Fraud AMDL Rules Configuration Guide

Powered by Featurespace

Version: 1.0

10 May 2024

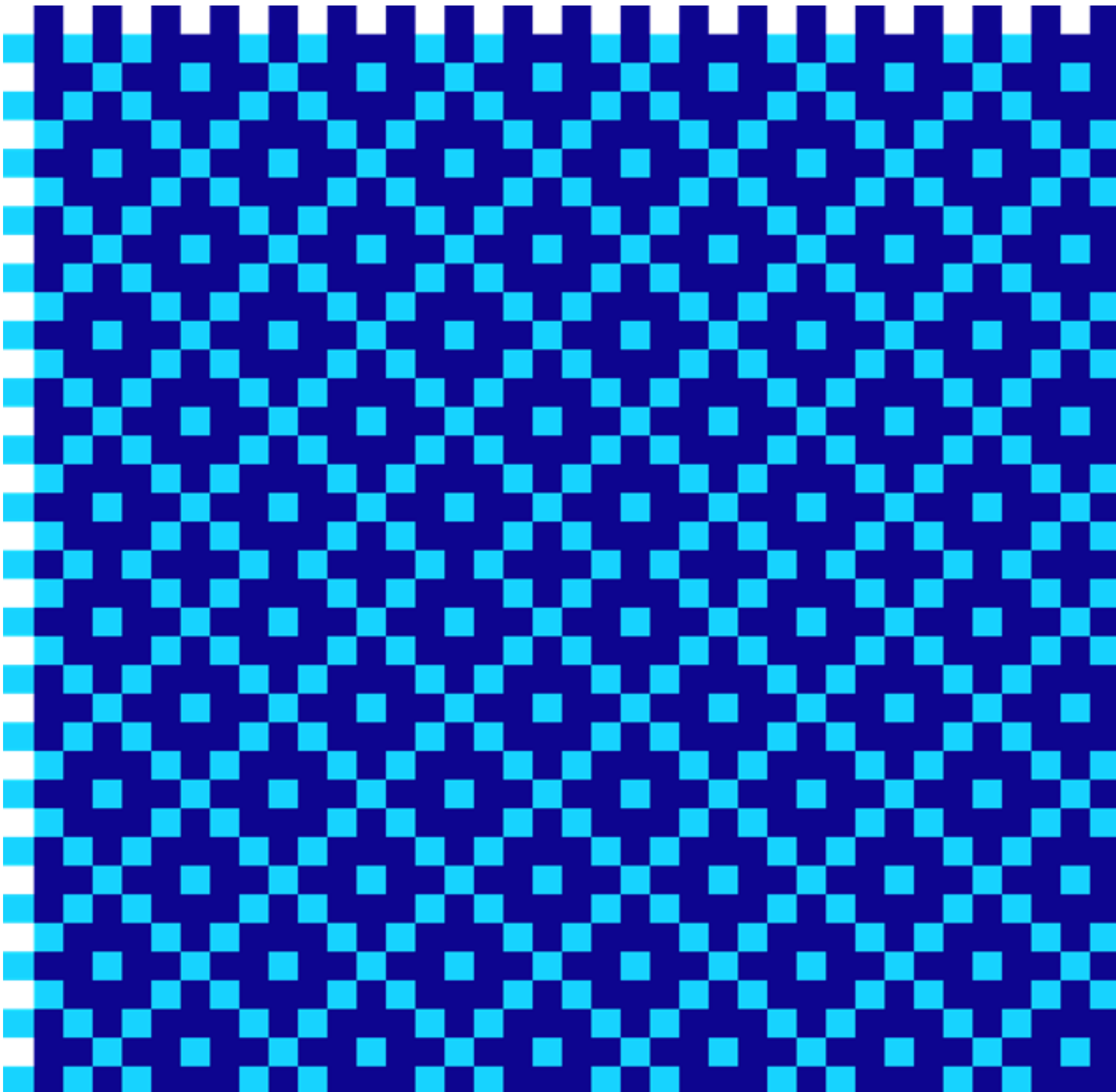
Publication number: FTMC-1.0-5/10/2024

For the latest technical documentation, see the Documentation Portal.

Thredd 6th Floor, Victoria House, Bloomsbury Square, London, WC1B 4DA

Support Email: occ@thredd.com

Support Phone: +44 (0) 203 740 9682



Copyright

About

This document describes how to configure the rules used by the Thredd Fraud Transaction Monitoring System, using the AMDL (Advanced Model Definition Language) Business Rules.

Note: The Fraud Transaction Monitoring System. is based on the Featurespace ARIC system. This guide refers to the Featurespace ARIC Risk Hub User Interface as the Fraud Transaction Monitoring Portal.

Target Audience

This audience for this document isThredd clients (Program Managers) who are using the Thredd Fraud Transaction Monitoring System.

What's Changed?

If you want to find out what's changed since the previous release, see the Document History page.

Related Documents

Refer to the table below for other documents which you use in conjunction with this guide.

Document	Description
Payments Dispute Management Guide	Describes how to manage chargebacks and the disputes management process using Thredd.
Smart Client Guide	Describes how to use the Thredd Smart Client to manage your account.
Fraud Transaction Monitoring Portal Guide	Describes how to use the Fraud Transaction Monitoring Portal to review high-risk incidents flagged by the system, configure the behaviour of the system, and view, manage and understand the various reports and metrics available in the portal.
Fraud Transaction Monitoring System Access Configuration Guide	Describes how to set up user access and user access roles.

Tip: For the latest Thredd technical documentation, see the Documentation Portal.

1 Overview

The Advanced Model Definition Language (AMDL) is a language for specifying rules and logic within the Fraud Transaction Monitoring System. AMDL is a declarative language for specifying state updates and executions on each event that passes through the system. An example of an event is a customer transaction, such as the withdrawal of money from an ATM. Every event contains a reference (for example, an ID field) to one or more entities of different types, such as a merchant and a consumer. You can use AMDL to create Business Rules where you apply the syntax capable in the modelling language.

1.1 About Business Rules

AMDL expressions are defined against a specific entity type in Business Rules. Business Rules can trigger actions such as alerts and tag outputs when a specific event matches a set of criteria. In addition, Business Rules can trigger actions based on a single event. The rules can also build up behavioural profiles of entities and triggers based on a pattern of behaviour across multiple events, or a single event that differs from the entity's profile.

The following outlines the main features of Business Rules:

- **Definition** – You define AMDL expressions for a Business Rule individually, one per tab, within the AMDL Rules and Features Editors in the Thredd Fraud Transaction Monitoring Portal.
- **Outputs** – You can configure expressions for individual Business Rules to generate or suppress alerts and tags when triggered using annotations (see section Using Annotations). For generating alerts and tags as a result of combinations of triggered rules, you can also incorporate Business Rules into aggregators,
- **Models** – You can use model outputs (e.g. risk scores) in Business Rules (see Appendix 3: AMDL Scopes).
- **Grouping** – You define Business Rules for an entity type, and apply these to a single type of entity. As optional, you can restrict Business Rules to only execute on one or more event types (see Rules).
- **Execution** – Business rules follow a particular pattern when executing. They are:
    - executed in parallel
    - grouped by scope
    - executed in a set sequence. Transient variables are evaluated first, then rules are executed, then state and global updates.

      Note that the exception to the sequence is where one expression explicitly references another, Business Rules expressions may be executed in any order, and in parallel.
- **Scopes** – You can use a range of AMDL scopes in Business Rules (for details, see Appendix 3: AMDL Scopes).
- **Annotations** – You can use the full range of AMDL annotations in Business Rules (see section AMDL Annotations).

1.1.1 Managing Business Rules

Business Rules can be created, edited and managed in the **Analytics** section of the Fraud Transaction Monitoring Portal. You can edit Business Rules in the **Rules** page, and feature extraction logic in the **Features** page.

Approving Business Rules

Changes to Business Rules are made as part of the analytics approval process. Your changes are included in a staging analytics version, and must be submitted for review and approved in your organisation before taking effect. Any changes you make will not take effect until that analytics version is live. See the Fraud Transaction Monitoring Portal Guide for more information on using the Fraud Transaction Monitoring Portal. These include information on the analytics approval process, including detailed instructions on how to make changes to analytics, and how to use the Business Rules editor in the Fraud Transaction Monitoring Portal.

1.2 Guide Examples

This guide uses examples to illustrate the AMDL features in the Fraud Transaction Monitoring Portal. However, not all examples are available in the Live environment. Check the available data fields or speak with your Thredd Account Manager for more information.

2 Getting Started with AMDL Syntax

AMDL (ARIC Modelling Data Language) allows you to create a wide-variety of rules for the Fraud Transaction Monitoring System through the Fraud Transaction Monitoring Portal. Before you create rules using the rich AMDL expressions, you will need to understand the prerequisites and gain familiarity with a basic rule.

