Axeda® Platform
Rules Design and Best Practices Guidelines

Reference Guide

Summer Release '11
Version 6.1.3
April 2011

Document Number: Enterprise Rules Design and Practices/Reference/6.1.2.1/0311

Table of Contents

Introduction

The Axeda Platform provides powerful tools for developing custom functionality. Business rules, including Expression rules and State machines, enable you to create and run business logic without programming knowledge. Expression rules can be configured to run based on incoming trigger messages (such as data item or alarm messages), as a state within a State Machine, or from system triggers like user login and logout.

Using these programming tools, you can create a wealth of logic specific to your business needs; however; some of the custom logic you create may result in unexpected side effects.

This reference is intended to be used when you are creating or troubleshooting rules and Custom Objects. The information and guidelines provided here will help you avoid potential issues with the custom logic you develop for Axeda Platform. As a rule developer, an understanding of the Axeda Platform runtime rule processing behavior is important to running rules without issue.

Runtime behavior of rule processing

- The rules engine processes trigger messages, in order, for each asset. If one message takes a long time to process, processing for all messages for that assets are affected.
- If a rule runs based on a value defined in the Platform, the Platform needs enough time to actually save that value in order for the rule to use it in its evaluation.
- Rule actions may cause a rule to run again, resulting in an undesirable recursive situation
- Rule actions may get stuck or may take a very long time to complete, thereby preventing the Platform from running the next rule for the related message.

The following sections explain how to ensure your rules avoid potential issues, using the runtime behavior as guidance.

Things to Consider when Creating Rules

Sequential data and event processing

If a single asset sends multiple trigger messages, the system will process those triggers in the order in which each was received. This means that the rules engine will process one trigger message per asset at a time.

If your asset sends multiple values for a single data item in different messages, the system uses the timestamp of each message to track the order in which the data items are processed.

For example, assume asset1 sends dataItem1 in messageA, and again in messageB and messageC. In this example, the Platform will process dataItem1 from messageA, then messageB, and finally from messageC.

On the other hand, if your asset sends multiple values for a single data item in a single message, those data items will not have unique timestamps and so the system may process them in any order.

For example, if messageA contains five values for dataItem1, the values provided in that message may be read and processed in any order. Although the asset sampled the data at different times, only the timestamp of the message as a whole is sent to the Platform. If you consider the case where a data item is meant to represent a sensor value, it doesn't make sense to allow multiple, and potentially different, values for a data item at the same instant in time.

What can you do?

Make sure your assets do not send multiple values for the same data item in a single message. Assets should send only one value for each data item provided in a message

Timing issues

The Platform requires a short period of time to save data. (The exact amount of time varies based on system load and currently processing message.) If you create a rule or action that needs to evaluate the value of a Platform object, it's important that the Platform has had adequate time to save that object first.

For example, consider that Rule1 operates by setting **Overflow_valve** to 1, and then Rule2 runs whenever **Overflow_valve** is equal to 1.

```
Rule 1:
Trigger: Alarm
IF: Alarm.state == "STARTED"
THEN: SetDataItem("Overflow_valve", 1)


Rule 2:
Trigger: Data
IF: DataItem.TEMP.value == 1
THEN: ExecuteCustomObject("SendTweet","user","password","Overflow
started")
```

If the server does not have time to save the results of Rule1 before Rule2 runs, the value of **Overflow_valve** might be out of date and Rule2 may not run. In this situation, there's no guarantee that the value for **Overflow_valve** from Rule1 has been persisted at the time that the newly-created 'internal' server event from the *SetDataItem* action runs.

Also consider the situation in which a single asset sends an excessive number of data values at the same time. For example, maybe you have an asset that comes online only once a week and sends all accumulated data values at once. All events defined for the asset's data items are processed sequentially. The spike in data may cause a backup and the overall processing time for that asset will increase. (Note that this issue affects only the specific asset.) As described above, a backup such as this may mean that the system will take longer to save values.

