules are defined directly on top of a single device and users cannot create rules that encompass multiple devices.

The framework proposed in this paper inherits some of the best design patterns of Drools rule engine and Thingsboard rule engine and optimizes them for their shortcomings in the IoT data stream monitoring problem.

The Drools rule engine saves the data to be processed to the working memory in a unified way, and hands over all rule actions to a unified executor for execution. The design of working memory and executor is suitable for the target scenario of this paper. Therefore the framework proposed in this paper considers to refer to this approach by consolidating all the data for rule monitoring in one common memory as well as executing all the actions uniformly using a separate executor thread running separately. The application of the RETE algorithm is the main reason for the performance issues of the Drools rule engine in the IoT data monitoring problem, therefore the RETE algorithm will not be used in the framework of this paper.

Thingsboard assigns rules directly to a particular device, which is equivalent to binding a particular data stream from the input platform to a particular process, making it difficult to define multi-device monitoring rules. In this paper, we consider importing all monitored data streams into a public cache, allowing all rules to be freely accessible. In addition, all rule nodes in Thingsboard inherit from a public parent class, and there is coupling between action nodes and condition nodes. In this paper, we consider completely separating the process of judging conditions and executing actions, and only transferring the parameters required for action execution between the two, so as to improve the flexibility of the framework. The Thingsboard rule model implements a corresponding rule node class for each judgment condition and action, which is intuitive and easy to use. This design is intuitive and easy to use. The framework in this paper inherits this design pattern and allows the user to specify conditions and actions in rules using specific strings.

3 IoT Rule Engine Framework

The IoT rule engine framework proposed in this paper is actually a method that can support users to customize the change of data flow monitoring rules in the system during the operation of the IoT software system. In this method, the user uploads a new monitoring rule through a REST API interface and controls the activation state of the rule, which specifies the data source on which the rule acts, the judgment condition of the rule, and the action to be executed after the rule is successfully matched. The active rule is parsed into a matching function that is called periodically by one of the system's subthreads.

Firstly, the following two notations are agreed upon: M, E, which denote map, and structure respectively; M{K|V} represents a map with key K and value V; E{V1,V2} denotes the definition of structure, and V1,V2 denote the member variables in the structure respectively.

3.1 DSL for rule description

To enforce user customizable monitoring and control strategies, we designed syntax of rules as below.

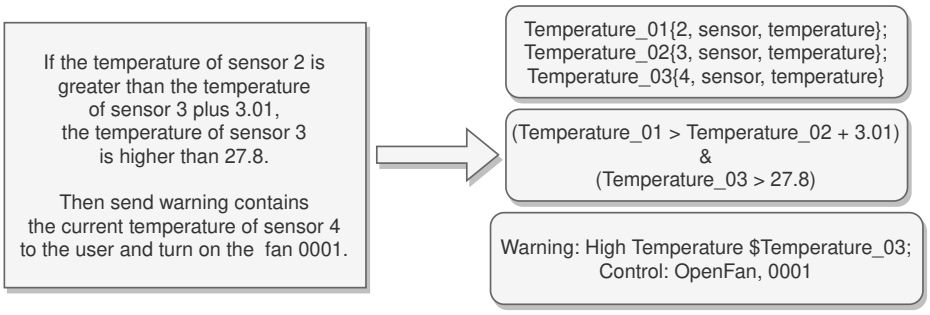
- Datasource: {name{device_id, device_type, attribute}}
- Condition: {expression of name} or {condition_type: params}
- Action:{action_type:string contains symbol $name}

The Datasource field defines the name and attributes of the datasource, where the attribute device_id is the unique identification number of the device in the system, device_type is the type of the device recorded in the system, and attribute is the name of a field in the body of the message uploaded by that device. This way of definition describes exactly which data in the system needs to be monitored. In this field, different data sources are separated by a semicolon ";".

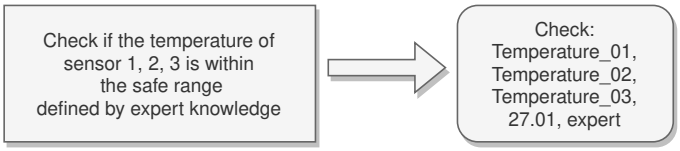
The Condition field defines the judgment condition for the data source, i.e., the judgment condition for the rule. There are two types of judgment conditions, those in the form of expressions and those in the form of functions that specify the symbols of the type of condition.

The Action field defines the action that will be performed if the rule matches successfully. The $ symbol is allowed to refer to a data source defined within the same rule.

Figure 1 shows example of rule DSL. We uses expression condition in Figure 1(a). And Figure 1(b) shows a example of functional condition.