2.1 Prerequisites

Before using AMDL in the Fraud Transaction Monitoring System, you need to be set up with the user role of **Risk Manager** with the appropriate permissions. A Risk Manager can create and edit rules, as well as push rules to the Live environment.

You should also have some knowledge of declarative programming languages in order to understand how to build and manipulate rules using AMDL.

Before creating rules Fraud Transaction Monitoring System, you need to be set up with the user role of **Risk Manager** with the appropriate permissions. A Risk Manager can create and edit rules, as well as push rules to the Live environment.

Creating fraud rules requires you to use AMDL (ARIC Modelling Data Language). You should also have some knowledge of declarative programming languages in order to understand how to build and manipulate rules using AMDL. However, an indepth knowledge is not required.

2.2 Creating a Rule

A rule is the fundamental building block that contains AMDL expressions. Alerts and tags are the outputs that are triggered from a rule to inform of changes.

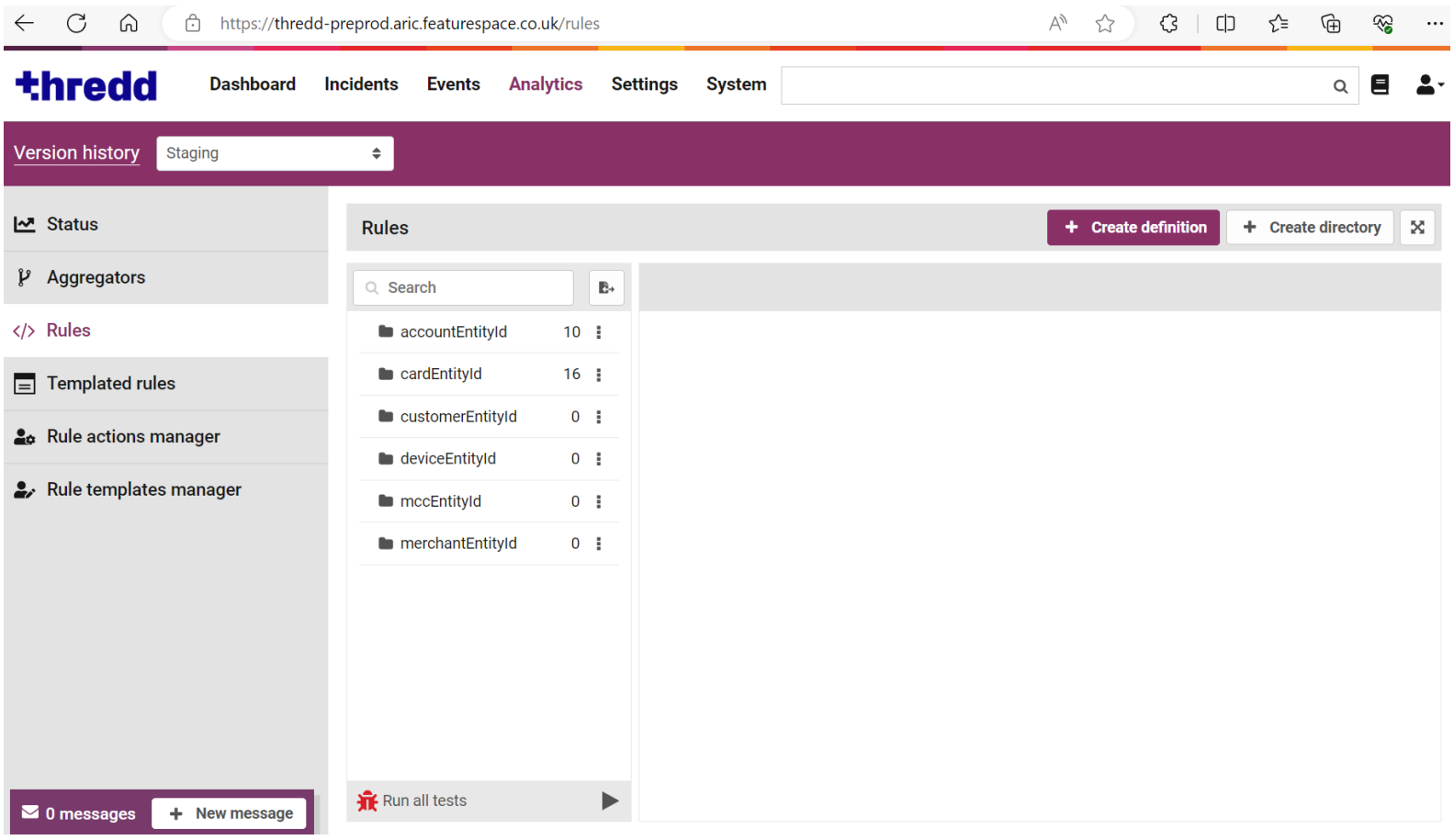
A basic rule consists of various components that is entered in the AMDL syntax. This includes:

- Alerts and tags
- Event type
- Rulename
- Events
- State information (see Adding State Information to a Rule)

This example rule sends an alert when a card transaction above the figure of 10000 is authorised.

```
@alert
@eventType("cardRT")
@eventType("cardNRT")
rules.highValue:
(event.msgType.lowercase() == "authorisation" &&
event.msgStatus.lowercase() == "new") &&
event.baseValue > 10,000
```

Using the Rule Builder interface in the Fraud Transaction Monitoring Portal you can add AMDL expressions. The Fraud Transaction Monitoring Portal also includes a set of for rules, represented as folders. To help you save time in rule building, Fraud Transaction Monitoring Portal includes the **accountEntityId** and the **cardEntityId** folders that contain pre-configured rules.

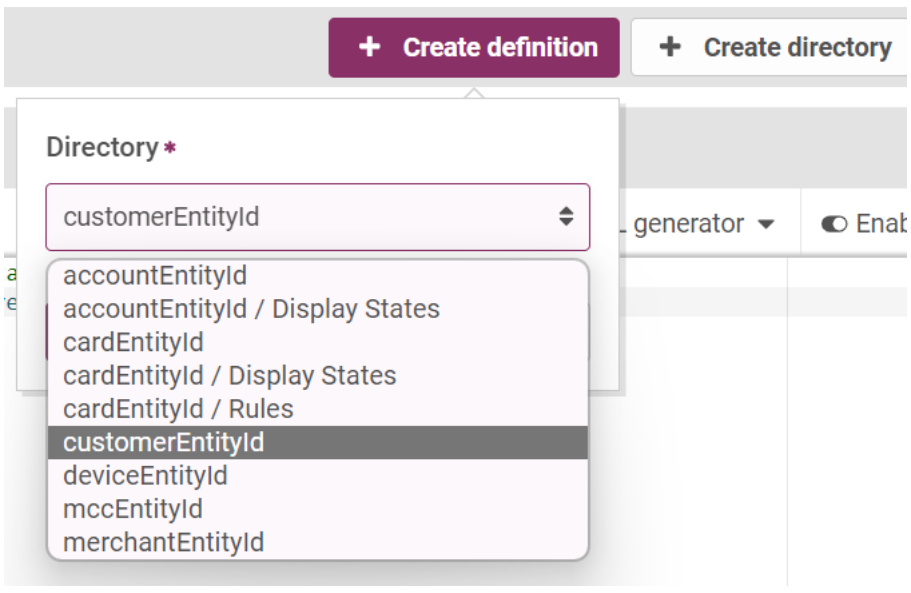


Note: More details on customising rules with AMDL are described in Customising Business Rules in AMDL

2.2.1 Creating a New Rule

You can create a new rule in an existing directory, or you can specify your own directory and its location.