What can you do?

You can create rules that depend on the outcome of another message or process as long as you space the messages properly in time so that the Platform rules engine has enough time to process a message and store its results before processing the dependent message.

Make sure the Platform has time to finish processing rules and actions, and saving resulting data, before it needs to run other rules and actions which may depend on that data. One possible solution is to define a sleep or wait action as the last action for the first rule. Another solution may to have your assets send less data more frequently, such as every 30 seconds.

Downstream events created from a single event message

A single asset-related message (such as data item or alarm) can create new messages (using actions such as CreateAlarm(), SetDataItem(), and SetAlarmState()), and downstream rules and messages. Running and processing all of the messages and downstream messages and rules may consume an unnecessarily high amount of system resources.

Why is this important?

When Axeda Platform detects that a message spawns many new internal messages through rule actions (such as 'CreateAlarm' 'SetDataItem' or 'SetState') and that the total processing duration from the longest running message sequence exceeds N minutes (by default, this value is 5 minutes), it will automatically stop processing that rule.

Assume the following rules are configured in your Platform and an alarm, Alarm1 (with severity of 100 and state of STARTED), enters the system. As you can see, Rule1, Rule2, and Rule3 all run. Rule4 and Rule 5 will run when a data message for data_item1 enters the Platform.

Rule1

```
Type: Alarm
IF: Alarm.severity == 100
THEN: CreateAlarm ("Alarm30", 200) &&
ExecuteCustomObject("Call_acme_api")
```

Rule2

```
Type: Alarm
IF: Alarm.name == "Alarm1"
THEN: CreateAlarm(ShutDown, 250, "Temperature = "
+str(DataItem.Temp.value)) && EnableRule("Rule99")
```

Rule3

```
Type: Alarm
IF: Alarm.state == "STARTED"
```

```
THEN: CreateAlarm (Alarm20, 100) && ExecuteCustomObject
("CustomObject1") && SendDataItem ("data_item1",30)
```

Rule4

```
Type:  Data
IF:    Data.data_item1.value > 15
THEN:  ExecuteCustomObject("Call_acme_api")
```

Rule5

```
Type:  Data
IF:    Data.data_item1.type == "Number"
THEN:  CreateAlarm(MyAlarm, 250)
```

As illustrated above, a single message for *Alarm1* causes the system to process many other messages and rules, consuming system resources as it does so.

System self-protection

To prevent a single event from consuming so many resources and negatively affecting performance, the Platform tracks the time consumed while processing a single event. (This is a property setting in the server.) If the process is still continuing when the defined time limit is reached, the Platform shuts it down. In the above example, if the time period for a single event was set at 2 minutes, the Platform would stop processing the rule after two minutes. Rules that would run based on that message would be disabled to prevent excessive resource consumption.

If the time exceeds a specified threshold (defined by the server administrator), the system halts processing rules for the original message. The descendent or downstream messages will stop.

So, in the above example, assume the threshold is defined at 2 minutes and the total time required to process all rules for the Alarm1 message reaches 2 minutes just before Rule4 is run, then both Rule4 and Rule5 will not run. (Assume that Rule2 and Rule3 finished processing within the 2 minute timeframe.)

Several indicators are provided when the processing for a rule is terminated: a notification is sent to the server administrator (as defined in the server properties) and an Audit message is displayed in the Audit page in the Administration application and/or the Service application - Asset dashboard for the related asset.

What can you do?

Avoid writing rules that spawn an unlimited number of new messages. Evaluate each rule and consider if it can be broken out to other messages or otherwise streamlined so as to be able to run within the defined timeframe.

After modifying the rule, you will want to re-enable it. You can do this in the Axeda Applications, Configuration application - View and manage Expression Rules page.

Note: When you re-enable a rule, the Platform will reset any metrics or statistics collected by that rule. The execution policy violation icon shown for the rule in the Configuration application is removed when you re-enable the rule.