(a) Example Of Rule With Expression



(b) Example Of Functional Condition

Figure 1: Example Of Rule DSL

3.2 Process of match function

First of all, we have to make clear how match function(MF) runs, which is generated by the rule parsing function, and it represents our idea of realizing runtime rule matching. Corresponding to the three rule statements, the running process of MF can also be understood as consisting of three parts, namely, obtaining the latest data, performing conditional judgment, and executing the required actions.

3.2.1 Maintaining and fetching of newest data

Figure 2 shows the logic for fetching and maintaining up-to-date data.

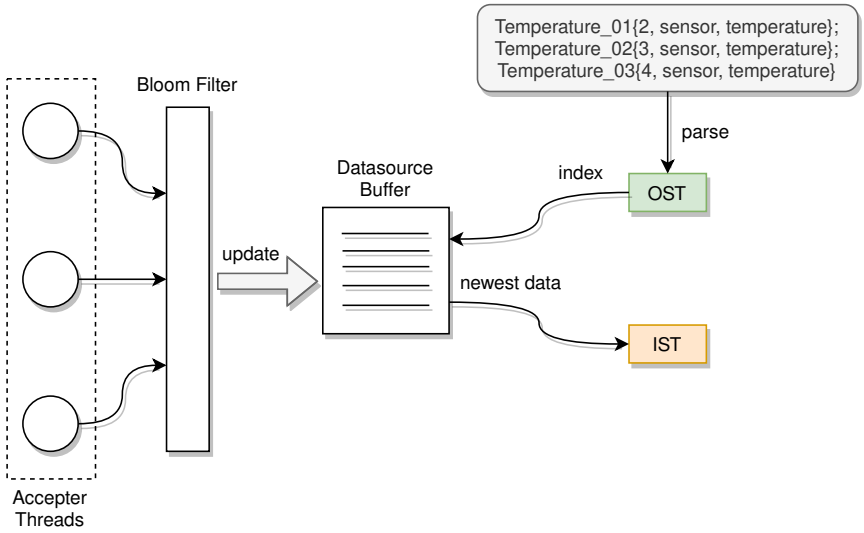


Figure 2: Maintaining And Fetching Up-to-date Data

All the rules deployed in the system have the latest values of their data sources stored uniformly in a global map called the data source cache. The first action done by the matching function is to fetch the data required by the rule from the data source cache. The structure of the data source cache (DSB) is $M\{Index|E\{\text{Reference,Data}\}\}$, where $Index$ is $E\{\text{device\_id,device\_type,attribute}\}$, Reference denotes the number of rules that refer to this data source, and Data denotes the current value of the data source.

MF utilizes the Datasource field as an index to query the corresponding data in the DSB and builds a table of symbols inside the function. Since the Datasource field is in string format when passed into the platform, it cannot be directly used as a query index. Therefore, in the process of parsing MF, the Datasource in string format is converted in advance into another symbol table with symbols as indexes and values as corresponding indexes in DSB. We call the symbol table that holds the latest data index corresponding to the symbol the outer symbol table (OST), and the symbol table that is built inside the function and holds the latest data of the symbol is called the inner symbol table (IST). From this, the OST is used to query the DSB and get the latest data, and the IST is used to receive and save this latest data.

At this point, the MF has obtained all the data it needs to run. Noting that the DSB should always hold the latest data received throughout the system, the data in the DSB is maintained by the subthread (accepter thread, AT) of the system that receives external data. The sub-thread, for each received message, queries the DSB to see if a data source exists in the DSB that monitors a particular attribute in the message, and updates the data source if it exists.

We note that in the IOT platform, there is a large difference between the number of message attributes that the user cares about and the full number of attributes contained in the message. The attributes contained in a message generally include the time of message upload, the number of sessions, device id number, device type, communication protocol, environmental data and other basic information, most of which are control information to ensure the reliability of communication and the legibility of the message, and only a small portion of which are environmental data that the user really cares about. The DSB, as a global data structure, requires certain synchronization measures to ensure the smooth progress of concurrent reads, and therefore its query process has concurrency overhead. Combining the above two points, we use Bloom Filter[11] to reduce the frequency of AT accesses to the DSB.