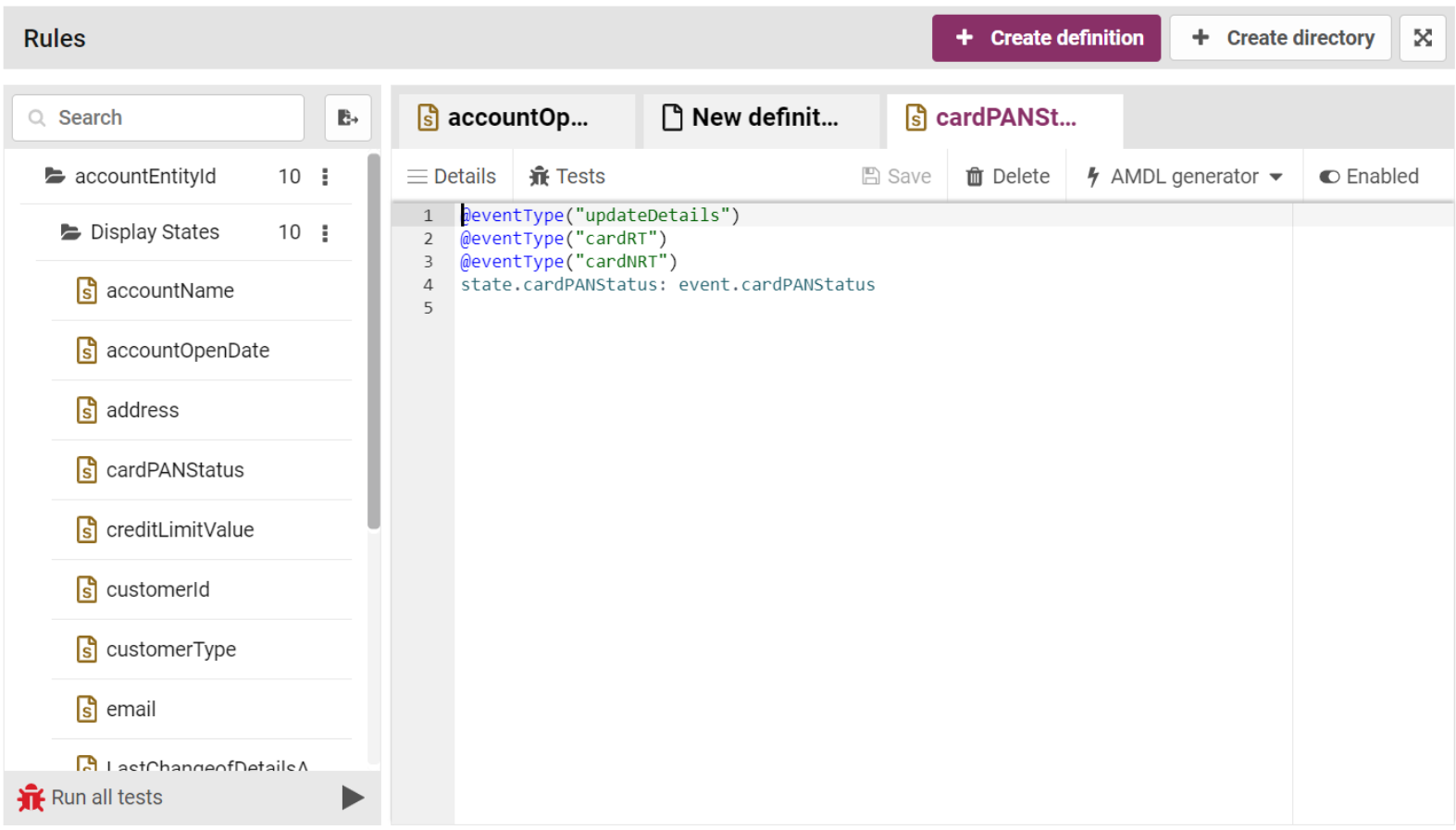
1. Click **+ Create definition**. A window displays.
2. Select a directory for the new rule.



3. To choose a different location, click **Create directory.** Then select a location in **Parent directory** and type in a **Directory name**.
4. Click **Create**. You can then enter your rule expression.

2.2.2 Selecting an Existing Rule

You can expand the folder items for selecting an existing rule to update.



2.3 Alerts and tags

Alerts and tag contain a specific syntax. An alert, represented as `@alert`, flags an event in the Fraud Transaction Monitoring System. Flagging the alert provides a prompt so that a user can prioritise, investigate and review high-risk events. Tags, represented as provide additional information to users on why an alert was raised. They can also trigger automated decisions such as a soft decline (request chip and pin entry at the terminal for card present transactions), a hard decline of a transaction and a block of a card. The tags appear in the Fraud Transaction Monitoring Portal associated with the event or alert they were added to. The tags also exist in the output the fraud system produces in response to a real-time event.

You can alter the above rule to include the following tags:

```
@alert
@tag("High value transaction or account transfer")
@tag(action="BLOCK")
@eventType("cardRT")
@eventType("cardNRT")
rules.highValue:
(event.msgType.lowercase() == "authorisation" &&
  event.msgStatus.lowercase() == "new") &&
  event.baseValue > 10,000
```

This rule produces a tag with the namespace of action and value "BLOCK", and another which displays as the text "High value transaction or account transfer" in the Fraud Transaction Monitoring Portal.

The general form of the @tag annotation is `@tag (<namespace>="<value>","<namespace>="<value>")` You can add more than tag by adding the @tag annotation multiple times. Or, you can add multiple tags using the same annotation:

```
@tag (<namespace1>="<value1>", <namespace2>="<value2>"...)
```

To add multiple tags in the same namespace, use this syntax:

```
@tag (<namespace1>="<value1>", <namespace1>="<value2>"...)
```

If you do not provide a namespace, you can use a default namespace (`_tag`). If the namespace is unimportant, you can write.... The portal displays this tag without a namespace.

2.4 eventType

This is the type of transaction in the fraud rule. This example includes the eventType as real-time and non-real-time transactions, represented as `@eventType("cardRT")` and `@eventType("cardNRT")`. The eventType acts as an annotation where it limits the events that are acted upon in a rule. In this example, the rule does not act on events involving cash transactions. However, the rule acts on card events.

2.5 Rule name

This is the name of the rule used. In this example, the rule is called "highvalue", which has the syntax of rules.highvalue:.

The general syntax of a rule is as follows:

```
rules.<rule name>:<Boolean expression>
```

A rule is an AMDL expression defined in the "rules" scope, and consists of a Boolean expression which is evaluated for each event that passes through the system. Each AMDL expression applies to one particular entity type, for example cards or merchants, and evaluates for each entity of that type in event. If the expression evaluates to true, the rule has 'triggers' on the event.

2.6 Events

An event represents a type of activity. The syntax of an event must match an event schema. Any AMDL expression can refer to data contained in the event payload, using the event.fieldName syntax. In this example, this is the message type, and message status data and base value of the transaction. This is represented as event.msgType, event.msgStatus, event.basevalue. There is also a condition associated with each event, where the message must appear in lower case, must be a new, and be an authorisation for amounts above 10,000. This is represented in the following syntax:

```
(event.msgType.lowercase() == "authorisation" &&
   event.msgStatus.lowercase() == "new") &&
   event.baseValue > 10,000
```

2.6.1 JSON Schema in Events

The Fraud Transaction Monitoring System system is event driven. The following shows an example event, illustrating the structure of the JSON data with the data that is stored in respective data types, for example, date and time in "expiry".

```
{
    "eventId": "a78e098d0bc7",
    "eventType": "transaction",
    "eventTime": "2019-05-05T18:02:55Z",
    "customerId": "3263827",
    "merchantId": "THX1138",
    "customerSegment": "B",
    "merchantCategoryCode": "1234",
    "amount": {
    "value": 100,
    "currency": "EUR",
    "baseValue": 85.70,
    "baseCurency": "GBP"
},
    "deviceData": {
    "deviceId": "a85531c1-02d8-44ed-964f-0706155209c7",
    "deviceType": "Android",
    "OSversion": "10.0.1",
    "deviceManufacturer": "CunningBadger",
    "deviceModel": "Slow 2.0"
},
    "paymentMethod": {
    "methodType": "card",
    "methodId": "WEA798GHSET98ERGX",
    "methodDetails": {
    "issuer": "Imperial Bank of Wales",
    "BIN": "123456",
    "expiry": "2020-12-01"
}
}"
    transactionType": "CP",
    "accepted": true
}
```

If there are several nested levels of JSON objects within the event, they can be accessed in a similar way to JSON objects, using periods to denote level transitions. For example, to reference the issuer field in the example transaction event above:

```
event.paymentMethod.methodDetails.issuer
```

Alternatively, you can use the square-bracket map accessor syntax, for example, ["level2field"]["level3field"]) to access one or more levels. You may need to do this if the event.topLevelField property names are reserved for AMDL keywords such as "state" or "values", for example, or if the field names begin with numbers:

```
event.paymentMethod["methodDetails"]["issuer"]
```

Note that this syntax cannot be used for root-level fields within the event data, as the AMDL parser expects an event at the beginning of any reference to event data.

Note: When you write a rule, you write it against a specific entity type. This is because the alert that is raised is against the entity of that type.

2.6.2 Entity Meta-Variables

Meta-variables are the state._type and state._id. These contain the entity type and entity ID values that can be accessed in AMDL expressions (in both cases represented as a string). These are particularly useful where there are two entities of a given type within a single event.

For example, if a payment event contains two customer entities, acting as the payer and the payee, it might be useful to write an expression that can distinguish whether the entity being evaluated is the payer or payee. If the event contains a payerId and payeeId field, you can distinguish between the payer and payee entity simply by testing whether the ID of the current entity matches the payer ID or the payee ID.