Time spent processing a single event

Axeda Platform can take so long processing a single event, such as a single expression rule, that its ability to process other rules and events for the same asset is severely impaired. Not only does this "log jam" affect the rules engine, but on very busy systems it may also affect processing of incoming asset messages.

If left unchecked, the resources spent processing a single event can prevent the Platform from processing other events for the same asset. Consider an example for which Rule1 is configured to run CustomObject1. If CustomObject1 takes a long time to process or has issues completing, the system can't finish processing that rule and it won't start another rule until that rule has finished.

System self-protection

To prevent a single event from consuming so many resources and negatively affecting performance, the Platform tracks the time consumed while processing a single event. (This is a property setting in the server.) If the process is still continuing when the defined time limit is reached, the Platform shuts it down. In the above example, if the time period for a single event was set at 2 minutes, the Platform would stop processing the rule after two minutes.

Several indicators are provided when a rule is terminated: a notification is sent to the server administrator (as defined in the server properties), and an Audit message is displayed in the to the Audit page in the Administration application and/or the Service application - Asset dashboard for the related asset.

What can you do?

You will want to evaluate the defined rule and consider if it can be broken out to other events or otherwise streamlined so as to be able to run within the defined timeframe.

Sometimes a rule may be "stuck" and may never finish, for example because it is waiting on system resources that are not available or because it is running a Custom Object that takes too long to process. If this is the case, the Platform will terminate the rule after a period of time (as defined by a server setting), and will create an Audit log entry and send notifications of the action. When reviewing the rule, you should consider if the configuration is causing it to get "stuck" and make any required changes.

Note: When you re-enable a rule, the Platform will reset any metrics or statistics collected that rule. The execution policy violation icon shown for the rule in the Configuration application is removed when you re-enable the rule.

Rules running too often or too long

Axeda Platform provides server administrator tools to monitor the rule processing load of each node in a Platform system. This load is defined by the frequency at which all rules run on a node, and the elapsed time that the rules have run for a given period of time. Several tuning parameters are provided to help the administrator determine when rule processing is causing problems or has the potential to cause problems. These parameters comprise the *rule execution policy* for the node.

System self-protection

The Platform administrator can specify the maximum number of rules that the system can process in a defined timeframe without causing problems. If all the rules on an application server node collectively run too frequently or take too long to execute, based on the administrator set execution policy, the system will find the biggest contributor to the violation and disable that rule. Information will be shown to you in the *View and Manage Expression Rules* page or *View and Manage State Machines* page.

The server property, **RuleFrequencyPeriod,** defines the time period the system will monitor; the property, **RuleFrequencyThreshold,** defines the maximum number of rules that can run within that time period. For example, if RuleFrequencyPeriod defines 2 minutes and RuleFrequencyThreshold is 6000 then the system can process 6000 rules within each 2 minute period without a violation. If a 6001st rule runs in that time period, the Platform determines which rule ran the most frequently during that two minute period and disables it. The logic is identical for execution time violation checks, except that the average rule execution time for the period is used to select the rule to be disabled. For example, if during the two minute period a 6001st run ran, the system would find and disable the rule that ran the longest.

Several indicators are provided when a rule is disabled: the rule is shown as disabled in the Axeda Applications, in the Configuration application pages for viewing rules; a notification is sent to the server administrator (as defined in the server properties) and to the user who last modified the rule; and an Audit message is displayed in the Audit page of the Administration application and/or in the Service application **-** Asset dashboard for the related asset.

Depending on the situation, at times one or more rules may run too frequently and therefore cause the rule execution policy to be violated. For example, perhaps on the first day of each month numerous new assets register at the same time. If the Platform is configured to run rules based on asset registration, the execution policy may be violated. Your Platform administrator can modify the execution policy to handle cases such as this.

What can you do?

If you find that a specific rule runs too frequently, consider how often it truly need to run, ensure that the rule is defined only for the required assets and ensure the rule is defined only for required message(s).