3.2.2 Calculation of specified condition with up-to-date data

Figure 3 shows the logic for calculating the rule matching results. After taking out the latest data and constructing

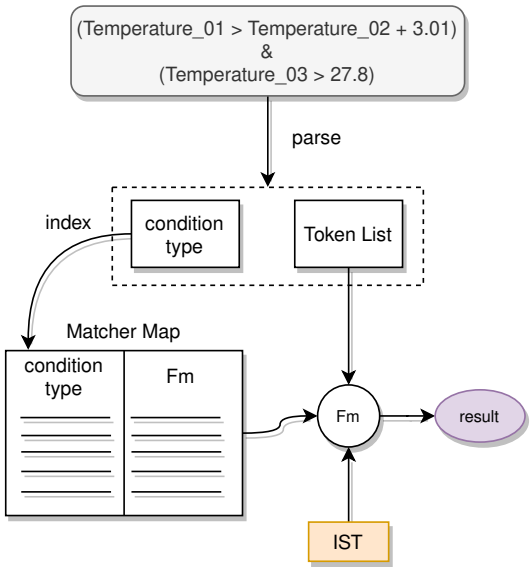


Figure 3: Calculating The Rule Matching Results

the IST, we take the data from the IST and perform the computation specified in the rule. We parse the Condition field, from which we parse the condition type identifier and the list of $Token$. the $Token$ is in the form of $E\{\text{type,value,real\_num}\}$, where type is the type of the $Token$, where there are variable type, bracket type, operator type, numeric type, string type; value is the $Token$'s string form value; real_num represents the current value of the string, only the real_num of the $Token$ of variable type and numeric type will be filled. After that, the condition type identifier index Matcher Map is used to get the pattern matching function (F_m) corresponding to this condition, and then the list of ISTs and $Token$ is fed into the F_m to get the running result.

There are only two kinds of $Token$s in the $Token$ list of the input parameter of the F_m for the functional matching condition, one is the variable $Token$, which can be queried for the value in the IST, and the other is the string $Token$, which is responsible for parsing by the MF. For the conditions described in expression form, a condition type symbol is assigned to it as an index by default during parsing, and the computation process corresponding to F_m is to use the latest data to replace the variable $Token$ in the $Token$ list, and then use the Stack-based expression evaluation algorithm to evaluate the value (i.e., the $Token$ list of the succeeding expression order is sent to the stack sequentially, and the numerical $Token$ is directly put into the stack. Operator $Token$ will take out the top two elements of the stack and then press the result into the stack until the final result is obtained).

3.2.3 Execution of the specified actions

Figure 4 shows the logic for executing the actions in the rule. After we indexed out the F_m with the conditional type

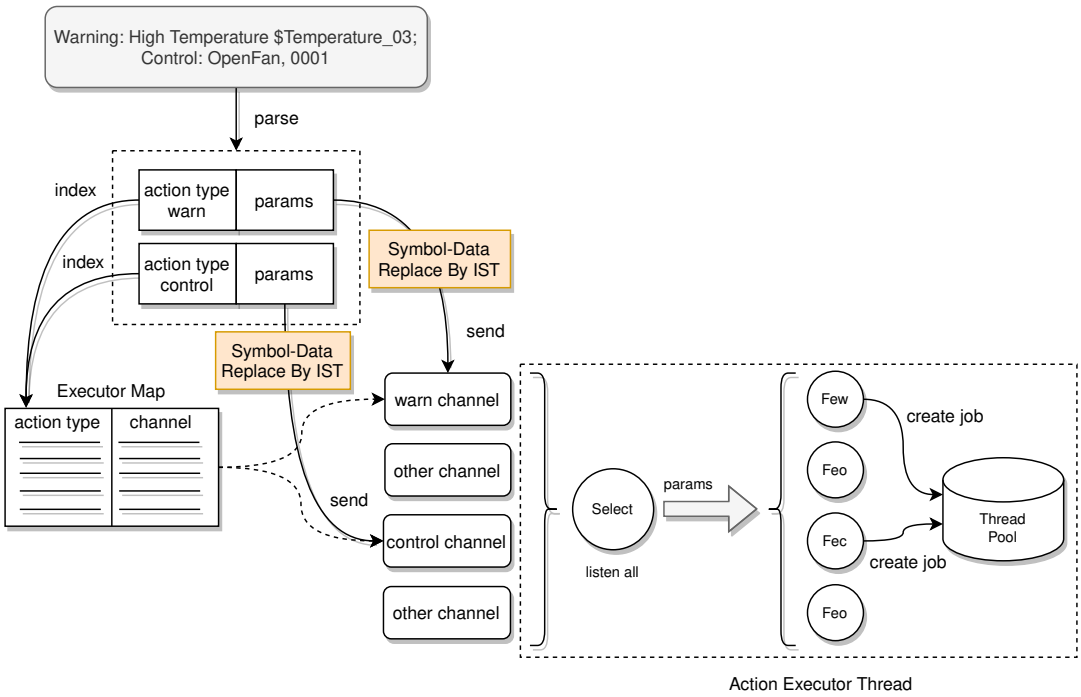


Figure 4: Executing Actions

notation, we passed the list of tokens transformed by the Condition field into the F_m along with the IST generated through the Datasource field, and got the result of this latest data matching with the rules, and when the matching result is true, we need to execute the actions described in the Action field.