This rule, for example, only triggers an alert for the payer if they are on a payer watchlist, or for the payee if they are on a payee watchlist.

```
@alert@eventType("payment")
rules.payerOrpayeeOnWatchList:
( state._id == event.payerId && lists.payerWatchList ~# event.payerId )
||
( state._id == event.payeeId &&
lists.payeeWatchList ~# event.payeeId )
```

These meta-variables avoid an alert being generated for the payee entity when the payer is the one on the watchlist, and vice versa. Meta-variables are also useful in cross-entity state references in Business Rules (see Cross-entity State References). This is because they enable an AMDL expression to distinguish between two entities of the same type when accessing state for those entities.

Using the example above where an event contains a payer and a payee entity, an AMDL expression could access the state of the payee entity using a predicate filter. For example, this rule (written for the 'customer' entity type) would generate an alert for the entity that was the payer of a payment if the state expression PEPFlag was true for the payee entity:

```
@alert@eventType("payment")
rules.payeeIsPEP_payerAlert:
state._id == event.payerId&&state.entities.customer[ $._id == event.payeeId
].PEPFlag.single() == true
```

2.7 Adding State Information to a Rule

A state allows a rule to store information on an event. For example, the state can include information on the number of transactions and the total value over a period. A rule uses information for entity states and global states.

2.7.1 Entity States

Entity states persist against multiple events. For example, you can store the date and time of the first transaction, which is then accessible when processing all future events with that consumer.

Alternatively, you can build a history of all the transaction amounts of a merchant over the past month, and access that information in future events. You can update state variables by values in an event and in other state variables. The fraud system automatically initialises an AMDL the first time it is needed for an entity.

You can specify state variables using an update expression. This type of expression provides a single value that is then used to update that state variable. You can use the value of the state expression to update the state variable on every event.

Every state definition requires a single update expression, but how the value of that expression is stored depends upon the type of the state variable. The default type of a state variable is a single value. The state variable of a single value contains the last value it was updated with, where a new one overwrites it each time the state is updated.

In this example, a rule includes the following state information, where there are no card-present transactions in the last 7 days. The rule also includes an event where there are new card authorisations for a value greater than 250.

```
@tag(Action = "Decline")
@eventType ("cardRT")
@eventTyoe("cardNRT")
rules.highEcommTxn
(event.msgType.lowercase() == "authorisation" &&
event.msgStatus.lowercase() == "new") &&
(event.amount.baseValue > 250) &&
(state.cardECommerce7d.size(7d) ?? 0) < 1 &&
!event.pointOfServiceContext.cardPresent
```

You can store a timestamp in an entity state event in the following format:

```
state.lastEventTime: event.eventTime
```

This timestamp stores the state for entities of the type this expression is defined for, where AMDL Business Rules expressions are all defined under a specific entity type.

To refer to the stored value of this state expression for the entity of the relevant type in the current event, you can use the scope "state" and the expression name. For example, to refer to the time of the last event for the current entity:

```
state.lastEventTime
```

IMPORTANT: In Business Rules, when an event is being processed, the state is updated after other AMDL expressions have been evaluated. This means that a rule that references state will use the value of the state as it was before the current event was processed. When referenced within another state update, a state variable is accessed as it was before the current event. This is necessary to preserve a well-defined expression evaluation that it is not dependent upon microsecond variations in the processing speed of each rule. However, this does not diminish the computation power of AMDL because the set of prior state plus the event contents comprises the complete known information about that entity.

2.7.2 Global States

A global state expression defines a single variable that is updated by events for any entity of a given type. This ensures that a global state stores information derived from the whole population of entities. One of the main uses of the global state is comparing the activity of one entity with a threshold or comparison value derived from the behaviour of the whole population.

There may be seasonal variations in average transaction values and volumes. Therefore, comparing transaction values to a fixed threshold might generate a much higher volume of false positives. For example, there are periods of increased spending in the run-up to Christmas, Black Friday, or Singles' Day in China.

Note: You are less likely to update a global state as frequently as entity state. Therefore, when referring to a global expression, the value might be up to several seconds out of date. You can also use global states to track rolling averages and similar aggregate values, but not to track values or collections that need to be completely up-to-date at all times. It is highly recommended that all global variables are declared as an aggregate type, for example, a collection (see section Collections) or a rolling average. See the section Using Annotations for a general discussion of this kind of annotation, or @rollingAverage for a detailed description of this specific type of variable.

You define global state expressions in a similar way. For example, to store a rolling average of transaction value across all merchants, you can define this expression under the "merchant" entity type using the @rollingAverage annotation. For more information on the use of annotations to modify how state is stored, see Using Annotations.

```
@rollingAverage(24h)
globals.averageAmount: event.amount.baseValue
```

This annotation, applied to entity or global state variables, keeps a track of the rolling average within a specified time period. You must specify this time period as it cannot keep track of a mean value over all time (because it is implemented as an exponential moving average for performance and storage purposes).

2.8 State Fragmentation and Size Limits

There are several mechanisms built into the event processing components of the system to prevent performance issues due to large amounts of state or global data. These include state fragmentation and limits on the amount of data that can be stored in individual state/global variables, the size of the state for an individual entity, and the total size of global state.

2.8.1 State Fragmentation

State fragmentation is an optional feature that you can enable for specific entity types. Fragmentation allows the fraud engine to break up an entity's state into parts, each of which can be loaded and updated independently. This ensures that the engine does not need to load the entirety of an entity's state when processing an event. State fragmentation results in significant performance benefits for entities with large amounts of state data. State fragments are recombined preiodiacally, and any changes are written to the fraud system datastore.

Fragmentation provides performance benefits when entities of a particular type are likely to have large amounts of stored state. For example, fragmentation exists for an entity type with few unique entities, where each entity occurs in a large proportion of events (such as an entity type like "country" or "merchant category"). Ask your Thredd Account Manager for more information about state fragmentation and how to enable it for the entity types.

Because fragment updates are only recombined periodically, the state for fragmented entities (including all global states), is not guaranteed to be consistent after every event, although it is eventually consistent. For events that occurred close to each other in time, state, global expressions and/or rules that rely on events being processed in strict chronological order may show unexpected behaviour.

Note: Global state is treated as a single global entity that is present in every event, and this entity is always fragmented.

2.8.2 Size Limits

The fraud system sets limits on the size of an individual state variable, and the total size of state stored for an unfragmented and fragmented entity. As a global state is treated as a single fragmented entity, the limits on the total size of state stored for a fragmented entity also apply to a global state.

Each of these types of states has a warning level and a hard limit. If the size exceeds the warning level, an error message is written to the AMDL error log, and a warning icon is displayed in the Rules Management page, next to the relevant expression on the Fraud Transaction Monitoring System. For an individual variable limit, if an entity's state or global state exceeds the warning level, a warning icon does not appear. If the size exceeds the hard limit, no further data is written to the relevant variable, entity, or to the global state. These limits are set to the following default values but can be configured (for details, please contact your Thredd Account Manager).

Type of state	Type of variable	Warning level (kB)	Hard limit (kB)
Entity state (fragmented or unfragmented)	Individual variable	60	100
Unfragmented state	Total entity state	200	1000
Fragmented state	Individual global variable	100	500
	Total entity state (or total global state across all entity types)	1024	4096

3 Using AMDL Expressions

This section shows you how to use various syntax in AMDL expressions for creating rules.

Syntax	Description
AMDL Expressions	The basic building blocks used in AMDL expressions.
Comments	How add comments into your AMDL.
Conditions	How to use Boolean (true or false) expressions in your AMDL rules.
Using Annotations	How to change when an expression is evaluated.
Static Values	How to use the AMDL "values" scope to define static values. A static value is a threshold value used in multiple expressions.
Transient Variables	How to use variables that are re-calculated for each event. For example, transient variables can include exchange rates.
Conditional Assignment	How to conditionally assign a value or update a state expression. For example, this can include the value of a transaction that meets a condition.
Dates, Times and Durations	How to compare times and use AMDL logic on the elapsed time between two events.
Collections	How to use data structures that contain multiple values. For example, this can include a list of merchant category codes.

AMDL has features that enable you to use advanced AMDL functionality for maintaining rules. These include:

Functionality	Description
Manipulating Strings	Manipulate strings using either concatenation or regular expressions (regex).
Specifying a Default Value Evaluating if an expression returns a non-null value Storing the first value for an expression	Manipulate the values used in an AMDL expression.
Using Histograms	Analyze time-series data using histograms.
Using Lookup Tables Using Data Lists in AMDL	Map values using lookup tables and data lists.
Filtering Collections Iterating over Elements in a Collection	Work with collections using AMDL. Filtering collections involves predicate filters.