Recursive expression rules

Recursion for expression rules means that the condition in a rule is created either by an action in the same rule or by an action in another rule in a sequence, thus causing that rule to run over and over again, potentially in an infinite loop. The Platform will monitor rule processing for recursion, including rule recursion that occurs as part of running states for a state machine. For example, the Platform will determine if a rule is running recursively within a single state or even across multiple states.

Example of recursion for a single rule

The following is an example of an expression rule defined for an Alarm type.

```
IF: Alarm10.severity == 50
THEN: createAlarm ("Alarm10", 50)
```

When this rule runs, it will create another alarm of the same name and severity, which causes the same rule to run over and over again, in an infinite loop.

Example of recursion for a sequence of rules

Consider the following sequence of rules:

Rule1 - defined for Alarm type

```
IF: Alarm20.severity == 100
THEN: createAlarm ("Alarm30", 200)
```

Rule2 - defined for Alarm type

```
IF: Alarm30.severity == 200
THEN: createAlarm ("Alarm40", 300)
```

Rule3 - defined for Alarm type

```
IF: Alarm40.severity == 300
THEN: createAlarm ("Alarm20", 100)
```

Rule1 causes Rule2 to run, Rule 2 causes Rule3 to run, and Rule 3 causes Rule1 to run. As you can see, this sequence of rules is recursive. When it detects this recursion, the Axeda Platform will disable Rule1 (but NOT Rule2 or Rule3).

System self-protection

When it detects recursion, the Platform disables the expression rule whose condition is repeatedly being evaluated and recreated. In addition, if a rule creates a message that generates a sequence of other rules that eventually generate the first rule again, the Enterprise server will disable that first rule in the sequence.

Note: Recursion detection is performed by an expression rules monitor that can be enabled and configured through properties in the main configuration file for the server. If this monitor is disabled, the server will NOT detect recursion nor will it disable expression rules automatically.

Several indicators are provided when a rule is disabled: the rule is shown as disabled in the Axeda Applications, Configuration application pages for viewing rules; a notification is sent to the server administrator (as defined in the server properties) and to the user who last modified the rule; an Audit message is displayed in the to the Audit page in the Administration application and/or the Service application - Asset dashboard for the related asset.

What can you do?

To fix recursion, modify the ***If*** condition for the rule and include more conditions so that the rule will run once rather than repeatedly. When you enable the rule, any metrics and statistics tracked for the rule are restarted. If you re-enable the rule without actually fixing the recursion problem, or without fixing the problem effectively, the server will end up disabling the rule once again.

If the Platform finds that a single rule is recursive, try renaming the alarm that is created by the action. If the Platform determines that recursion has occurred for a sequence of rules, review each action in a rule and modify any action that causes rules to run recursively.

Queued event processing and management

Agent messages sent to Axeda Platform enter through the Platform's Ingress queue. Commands or data to send to agents are processed through the Platform's Egress queue. This system of queuing and processing messages operates without issue when the message load is steady and typical; however, when there's a sudden surge or spike in messages or the system, encounters problems processing a single queued message, the queues fill up and the messages may temporarily back up.

Axeda Platform provides queue management tools to help ensure the system runs in the most efficient manner. Queue monitor processes manage the queues and ensure that the system does not get overloaded. Server properties are provided to specify the maximum number of messages each queue may hold. When the Ingress queue has reached its maximum, it will stop accepting additional trigger messages until the number of queued messages drops below the defined limit. When the Egress long-term, per asset queue has reached its maximum (that is, 10 messages), it will start deleting older messages so that the queue will always contain the most recent messages to send to the agent. The Egress per asset queue cache will hold egress messages for 100K assets at the time. When egress messages are sent for assets that are not in the Egress cache, messages for the assets that have been in the cache for the longest period of time are removed, based on the 'Least Recently Used' (LRU) caching principle. A warning message in the drm.log file is shown when this happens.