We first assume that the execution of each action is done by an execution function (F_e) respectively. This assumption is valid, because even more complex action logic can be ultimately encapsulated into a callable function, the function's parameter list for the execution of all the parameters required for the action, such a design to enhance the hierarchy and neatness of the code, easy to maintain and extend the subsequent development.

Under this assumption, we can continue to analyze how F_e is called. The practice of calling F_e directly by MF is not reasonable, because it binds the execution results of multiple F_e's, if one of them has a bug during execution, all the subsequent F_e's will be affected. In addition, the execution order of multiple F_e's that are supposed to be at the same level may also differ depending on the order in which the main function is called. One possible solution is to create a child thread for each F_e to execute the call. However, there are two drawbacks of such a solution, one is that the creation of too many sub-threads affects the normal operation of the system when there are many rules, and the other is that there is still no separation of condition matching and action execution, which is not conducive to the modularization and distribution of the system (if the resources required for the execution of an action exceed the upper limit of the system's ability to cope with the system, then the distributed solution will probably be proposed, and at this time, if the separation of the condition matching and action execution process, it can be easily done). the process of action execution, the actual running of the action execution can be easily migrated to another system for completion).

So we assign each F_e a pipeline for transferring the parameter list (a pipeline can be viewed as a kind of shared memory area implemented by the consumer-producer model.) The MF passes the parameter list to the pipeline in the form of a string after a successful match (it should be noted that the data sources marked with $ symbols in the string are replaced numerically before the parameter list string is sent). All pipelines are uniformly listened to by a single thread(action executor, AE), the received parameter list is passed to the corresponding F_e, and the F_e is uniformly executed by a pool of public threads. In this way, we simultaneously limit the system resources that can be occupied by action execution, reduce the system overhead of creating and destroying multiple threads, and 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 execution process of MF is clear, the next question is how the MF is generated from the rules. that is, how the parsing function modifies the existing system when parsing the text of the rules and how it constructs the corresponding matching function.

3.3.1 Parsing the Datasource, creating foreign symbols

The first step in generating an MF is to parse the Datasource field and construct the OST.

3.3.2 Parsing the Condition, determines the matching condition category, and creates a Token List

After that, the Condition field is judged by its category, and according to its category, different parsing functions are called to generate the $Token$ list. For the expression type condition, the string expression is first parsed into a middle-ordered $Token$ list, and then the middle-ordered $Token$ list is converted into a back-ordered $Token$ list, so as to allow F_m, the expression evaluation algorithm based on the $Token$ Stack, to perform the calculation. For the functional type condition, it can be directly parsed into a $Token$ list.

3.3.3 Parsing Action, create Action List

The parsing of the Action field is easy to do, and only requires parsing the action type identifier, which we form a structure E{action_type,params} with its corresponding parameter list. All of these structures together form a list, which we call the Action List(AL).

3.3.4 Generating matching function and pass the resulting parsed object into the function block as a closure free variable

Pass the OST, ConditionType, $Token$ list, and AL as arguments to the matcher generator function, whose return value is the MF corresponding to the rule. The flow of the matcher generator function is as follows: it creates and returns a function with no inputs or outputs as follows.

1. Use the OST to query the DSB to get the latest data on the symbols and build the IST from that.
2. After that, use Condition Type to index the Matcher Map and call F_m with the IST and token list as input parameters, and its return value as a condition for determining whether the pattern match is successful or not.
3. If the match is successful, traverse the AL, perform IST-based symbolic-value substitution of the data source on the parameter list, and forward it to the appropriate pipeline after substitution.

3.4 Scheduling of rules

The corresponding MF exists only for running rules, but the framework allows users to register rules that do not need to run for the time being into the system as well, and defines four states for rules to facilitate scheduling. They are undefined state, inactive state, booked state, and active state. The following rule state transition relationship is also defined. In addition, we start each AT and AE at system startup to support the running of Matching Thread (MT). We will also implement our scheduling logic with the help of an active rule map, a scheduled rule map and Rule Database(RDB). Figure 5 illustrates the relationship between the state transitions of rules and the scheduling interface.

3.4.1 Create Rules: undefined to inactive

The rule is created and saved to the RDB with the rule state initialized to inactive.