Further Information

For more information about:

- The syntax used by AMDL, see Getting Started with AMDL Syntax.
- How to use AMDL to write rules, see: Getting Started and Writing Business Rules in AMDL.
- How to write tests to check your AMDL rules, see Unit Tests for AMDL .
- Details of the types, scopes, methods and annotations in AMDL, see the Appendices.

3.1 AMDL Expressions

AMDL includes expressions, which are the basic building blocks of Business Rules. The general form of any AMDL expression is as follows:

```
@annotation
scope.name: definition
```

An AMDL expression consists of a scope, name, definition and, optionally, one or more annotations:

- **scope** — the scope of the AMDL expression, denoting the type of expression that is being defined. Some scopes allow you to define expressions, while some are only for referencing values. Appendix 3: AMDL Scopes lists all the possible scopes you can use in Business Rules. For example, you can define a scope for a state or for lists.
- **name** — the name by which the definition is referenced in output and other AMDL expressions. A name must be unique within a scope, but can be shared by AMDL expressions of different scopes (for instance, `state.foo` and `values.foo` could both distinctly exist).
- **definition** — the value to which the AMDL expression evaluates if run.
- **annotation** — additional information about the AMDL expression that can influence its evaluation behaviour, which are optional. AMDL expression can have no, one, or several annotations. For more details. refer to Using Annotations and Appendix 4: AMDL Annotations (which lists all the annotations that can be used in AMDL).

3.2 Comments

You can write comments in AMDL using `//...` and `/*...*/`. These are ignored by the parser. `//` ensures that everything remaining on that line is interpreted as a comment. `/*` ensures that everything is treated as a comment until a closing `*/` is found.

```
// This is a single-line comment
rules.thisIsAMDL: event.amount.baseValue > 100 // This is a comment too
/* This is a multi-line comment.
This is still part of the comment.
So is this. */
```

3.3 Conditions

In AMDL, a condition is a Boolean expression that evaluates to either true or false.

You can use conditions in various types of AMDL expressions. A condition consists of an operator and one or more operands. For a full list of AMDL operators, see AMDL Operators.

An example of a simple condition is the following, which is true if the value in the "country" field in the event is "GB":

```
event.country == "GB"
```

Conditions are also numerical comparisons, such as:

```
event.amount.baseValue > 100
```

In place of a condition, you can use a Boolean variable, or something that returns Boolean value. For example, using the example event from section Event Data, you can write this condition to check if a transaction was accepted:

```
event.accepted == true
```

Or you can write:

```
event.accepted
```

You can link conditions using the Boolean operators AND, OR and NOT. In AMDL, these are represented by && (AND), || (OR), and ! (NOT).

For example, this condition is true if the transaction event took place in Cambridge, UK (the country is "GB" and the city is "Cambridge"):

```
event.country == "GB" && event.city == "Cambridge"
```

The following condition is true if the event took place in Cambridge, or in the UK (or both):

```
event.country == "GB" || event.city == "Cambridge"
```

This condition is only true if the event neither took place in the UK, nor in Cambridge:

```
!( event.country == "GB" || event.city == "Cambridge" )
```

Note the use of parentheses to ensure the comparisons are executed in the correct order. The parameters ensure that the combination of conditions using OR is evaluated first, followed by the NOT. Parentheses are needed when you combine different Boolean operators, such as AND and OR, to ensure the logic is evaluated correctly. For example, this condition is true if the city is Cambridge and the country is GB or the US:

```
event.city == "Cambridge" &&
( event.country == "GB" || event.country == "US" )
```

However, this condition is true if the city is Cambridge, UK (city is Cambridge, country is GB). It is also true if the country is the US (if the country is the US the city has no effect):

```
(
event.city == "Cambridge" && event.country == "GB" ) ||
event.country == "US"
```

You write rules using conditions. The condition enables a rule to evaluate to true or false. If evaluated to true, a rule trigger actions such as alerts and tags (see Creating a Rule).

3.4 Using Annotations

There are a variety of annotations that you can use in AMDL to change when an expression is evaluated, or the effects of evaluating an expression Appendix 4: AMDL Annotations includes a complete list of all the annotations that you can use in AMDL. You can use annotations in the following ways:

Usage	Description
Change the way a state or global expression stores data	This enables you to create an expression so that it stores a collections of values rather than a single value (see Collections), stores a rolling average, only stores a value the first time it is updated or takes a default value if it is undefined (see @defaultValue). For example, the rolling average annotation (see @rollingAverage) changes a state or global expression from one that stores a value that is overwritten each time the expression is executed into one that stores an exponentially weighted moving average: `@rollingAverage(12h)`
Change what happens when a rule is evaluated	You can use annotations to trigger effects such as alerts and tags, to output a risk score, or to suppress alerts and tags (to prevent them from being generated). These effects are achieved using annotations to AMDL rules.
Change when an expression is evaluated	You can use the `@eventType` annotation in AMDL Business Rules expressions to limit the evaluation of an expression to a particular event type (or more than one event type).
Provide additional information to other users	Annotations also allow you to add commentary and descriptions to your AMDL Business Rules expressions.

3.5 Static Values

You can use the AMDL "values" scope to define static values, which are constants that do not change unless a user edits the expression. For example, you can define the following threshold value for use in other expressions:

```
values.threshold: 50
```

These values can then be referenced in other expressions using the standard reference syntax of values.<nameOfExpression>. Typically, you use the standard reference syntax to prevent repetition of a value across multiple AMDL definitions, which allows easier updates. For instance, you can have a list of countries that are shared by several rules, where storing these in a values definition allows easy modification.

3.6 Transient Variables

Transient variables are AMDL expressions which are re-calculated for each event, and not stored for future events. You declare these variables in a similar way to entity state, but with the scope "var". For example, you could calculate the current exchange rate used in each event by finding the ratio of the value in the original currency to the value in the base currency:

```
var.fxRate: event.amount.value / event.amount.baseValue
```

Other AMDL expressions, in the "'var" scope and other scopes, can reference the results of these calculations using the standard reference syntax of var.<nameOfVariable>. This appears in the following example:

```
var.thresholdInOriginalCurrency: values.threshold * var.fxRate
```

In Business Rules, transient variables can help tidy a set of expressions which all contain similar expressions with slight variations, or which contain references to the result of a common calculation. For example, many expressions might refer to the exchange rate defined above as var.fxRate.

3.7 Conditional Assignment

For many use cases, you need to be able to conditionally assign a value or update a state expression. For example, you might want to store the value of the most recent accepted transaction, the last unsuccessful transaction, or the merchant of the most recent transaction of over $1000 in value.

AMDL uses the conditional operator, ?, for conditional assignment (see Ternary operator for other uses). This operator acts like an "if/then" statement in other languages, providing a way to evaluate some expression only if a certain condition is true. The ? operator must be preceded by a Boolean expression, or a condition (see Conditions) or some other expression that evaluates to true or false. The operator is followed by the expression or value to evaluate if the condition is true.

The following examples is for creating a state expression that stores the value of the most recent card not present (CNP) transaction:

```
state.lastCNPTransactionValue:
event.transactionType == "CNP" ? event.amount.baseValue
```

Note that this uses the Business Rules syntax for state expressions (see Adding State Information to a Rule).

This state variable is only updated for events where the transactionType field is "CNP". For other events, the condition before the ? operator evaluates to false (or evaluation stops altogether if the transactionType field is missing from that event). When this happens, the expression stops being evaluated and any value currently in the state variable is therefore not overwritten.

When using the conditional operator, you can also supply an expression to evaluate if the condition is false. This enables you to implement an "if/then/else" logic, storing one value if the condition is true and other if it false. If the evaluation of the initial condition stops, the expression will not be updated at all.

For example, for a deposit and withdrawal event types, you might want to calculate a signed version of the event value, which is - for a withdrawal and + for a deposit. Signed values for an event make it easier to keep track of a customer's balance. You can enter the following code using a transient variable:

```
var.signedAmount: event.eventType == "deposit" ?
event.amount.baseValue :
-1 * event.amount.baseValue
```

If the event type is "deposit", this variable takes its value from event.amount.baseValue. Otherwise (if the initial condition is false), it takes the value of this field multiplied by -1.

3.7.1 Switch Case Operator

In a situation where a variable can take multiple values, you can create an expression that evaluates differently depending on the value. You could do this by chaining multiple conditional statements together (see Conditional Assignment). However, you can create more readable expressions using the AMDL switch case operator in this example:

```
@eventType
(deposit)
state.lastAmount:
event.country ~?
"GBR": event.currencyGBP;
"IRL": event.currencyEUR;
```

```
default: event.currencyUSD;
```

The ~? operator initiates the switch on the value of the country field. Each case is terminated by a semicolon. The values switched on must be either a fixed value or default.

The default value is optional, where including the default keyword means that if the rule engine gets to the end of a switch case execution without detecting a match, it uses the default case. If no default is included and the rules engine does not find a matching case, AMDL execution of the expression stops.