What can you do?

If you find that events are overloading the queue, you should consider configuring your agents to send messages at different times so they are spaced out over time and don't enter the Platform at the same time. This will give the Platform more time to process each message.

Best Practices

When writing expression rules and custom objects, keep in mind the multi-threaded nature in which they are called. Expression rules and custom objects are run sequentially for each message per asset and are run concurrently across assets. The same rule or object may be invoked multiple times at the same time. Different rules and objects may be invoked concurrently. While rules may run concurrently for multiple messages, understand that multiple rules for a single asset message run *serially* and multiple rules for the same event run *serially*.

The trigger messages that cause expression rules to run(data items, alarms, uploaded files, registrations, and other triggers) happen concurrently. The trigger messages cause expression rules or custom objects to run. Messages are processed concurrently for each asset so race conditions may arise when you apply a rule that executes a custom object to multiple assets. You need to ensure that your custom object code does not contain shared states that can be altered by the different threads.

The best way to deal with the concurrency is to keep rules and objects stateless, independent, and tolerant of exceptions that could be caused by race conditions.

The following instructions are displayed in the notifications sent when the system disables a rule.

- *Limit the number of assets associated with the rule* - only associate the required assets to a rule; don't associate all assets to your rule.
- *Limit how often a rule runs* - one way to do this is to limit the number of messages entering the Platform to only those necessary to your needs; another way is to ensure the rule will only run when absolutely necessary by carefully designing the If evaluation statement.
- *Make sure asset associations are used for a reason* - ensure that all your asset-asset associations make sense and are truly necessary. The Platform evaluates a message for each associated asset by running all associated rules; any unnecessary asset associations result in additional processing.
- *Make sure rule actions don't generate too many additional rules* - for example, if an action for a rule causes several more actions to run, and each of those actions causes more actions to run, eventually the Platform may disable the rule because the downstream messages take too long to process.
- *If the rule runs a groovy script, try to optimize it* - try modifying the Custom Object so that it performs only necessary operations. Also, avoid using SDK objects that query or search high volume information (such as, data items and alarms). For best performance, find objects by ID whenever possible. (See the section, "*Calling the Axeda SDK from Custom Objects".)*

Additional things you can do:

- Catch exceptions that result from creating objects that already exist, even if you have previously tried to find the asset without success (classic race condition). In the catch block, try to retrieve the object again, as another thread may have created it.
- Be aware of the implications for handling of asset data and processing of rules if you are using model or asset associations. When these associations exist, the same data may need to be reported and processed for multiple assets or models, depending on how the associations are set up. The associations can mean that the same rule is running in multiple instances.

- Construct your `If` condition in an Expression Rule as precisely as possible to ensure that the `Then` expression runs only when you really want it to run, especially if the `Then` expression could result in an expensive operation.
- Consider using state machines instead of complex conditions and expression rules that may have expensive `Then` operations.

Things you should NOT do:

- Alter the state of the system such that the change affects the way rules will be evaluated.
- Use calls that are harmful if they happen twice (due to a race condition).
- Query for more data than is absolutely necessary. For example, if you need the most recent data item in a Custom Object, do not use the HistoricalDataFind and "findAll". Instead, use the CurentDataItemFinder.

Inter-dependent rules

Rules should be autonomous and should NOT depend on side effects of other rules. If a need exists to impose an order on the evaluation of rules, consider modeling the problem with state machines or use a conventional approach to ordering rule processing. For example, one rule could fire for a given data item, and then produce a different data item that would generate other rules. Keep in mind the threading for rule execution -- the second stage rule could fire concurrently with the first rule (if it's not for the same asset), once the second data item is produced.

Data Items and trigger messages

A set of data items generate rules to fire once per set (not once per data item). A data item set is intended to be a group of readings for multiple data items at a given point in time. Given this definition of a data item set, it does not make sense to have multiple instances of the same data item in the same data set, even through the system allows it. Each uniquely-named data item can be accessed from an expression rule that is invoked on the set.