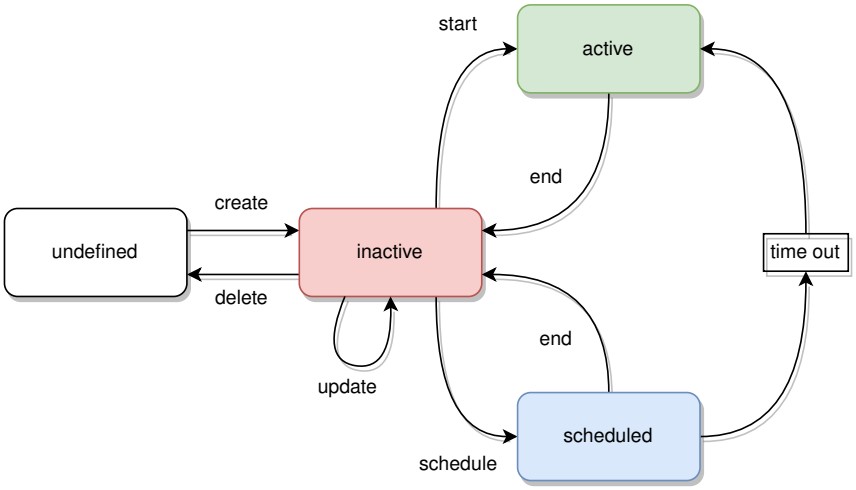


Figure 5: Scheduling Logic

3.4.2 Start Rules: inactive to active

For rules in the active state, a single run of the MF process completes the matching of an up-to-date data with the rule, which is required to run continuously, so we utilize an infinite loop of running subthreads(matcher thread, MT) to periodically call the matching function. From a practical point of view, Cron[12] timed task framework meets our application requirements, so we generate a global Cron object and use this Cron object to execute all matching functions. The specific approach is very simple in practice, just pass the execution period and the generated matching function into the Cron object.

The rules in RDB are taken out, Datasource is pre-parsed and corresponding entries are added in DSB (increasing the reference count of the target datasource and creating the datasource if it doesn't exist) so that AT can update the latest data. After that, the rule text is parsed into MF by the parsing function and added as a timed task to the Cron entity as a matcher task, and the Cron entry id is saved to the active rule map with the Rule Table id (RID) as the index. Finally, the rule status is updated to active. Figure 6 shows the flow of activating a rule.

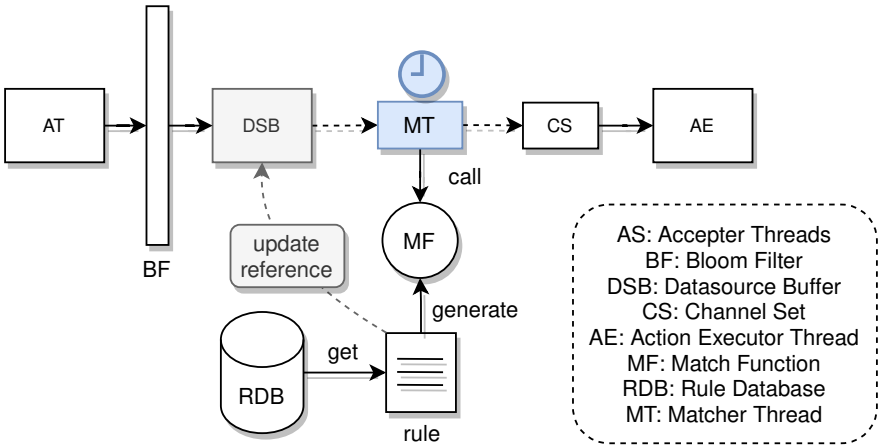


Figure 6: Start Rule

3.4.3 Schedule Rules: inactive to scheduled and schedule to active

Create a new timer, Timer, which is added to the scheduled rule map, indexed by Rule Table id. when Timer arrives, it will call the Start Rules operation and remove Timer from the scheduled rule map, updating its status to active. the rule status is updated to The rule status is updated to scheduled.

3.4.4 Update Rules: inactive to inactive

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

3.4.5 End Rules: scheduled or active to inactive

Parses the Datasource and deletes the corresponding entry in the DSB (reduces the reference count of the target datasource, and deletes the datasource if it is reduced to 0).

If the operation object is a rule whose status is scheduled in the RDB, use its RID as the index, fetch and delete the entries in the scheduled rule map, and stop the fetched Timer.

If the operation object is a rule whose status is active in the RDB, use its RID as an index to fetch and delete entries in the active rule map, and use the fetched object entry to remove the corresponding task in the Cron entity.

Finally, the rule status is updated to inactive.

3.4.6 Delete Rules: inactive to undefined

Rules with a status of inactive in the RDB are deleted.

4 Evaluation

We build a prototype system for evaluating the effectiveness of the framework proposed in this paper.

4.1 Prototype system with proposed rule engine framework

We first analyze the theoretical application scenarios of the rule engine framework in IoT systems, and then build a prototype system based on this for testing.