3.7.2 Switch Case Operator Usage

The ~? operator allows one of many operations to be evaluated based upon a switch expression. This is useful if you have several special cases for the same variable, each of which needs different treatment.

Example 1:
Here, you check the event amount against a different threshold based on the mcc field.

```
event.mcc ~? "7995": event.amount.baseValue > 150;
"5912": event.amount.baseValue > 200;
"4722": event.amount.baseValue > 5000;
default: event.amount.baseValue > 500;
```

Example 2:
Here, the 'default' case is optional. If the expression does not evaluate to one of the case labels, the expression evaluation stops.

```
event.mcc ~? "7995": event.amount.baseValue > 150;
"5912": event.amount.baseValue > 200;
"4722": event.amount.baseValue > 5000;
```

Note that case labels must be a fixed value or 'default'. Currently, switching based upon variable expressions is not supported. Also note that every switch case must be terminated by a semicolon.

3.8 Dates, Times and Durations

Date-times and durations in AMDL make it simple to compare times and base AMDL logic on the elapsed time between two events. Date-times in system events, and in AMDL, must be written in ISO-8601 format (http://www.w3.org/TR/NOTE-datetime), such as "2020-02-01T12:34:56Z", and must have a time zone designator. The designator can be:

- Z for UTC
- an offset in the format + (for time zones head of UTC)
- followed by hh for an integer number of hours (for time zones behind UTC)
- an offset in hours and minutes expressed as hhmm orhh:mm.

You can enter durations, which represent elapsed time, by appending a unit to an integer. For example, entering 7d represents 7 days.

The following table shows the duration units that you can use in AMDL.

Unit	Meaning	Example (all = 1 day)
d	Days	1d
h	Hours	24h
m	Minutes	1440m
s	Seconds	86400s

In AMDL, you can subtract one date-time from another to give a duration. You can also add durations to and subtract durations from a date-time to get another date-time. For example, this expression gives the date-time three hours after the date-time of the current event:

```
event.eventTime + 3h
```

This expression gives the date-time 30 minutes before the current event:

```
event.eventTime - 30m
```

The expression below shows the use of date-times and durations in a condition. The condition is true if less than 60 days have elapsed between the current event and the time the customer registered for their account. The first part of the expression evaluates to a duration representing the amount of time elapsed between the eventTime and the accountOpenDateTime; this is then compared to a fixed duration of 60 days (60d).

```
event.eventTime - event.accountOpenDateTime < 60d
```

Note the order of the two fields. If the account open date is known to be before the time of the current event (which seems likely), this order results in a positive duration. Inverting the order results in a negative duration, which is always less than 60 days (resulting in incorrect logic).

3.9 Collections

In AMDL, collections are data structures that contain multiple values. The simplest type of collection is an array. An array consists of multiple individual elements, which may be of any type. For example, this expression defines an array of strings:

```
values.dwarfs:
[ "Sleepy", "Dopey", "Happy", "Grumpy", "Sneezy", "Bashful", "Doc" ]
```

You can check whether a collection contains a given value using the "contains" operator, ~#. For example, given the collection above, this condition is true:

```
values.dwarfs ~# "Doc"
```

AMDL also has a "does not contain" operator, !# where, for example, the following condition is true:

```
values.dwarfs !# "Gandalf"
```

You can define arrays in-line as part of your AMDL expression. For example, the following is true if the value of the field merchantCategoryCodeis listed as follows:

```
[ "5122", "5912", "5993", "7841", "7995" ] ~# event.merchantCategoryCode
```

You can also use annotations to transform AMDL expressions, such as state and transient variable definitions, into collections. By default, these expressions store a single value, and when the expression is evaluated the stored value is updated and overwritten with the latest value. Collections stored in state allow you to store, for example, the values of a merchant's 10 most recent transactions, or the unique IDs of all the mobile devices used to access an account.

3.9.1 Arrays

Annotations let you change the way that data is stored in state and global expressions. The @array annotation let you turn a state or global expression from one that stores a single scalar value into one that stores multiple values. You must provide an argument to the @array annotation, which can be either a number or a duration. If the argument is a number it specifies the maximum number of values to store in the collection. If the argument is a duration, values are retained for that duration in the collection. Values that have been in the collection for longer than the specified duration are deleted when that expression is referenced or updated.

For example, this rules expression, defined for the 'consumer' entity type, stores the values of the last 10 transactions for each customer entity:

```
@array(10)
@eventType("transaction")
state.last10TransactionValues: event.amount.baseValue
```

This rules expression stores the values of all the transactions seen for each entity over the last 24 hours:

```
@array(24h)
@eventType("transaction")
state.transactionValuesLast24h: event.amount.baseValue
```

3.9.2 Sets

Whilst an array is an ordered collection of any values, a set is an unordered collection of unique values; for example, the collection of all unique IP addresses from which an account has been logged in to. Sets are declared in exactly the same manner as arrays, except using the @set annotation.

Sets have varied use cases. For example, you can use sets to:

- Store the different payment methods of a customer over the last 30 days.
- Store the last 10 mobile device IDs seen logging into a particular account.
- Store the IP addresses a device has logged in from in the course of an hour.

The following example shows the different payment methods used by a customer over the last month.

```
@set(30d)
state.methodIdsLastMonth: event.methodId
```

3.9.3 Using Collections

A single value can be checked against the elements of these collections using the ~# and !# operators, or other collection operators. This allows you to check whether a value is contained in an array or set, or check all the elements in a collection against a single value. For example, you can check if all the elements in the array are greater than 100, or if all the elements in the array are less than the value of the current transaction.

You can use the collection methods described in Appendix 5: AMDL Methods to extract data from a collection. For example, there are methods for calculating the number of elements in a collection, the total value, the mean, median and mode, and other statistical measures. Methods also exist for sorting collections alphabetically or numerically, or combining collections.

There are also more complex types of collection, including histograms (see Using Histograms) and maps (see Using Lookup Tables).

3.9.4 Collection membership operators usage

These operators are used to test membership of a collection, and return a Boolean value. In usage, the left-hand operand must always be some form of collection, and the right-hand operand the element of the collection whose membership is being tested.

Operator	Usage
~#	Contains
!#	Does not contain

Example 1:
Checking that a list in the event contains the value 20.00.

```
event.transactionAmounts ~# 20
```

Example 2:
Checking whether a string in the event is contained within a pre-defined collection of values.

```
values.declineStatusCodes ~# event.responseMessage
```

Example 3:
Checking that the country field in the event is not one of "GB", "US" or "IS".

```
{ "GB", "US", "IS" } !# event.country
```

3.9.5 Collection comparison operators usage

Operators: ==# !=# <# <=# >=# >=#

These operators perform comparison operations upon all elements of a collection and return the Boolean combination of the results. The left operand must always be a collection and the right operand that which is to be compared.

Example 1:
All elements of the ordered collection are equal to 1, so this is true.

```
[ 1, 1, 1, 1, 1 ] ==# 1
```

Example 2:
None of the elements in the collection are equal to "strawberry", so this is true.

```
{ "apple", "pear", "banana" } !=# "strawberry"
```

Example 3:

All values are less than or equal to some pre-defined threshold.

```
event.transactionAmounts <=# values.threshold
```

Example 4

`>` operator for comparing datetimes, ensuring that `>#` checks that all datetimes exceed the right operand.

```
state.transactionTimes ># event.accountOpenDateTime
```

3.10 Manipulating Strings

In AMDL, you can manipulate strings using:

- String concatenation
- Regular expressions

3.10.1 String Concatenation

You can concatenate strings using the `..` operator. For instance, to store a full name as state, you could write:

```
@eventType(registration)
state.fullName: event.firstname .. " " .. event.surname
```

String Concatenation Usage

Operator: `..`

This operator concatenates any two strings, numbers, durations, or times as strings. For non-strings this is done in such a format that they can be coerced back to their original types.

Example 1:

Evaluates to "Hello World".

```
"Hello " .. "World"
```

Example 2:

Produces a full name from component parts.

```
event.firstName .. " " .. event.lastName
```

3.10.2 Regular Expressions

Note: By default, regular expression matching and replacement are disabled in Solution and Thredd-level AMDL. This means that you cannot use the regular expression operators described below in Solution or Program Manager level Business Rules.

AMDL supports two regex operators: the regex match operator `~=`, and the regex substitution operator `~`:

The regex match operator `~=` takes a match regex as its right-hand argument. This is a string enclosed by forward slashes (for example, "/.*/"). The operator results in a Boolean value true if the regex matches a substring of the left-hand argument. Full-text matches can be implemented by adding `^` to the beginning and `$` to the end of the match regex, in order to match the beginning and end of the string. For example, you may wish to detect whether an email address field ends in "protonmail.com". This could result in a rule such as:

```
rules.protonmailDomain: event.email ~= "/protonmail\.com$/"
```

Use of the Backslash

You can use the backslash to escape the period. Internally, the system evaluates AMDL using Java. This means the allowed regex syntax is identical to Java's Pattern syntax, except that you escape forward slashes with a backslash. For example, to match `"/"`, you use the following regular expression: `"/\//"`.