Calling the Axeda SDK from Custom Objects

Custom objects can call methods from the Axeda® Platform SDK. (Full documentation of the SDK is provided within the Axeda SDK Javadoc. Also, the *Axeda® Platform API Reference* guide provides a high level introduction to the Axeda SDK.) Not all SDK functionality is appropriate for use in custom objects; Axeda provides recommendations for using SDK methods in a manner that ensures the best performance. This section provides some guidelines when using SDK methods and objects.

Using SDK Finders

Searching for data in the Platform can be expensive and could impact performance. When using SDK finder methods to return data from the Platform, it's best to limit your searches to a subset of objects rather than searching through all data and objects to which you have permissions.

As a means of enhancing data searches, Axeda provides optimized means of performing lookups. Table 1 lists the Finder objects and their efficient usage. These objects have been enhanced for high volume processing by Custom Objects executed from Expression Rules Engine. Most of these objects provide IDs by which you can find them; some of these also provide alternate key caches which can be used for searches.

Any Creates, Finds (other than those listed in Table 1), Updates, or Deletes for the objects shown in Table 1 should occur only occasionally; that is, no more than 10 operations per minute. An exception to this is the HistoricalAlarmFinder, which should occur no more than 1 time a minute.

As much as possible you should avoid needless data lookups; however, when you do need to find data in the Platform, searching by ID will provide for the most efficient searches. Most of the optimized objects provide IDs by which they can be searched. If you do NOT know the ID for an object you are trying to find, you will need to use another key for the object's finder method. Alternate key caches are provided for several of the optimized objects; this is shown in the table.

Table 1 Finders and Optimized SDK Objects

Class	Lookup Usage
CaseFinder	ID
CompositionFinder	ID
ConditionFinder	ID
CurrentDataFinder	find(DataItem), find(DataItemName)
DataItemFinder	ID; otherwise by Model ID and Data Item Name (Composite Key)
DeviceFinder	ID; otherwise, by Serial Number and Model (Composite Key) or Address
DeviceGroupFinder	ID
ExpressionRuleFinder	ID
GeofenceFinder	ID
LocationFinder	ID
MaintenanceItemFinder	ID
ModelFinder	ID; otherwise you can search by Model Name
OrganizationFinder	ID
RegionFinder	ID
RuleAssociationFinder	ruleId
StateMachineFinder	ID
ThresholdRuleFinder	ID
UnitFinder	ID
UserFinder	ID
UserGroupFinder	ID

Example of performing lookups

When searching for objects using these optimized Finders objects, make sure you use exact matches for all string values. The following code snippet shows how you can use the ModelFinder to search for models.

As explained in Table 1, ModelFinder provides an ID for searches. If you do not have the ID for the model you want to find, you can locate the model by Model Name. Both methods have been optimized for use; however, it's suggested that you search by the ID when possible. Note that all string values for these Finder objects are exact matches. For example, when using a model name to search for a model, the model name must be an exact match.

```
ModelFinder modelFinder = new ModelFinder (Context.getContext(), new
Identifier(1));
Model model = modelFinder.find();
```

OR

```
ModelFinder modelFinder = new ModelFinder (Context.getContext());
modelFinder.setName("MyModelName");
Model model = modelFinder.find();
```

Use non-optimized methods with discretion

If your Custom Object needs to use Finder method not listed above make sure you limit the frequency of those search operations.

Getting Help

If you still have questions after reading this guide and the online help for the Axeda Policy Server, you can go to *http://help.axeda.com* for assistance.

Documentation Feedback

As part of our ongoing efforts to produce effective documentation, Axeda asks that you send us any comments, additions or corrections for the documentation. Your feedback is very important to us and we will use it to improve our products and services.

Please send your comments to *mailto:documentation@axeda.com*. Thank you for your help.