4.1.1 General application model

Data from IoT devices is accepted via AT, with the option of storing it in MongoDB via historical data while updating the DSB with real-time data. the MQTT Broker is used to issue control commands to the device, and the device only needs to listen to a specific MQTT topic. using MySQL as the main service database (which also serves as the RDB). Figure 7 shows the general application model.

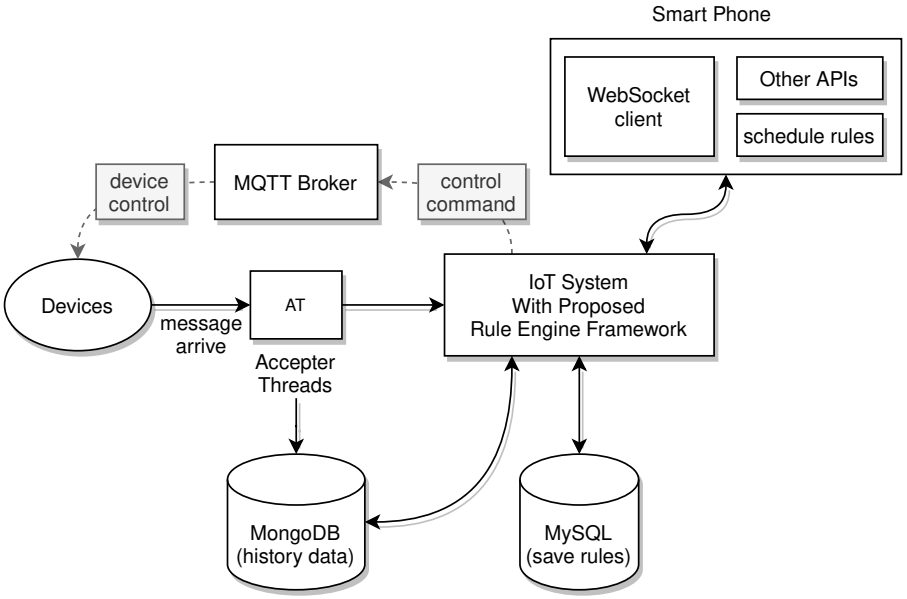


Figure 7: General Application Model

4.1.2 Actual prototype system

We simulate an IoT device uploading data by writing a Python script for a TCP client that sends data to the server and implementing an AT (TCP server) that receives data using the TCP protocol. Use the MQTT client to listen to a specific topic to simulate the device receiving commands. Use a WebSocket client to establish a connection with the prototype system to simulate a user-side smartphone receiving information pushed from the rules engine (including alarms). Use Swagger UI web page to call the REST API interface provided by the system to test our framework and simulate user actions.

In our prototype system[1], F_e for pushing MQTT messages and F_e for pushing messages to WebSocket clients are implemented, as well as a set of other interfaces (non-regular interfaces) that support the operation of the system, including providing the interfaces for WebSocket connection services. Figure 8 shows the actual prototype system for testing.

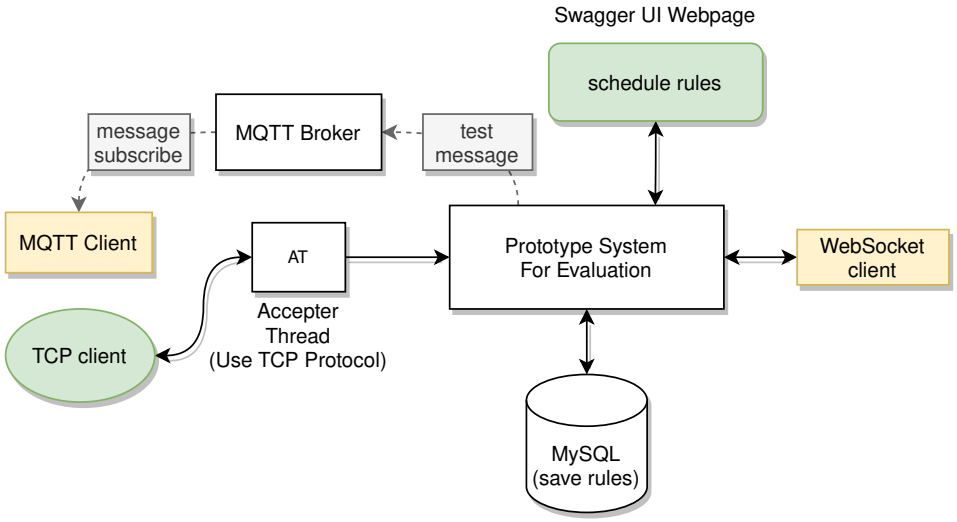


Figure 8: Prototype System For Testing

4.2 Test case and result

Below are the test cases that we used for testing in our prototype system, all the test cases got results as expected, which shows that our proposed framework was validated successfully.