Regex Substitution

The regex substitution operator takes a substitution regex as its right-hand argument. This is two strings, a match regex and a replacement string, enclosed and separated by forward slashes. For example, `"/\s/_/"` replaces space characters by an underscore. The result of the operator is a copy of the left-hand argument, except with each substring that matches the match regex replaced by the replacement string. For example, `"Hello world!" ~: "/l/LL/"` returns the string `"HeLLLLo worLLd!"`. You can refer to matching groups in the matching regex using `$` followed by the match group number (for example `$1` for the first match group). To replace each letter in a string with the same letter followed by an asterisk for example, you can use the regex `"/(.)/$1*/" - "Hello world!" ~: "/(.)/$1*/" returns "H*e*l*l*o* *w*o*r*l*d*!*"`.

For example, we can write a rule that removes any initial titles (e.g. "Mr", "Ms", "Mx" or "Prof.") from the cardholder name in the event, and compares the remaining name to the stored customer name:

```
rules.cardholderNameWithoutTitle_DoesNotMatchStoredName:
( event.cardholderName ~: "/^(([DdMm][RrXx]?[Ss]?|Prof)\.?\s+)+//" ) !=
state.customerName
```

Regular Expression Match Operator Usage

Operator: `~=`

This operator checks whether the left string operand matches the regular expression contained in the right operand. Note that you can only apply the operator to strings. For more details on regular expressions, see Outputting Values.

Example 1:
Checking whether the postcode starts with "CB".

```
event.address.postcode ~= "/^CB/"
```

Regular Expression Substitution Operator Usage

Operator: `~:`

This operator performs a substitution specified in the right operand on the left string operand. Note that you can only apply the operator to strings. For more details on regular expressions, see Outputting Values.

Example 1:
This expression substitutes dots with dashes, resulting in "1970-01-01". Note that it does not matter whether regexes are quoted or unquoted.

```
"1970.01.01" ~: "/-/./"
```

3.11 Specifying a Default Value

If a reference within an AMDL expression cannot be found (for example, an event field which is not present, or a state which hasn't been updated yet), it returns a null value. If the null value is not handled in the logic of the expression, the AMDL execution of the expression stops and the value of the expression being evaluated also becomes null.

For scenarios where the expression is a rule, the rule does not trigger. If the expression is a variable update expression such as a state, global or transient variable, that variable is not updated. Although this is the desired behaviour, you need to be careful you handle such possibilities when needed.Note that executing AMDL does not short-circuit Boolean operators and evaluates all references. The evaluation of expressions stops if any of these references evaluates to null.

For example, even if the field accepted has a value in the processed event and `var.acceptedTransaction` is undefined, evaluation of this rule ceases. As a result, the rule does not trigger:

```
rules.accepted: event.accepted == true || var.acceptedTransaction == true
```

The 'default value' operator `??` automatically replaces a preceding reference with the value following it, if that reference is found to be null during execution. For example, to ensure that the rule above executes even if the accepted field is not present:

```
rules.accepted:
( event.accepted ?? false ) == true || var.acceptedTransaction == true
```

This ensures that the evaluation of the field accepted always returns a value. If the field is missing, the default value false is returned. Note that the parentheses ensure that the components of the expression are evaluated in the correct order (see Operator Precedence for more information).

3.11.1 Usage

Operator: `??`

In AMDL, expression evaluation stops if a referenced variable does not exist. The `??` operator lets you specify a default value if an expression cannot be evaluated.

Example 1:
If the event does not contain a field called `amount.baseValue`, this evaluates to 0 instead. This expression is particularly useful when dealing with optional fields that may or may not be present in the event.

```
( event.amount.baseValue ?? 0 ) > 100
```

Example 2:
You can default to the value of another field, or a state, or other variable as well as a fixed value.

```
( event.chargebackTime ?? event.eventTime ) < state.prevTime + 90d
```

3.12 Evaluating if an Expression Returns a Non-Null Value

You can use the 'exists' operator `~` to query whether the result of an expression exists or not. This allows you to create conditions which are true if an optional event field exists within the specific event being processed, or if a specific state variable, transient variable or values expression has a value. For instance, you can check whether a field exists in the event by using the following syntax:

```
rules.fieldExists: ~event.field
```

You can construct an inverse using the "does not exist" operator. The inverse is a condition which is true if a field does not exist. This is a combination of the 'exists' operator and the 'NOT' Boolean operator.

3.12.1 Usage

Operator: `~`

This operator checks whether evaluation of a reference returns a non-null value.

Example 1:
Evaluates to true if `optionalField` is present in the event being processed.

```
~event.optionalField
```

Example 2:
Evaluates to true if `previousTransactionValue` is defined in state for the entity being processed.

```
~state.previousTransactionValue
```

Note: Due to the wide-varieties of regions in which Threddcustomers operate in, there may be unreliability in some of the above data.

3.13 Storing the First Value for an Expression

You can capture a state once without it being updated again, for instance, the first login date of a user. You can use the 'exists' operator and the ternary operator. However, because it is a relatively common occurrence a special annotation is provided: `@firstValue`.

When applied to an AMDL expression, once the expression evaluates to a value, it maintains that value and is not updated any more.

```
@eventType(login)
@firstValue
state.firstLoginTime: event.eventTime
```

3.14 Using Histograms

Arrays and sets store all values they are updated with for a duration. When you expect to use a large number of values for updating the collection in the period of the duration, this could produce excessive storage requirements and adversely affect the engine performance. For instance, globally storing the value of all transactions over a month could result in a very large collection.

3.14.1 Histogram Buckets for Time-Series Analysis

To facilitate analysis of time-series data, AMDL provides a histogram type. A histogram consists of a group of buckets, each of which represents a group of updated values that occurred during a specific time window. A histogram is granular on the level of the buckets, not the values. Therefore, a histogram can be a powerful approximation to expedite performance.

You can provide any state-like AMDL expression in a histogram annotation which has the form `@histogram(historyLength=<duration>, bucketSize=<bucket duration>)`. The duration indicates the period for which the histogram should store buckets before expiring them. The second argument determines the number of buckets in the histogram and is optional. If left unspecified, a default is chosen to ensure at least 10 buckets are used. More buckets mean more accurate summary figures but a greater storage/performance impact. Bucket size is limited to a series of acceptable durations. If not one of the acceptable bucket sizes (see below), the nearest acceptable bucket size is used. This is ensures the histogram uses the expected number of buckets.

Histogram Bucket Data

Acceptable bucket sizes for histograms include:

- **Monthly buckets**: 12 months, 6 months, 4 months, 3 months, 2 months, 1 month
- **Fixed-length buckets**: 7 days, 1 day, 12 hours, 6 hours, 3 hours, 2 hours, 1 hour, 30 mins, 20 mins, 15 mins, 10 mins, 5 mins, 1 min, 30 seconds, 5 seconds, 1 second.

Bucket durations are expressed in months (e.g. 6 months, 12 months). The durations use ‘M’ as their unit (for example, 6 months is represented as ‘6M’).

Histogram buckets align with calendar time periods. For example, monthly buckets align with calendar months, so the first relevant event that occurs after midnight (UTC) on the first day of the month contributes the first datapoint to that month's bucket. Buckets with a shorter duration (expressed in days, hours, minutes or seconds) align with calendar days. All non-monthly histogram buckets align at midnight (UTC) each Monday, so buckets with a duration of 7 days start at midnight on Monday; buckets with a duration of 1 day start at midnight each day; buckets with a duration of 6 hours start at midnight, 6am, noon, and 6pm each day.

Like arrays, you can call the size, total and mean methods on a histogram state.

3.14.2 Expressions and Rules

You can define a global histogram that stores transactions over the last week as shown in the following example:

```
@eventType
(transaction)
@histogram (historyLength=7d, bucketsize=1d)
globals.txAmounts: event.amount
```

You can then write a rule which triggers if an entity performs a transaction of an amount greater than 1000, but normalised for the day's global average transaction amount in relation to the week's.

```
@eventType
(transaction)
rules.highTx:
event.amount *
( globals.txAmounts.mean(7d) / globals.txAmounts.mean(1d) ) > 1000
```

Note that you add a duration argument to the mean method. This limits the values of the histogram to those within that passed period, so far as the granularity of the bins allow. In the example above, the mean(1d) method returns the mean of values from today. The `mean(7d)` method returns the mean of values from today, and the previous 6 days.