4.2.1 Verify Single device rule with expression condition and multi actions

Rule 1:

- Datasource: tem1, Portable, temperature
- Condition: tem > 22.1
- Action: WebSocket: 1,rule Matched, temperature is $tem!;Mqtt: localhost, 1883, admin, emqx@123456, test, control temperature

A test script written in python sends different messages to the server, of which only some fulfill the conditions defined by the rule. We got the results as expected.

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 test results for the following two rules are similarly as expected, and the test logic is similar to this rule (dynamically change the message body of the reach, and observe whether the MQTT client and the WebSocket client receive the expected message).

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

4.2.2 Verify multi-device linkage rule with expression condition and multi actions

Rule 2:

- Datasource: tem_11, Portable, temperature; tem_21, 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

4.2.3 Verify functional condition with multi acitons

Rule 3:

- Datasource: longitude3, Portable, longitude; latitude3, 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 function of the PointSurface functional condition used in Rule 3 is to determine whether a point is in a surface. We use this rule to determine if the target device is in the given range. Starting from the third parameter and working backwards, we fill in the longitude coordinates and latitude coordinates of the points that enclose the polygon of the given range (the coordinate points are taken from the Amap API). In order to protect privacy, in the paper we have blurred the specific coordinate points in the test cases, using x instead of some numbers.

4.2.4 Verify multi rules concurrent running

In order to test whether multiple rules run concurrently to get the expected results, we first tested Rule 1, Rule 2, and Rule 3 concurrently using the four REST API interfaces of the scheduling rules, and the predefined state transition relationships and transition conditions behaved as expected. Subsequently, we added more rules to test simultaneously and still got the expected results.

5 Discussion

We have verified that the theoretical approach to constructing the framework is sound, and we proceed to discuss the framework's validity, limitations and feasible countermeasures, as well as the application scenarios in which the framework is applicable.

5.1 Effectiveness

Reviewing the research background and research objectives analyzed in the beginning part of the paper, and discuss the effectiveness of the proposed framework in the context of theoretical approaches and test results. First of all, our proposed multi-device monitoring rules with runtime changing data flow have been experimentally verified. Therefore, we mainly discuss generalizability, ease of use and extensibility here.

5.1.1 Easy to embed into almost any IoT system

Proposed framework is easy to embed into the following mainstream structures of IoT system.

- Resource Shortage System
  Proposed framework does not incur a large memory overhead in building the rule engine as in the case of the RETE algorithm, and can still be used in resource-constrained systems. The full details of the framework can be implemented as long as the deployed system supports multithreading. If the lower machine only supports single-threaded operation, the framework can be implemented in the upper machine software.
- Distributed Systems
  It is only necessary to keep the individual components less coupled, and the modification effort to accommodate the distributed system is not significant. For example, for the DSB, it is only necessary to maintain it in a separate microservice and provide interfaces externally for updating, deleting, and querying the data source.

Another component with high resource overhead is the AE, which we can also maintain in a separate microservice and provide a set of interfaces to receive a list of action parameters.

- Systems dependent on third-party services
  An IoT system that relies on a third-party service means that certain devices in the system do not upload messages directly to the system, but upload messages to a server maintained by the device manufacturer, which in turn serves HTTP data requests to the device users. And we only need to implement an AT that polls the third party service to access the data to the DSB.

5.1.2 User-friendly

The rule syntax of Drools is based on the Java language, which has a certain threshold for users, while the rule writing method of the Thingsbaord rule engine is based on modular UI and JavaScript language, which is a little less difficult to use compared to Drools. The framework proposed in this paper uses a specially designed DSL to deploy rules, the DSL contains only three statements, and the syntax is easy to understand, and is easy to grasp by people who do not have computer expertise. At the same time in the framework for scheduling rules in the REST API logic is clear and easy to use.

5.1.3 Easily expandable

As a sacrifice of user-friendliness, the framework cannot support users to define their own data processing procedures using a general-purpose programming language (in contrast, Thingsboard encapsulates a JavaScript interpreter service and thus supports editing of custom rule nodes using the JavaScript language), but the framework's strong extensibility compensates for this shortcoming. The main components of the framework have low coupling, which is manifested in the fact that there is no direct call between MF, AE, and AT, but rather they depend on each other through indexing or intermediate caching, so F_m, F_e, and AT are all easily extensible.

We refer to the design ideas of some general-purpose programming languages when designing the rule DSL. Therefore the extensibility is also reflected in the rule DSL. For example, we can allow constants to be defined in a rule DSL, and when parsed, they are counted in a constant table along with their symbols and data values, and the constant table is queried for subsequent use. We can also implement chaining of rules (chaining means that the next rule is automatically triggered when a current rule matches successfully, and the next rule is not triggered when the previous rule does not match successfully) by means of a new F_e, which functions by modifying a specific cell of the DSB to a certain number. The specific way to implement chained rules is to have the previous rule use the F_e that updates the DSB, and to have the later rule monitor the DSB cells that were updated by the previous rule.

5.2 Limitations and Feasible Countermeasures

Since MF is invoked periodically by Cron, the trigger condition for pattern matching is that the execution interval of Cron is timed out. When pattern matching is triggered, the corresponding data is read from the DSB. Under the existing framework, it is not possible to determine whether the read data has already been validated once, or to detect whether there is any message omission. If the corresponding data source in the DSB has not been updated when the pattern matching is executed, this can lead to the occurrence of processing duplicate messages. When in the execution interval of pattern matching, if the DSB is updated with two or more consecutive data sources, only the latest one will be pattern matched.

A simple solution is to record the frequency of message uploads for each type of device and set a pattern-matching execution period that is similar to that frequency. However, if there is network congestion or IoT devices are down, the DSB will be updated late and processing duplicate messages will still happen. When a message arrives late due to network congestion, two messages will arrive consecutively in a certain execution cycle afterward, resulting in the occurrence of message misses. Setting a reasonable pattern matching execution cycle can avoid most of the message omissions and duplicates, and the omissions and duplicates of individual messages will not cause bad effects in most application scenarios. However, for some special scenarios, it is necessary to give a replacement scheme for the pattern matching trigger condition.

If you only need to avoid processing duplicate data, you can add session field in both DSB and OST to prevent checking duplicate data. Before each execution of matching, query whether the session of the data source in the OST is smaller than the session in the data source cache; if so, update the foreign symbol table session and execute matching afterward; if not, do not execute matching. This avoids message duplication and detects message misses, but there is no guarantee that no message misses will occur.

If both the need to avoid processing duplicate data and the strict impossibility of accepting missing data, the following idea can be used: the DSB actively pushes the new data to the MT using a pipeline, which is called a push trigger condition. It is mainly used in scenarios where every message needs to be processed. In the system using push-type trigger conditions, it is not suitable to control the execution period, because if the data is not fetched during the execution interval, it will cause the AT of updating DSB to be blocked for a long time. This is achieved by storing a map with the rule ID as key and the pipeline as value in the DSB without storing the actual values. the AT sends the same message to all the pipelines in the map when updating a particular data source. At the same time, a sub-thread that executes an infinite loop is used to invoke MF (instead of using Cron), which uses the rule ID and the table of foreign symbols, listens to each data source pipeline, and whenever data arrives in a pipeline, fills the IST with it, and then performs pattern matching as soon as the IST is filled. At the same time, DSB no longer needs reference counting because the map capacity represents the number of rules for the demanded data source.

When using the push-style trigger conditions described above, it is not recommended to write multi-device processing rules where the frequency of message arrivals varies too much. This is because it will result in a slower build of the IST (the reason for this is that it may take a long time for data from one symbol to arrive before data from another symbol arrives). At this point, if several more batches of new messages arrive, the threads sending data to the pipeline in the DSB will be blocked for a longer period of time due to the fact that the last pattern match did not finish and there is data in the pipeline that has not been fetched, which will affect the updating of other data sources.

5.3 Application Scenarios

Based on the effectiveness and limitations analyzed above, we can conclude that the framework can be applied to a wide range of scenarios. However, it is worth noting that the framework is not very suitable for scenarios with very strict control real-time requirements, because device control is triggered by pattern matching, which is executed periodically rather than continuously listening to a certain data stream, so the timeliness of the control instructions issued may be lost. IoT scenarios with strict control real-time requirements are better off not using a rule engine framework to implement control logic, but using traditional PLC control.

We believe that the rule engine framework proposed in this paper works well in IoT scenarios such as home IoT, agricultural IoT, personal wearable devices, and so on. While the effectiveness may be lost in scenarios with strict real-time control requirements such as chemical production.

6 Conclusion

This paper proposes a framework for an IoT-specific rule engine with user-friendly DSL. the framework is able to update the data flow monitoring policies in it without interrupting the operation of the IoT system. The framework has high scalability and can be applied to a wide range of scenarios, and is particularly effective in scenarios with less stringent real-time requirements for device control. In the IoT data stream monitoring problem, our work does not have the problem of taking up a lot of extra memory compared to the generalized rule engine based on RETE, and has higher flexibility and user friendliness compared to the rule engine of the Thingsboard platform.

In the future, we are planning to further explore strategies that can make the rule engine proposed in this paper run more stably, such as allowing users to assign pattern matching priorities or action execution priorities to rules, or proposing a method for semantic conflict detection among multiple rules when they are in use.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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