Monthly durations (such as 1 month, 6 months) use calendar months, which can be of different durations. To avoid issues when comparing histogram data between months of different lengths, normalised versions of these methods are also provided (`normalisedSize`, `normalisedTotal`, `normalisedMean`). These methods normalise the total, size or mean to a month length of 30 days.

Used in this way, these methods can access data from the most recent bucket, or from that bucket and previous buckets. However, for comparison purposes or for defining a baseline, you can refer to previous buckets. You can provide a date-time as an argument to the size(), mean() or total() method which allows you to access the bucket that contains that date-time. This date-time can be derived from event data through subtraction of a duration, to return data from the bucket from a specified duration in the past.

For example, the expressions below enables you to compare the total value of a customer's transactions this week with the total value last week.

```
@eventType
(transaction)@histogram(bucketSize=7d, historyLength=28d)
state.TransactionValueWeekly: event.amount
```

```
@alert
@eventType
(transaction)
rules.txnValueThisWeekGtr3timesLastWeek:
event.amount + state.TransactionValueWeekly.total(7d) >
3 * state.TransactionValueWeekly.total( event.eventTime - 7d )
```

This example rule triggers an alert if the total value of a customer's transactions so far this week is greater than three times the total value of the customer's transactions last week. Data is returned for last week's bucket instead of for the latest bucket by passing the date-time exactly 7 days ago. This is in the format of event.eventTime - 7d as the argument to the total() method.

However, this approach only allows you to access data from a single bucket. To access data from multiple previous buckets, you can use the atTime() method. This method only applies to histograms, and takes a date-time as an argument. This date-time can be derived by subtracting a duration from a date-time.

Example of an Alert

You can modify the rule in the previous example to generate an alert. In this example, the total value of a customer's transactions this week is greater than the total value of a customer's transactions over the previous 2 weeks.

```
@alert@eventType(transaction)
rules.txnValueThisWeekGtrTotal2PreviousWeeks:
event.amount + state.TransactionValueWeekly.total(7d) >
state.TransactionValueWeekly.atTime( event.eventTime - 7d).total(14d)
```

To access data in this example, you use the atTime() method with an argument 7 days in the past in combination with total(14d). The total value is in two buckets that does not include the current one. It includes the bucket with the date-time supplied as an argument (7 days ago), and the previous bucket. This is because the bucket size is 7 days and the duration supplied as an argument to the total() method is twice the duration.

Time Field

By default, histograms use the event data field eventTime to determine the time at which the current event occurs. This therefore determines which bucket to update and/or which bucket(s) to access.

The @histogram annotation can take an optional argument, timeField, which allows you to define a non-default time field to use. This can be an event data field or a transient variable, specified as a string. For example, for an event data field called "realTime", you write timeField="event.realTime"). The eventTime might not signify the time that the original transaction took place, particularly with batch processed event, so you might want to use another field as the time field:

```
@histogram(bucketSize = 1d, historyLength = 7d, timeField =
"event.realTime")
state.transactionValues: event.amount.baseValue
```

Alternatively, you can use a derived time calculated in a transient variable. For example, if the transient variable "newTime" contains the date-time in the histogram time field:

```
@histogram(bucketSize = 1d, historyLength = 7d, timeField =
"var.newTime)
state.transactionValues: event.amount.baseValue
```

3.14.3 Testing Histograms

To test an expression that refers to a histogram, you must specify a simulated version of the referred histogram. For example, the following rule displays with the initial state form a unit test.

```
@alert
@eventType(transaction)
```

```
rules.highTotalTransactionValueToday:
event.amount + state.txnValueDaily.total(1d) > 5000
```

```
@histogram
(bucketSize = 1d, historyLength = 3d)
state.txnValueDaily: {
"data": {
"2019-03-31": { "size":17, "total":5325 },
"2019-04-01": { "size":10, "total":3575 },
"2019-04-02": { "size":14, "total":4500 },
}
}
```

The state definition in the 'Initial State' test panel must begin with a copy of the histogram annotation from the state being referred to. The histogram is defined as a JSON map with a single key, "data", within which is a series of JSON objects representing the buckets in the histogram. Each bucket has a key, which is the start date(-time) of that bucket, and the value is a JSON object containing two properties, "size" and "total".

3.15 Using Lookup Tables

A map is a lookup table where there is a map between two values that return a resultant value. This enables the storage of a key-value state. For instance, in a transactional system every entity may have several different payment methods, and you may want to keep track of the number of transactions made using each method. You can use each method identifier as the key, and the count of the number of transactions as the value (i.e. what each key is mapped to).

The data in this map can be represented as follows:

```
transactionCountByPaymentMethod: {
"method1": 10,
"method2": 3
}
```

For `"method1"` as the key, `transactionCountByPaymentMethod` returns the value 10.

You can use AMDL to define both static maps (using the "values" scope) and dynamic maps for updating and modification with each incoming event, stored in an entity or a global state.

3.15.1 Lookup Tables in Static Maps

Static maps store lookup tables such as a currency conversion table, or a dictionary of country codes mapped to country names. For example, you can store a list of threshold values for different merchant categories, and raise an alert when the value of a transaction exceeds the threshold for the appropriate Merchant Category Code (MCC). You can implement static maps using nested conditional statements or for better readability, a switch case statement (see Switch Case Operator). However, using a static map can simplify this kind of expression by removing the thresholds into a different expression from the business logic of the rule itself. You can define a map like this:

```
values.MCCSpecificThresholds: {
"7999": 300,
"7995": 1000,
"5912": 200,
"5411": 450,
"5311": 750
}
```

A rule can then reference this map using the syntax `<scope>.<map expression>[<key>]` for example, `values.MCCSpecificThresholds["7999"]` returns 300.

For example, this rule generates an alert if the value of a transaction exceeds the threshold for the relevant MCC:

```
@alert
@eventType("transaction")
rules.valueOverMCCThreshold:
event.amount.baseValue >
values.MCCSpecificThresholds[ event.merchantCategoryCode ]
```

Using Switch Case Statements

You can write the above expression for lookup tables using a switch case statement (see Switch Case Operator), which allows you to provide a default value. A map does not provide a way of specifying a default, but you can use the default value operator (see Specifying a Default Value). The default value is returned if the referenced key is not present in the map.

This example specifies a default threshold of 500:

```
@alert
@eventType("transaction")
rules.valueOverMCCThreshold:
event.amount.baseValue >
(values.MCCSpecificThresholds[ event.merchantCategoryCode] ?? 500 )
```

3.15.2 Storing Maps

You can store maps dynamically through state or global expressions. For example, you can store the timestamp of the last time a customer has used the transaction method of the current transaction. The transaction could be from a specific payment card or tokenised payments such as Apple Pay or Google Pay. If each payment method has a unique ID in the event field event.paymentMethod.methodId, you could define a state expression that stores the timestamp of the last transaction for each unique method ID:

```
state.lastTimeMethodSeen[ event.paymentMethod.methodId ]: event.eventTime
```

Each time you receive a transaction with a particular method ID, the timestamp is stored as the value, with that method ID as the key. After receiving the following transactions:

Method ID	Timestamp
method1	1st Dec 2019 10:01:24
method2	5th Dec 2019 08:17:54
method3	10th Dec 2019 17:26:12

The map stored in state looks as follows:

```
state.lastTimeMethodSeen: {
"method1": "2019-12-01T10:01:24Z",
"method2": "2019-12-05T08:17:54Z",
"method3": "2019-12-10T17:26:12Z"
}
```

If you received another transaction with the method ID "method2", at 15:26:41 on 11th Dec 2019, it overwrites the existing value for that key, and the resulting map looks as follows:

```
state.lastTimeMethodSeen: {
"method1": "2019-12-01T10:01:24Z",
"method2": "2019-12-11T15:26:41Z",
"method3": "2019-12-10T17:26:12Z"
}
```

You can then write a rule or other expression referencing this state. For example, you can reference the time a particular payment method was last seen, or check whether a payment method has been seen within a certain period of time. The following transient variable stores true if the payment method in the current event has not been seen in the last 180 days:

```
var.paymentMethodNotSeenInLast180days:
event.eventTime - state.lastTimeMethodSeen[event.paymentMethod.methodId] >
180d
```

You can also write an expression that updates values for multiple map keys for a single event, using syntax like the following:

```
<scope>.<expression name>[<key 1>]: <value 1>;
[<key 2>]: <value 2>;
...
[<key N>]: <value N>
```

The key values can be static or refer to event data fields or other AMDL expressions. For example, consider a transaction event that represents an e-commerce order. The event contains a shipping address and a billing address, uniquely identified by the shippingAddress.addressId and the billingAddress.addressId respectively.

You can store the last used billing and shipping address for each customer as follows:

```
state.lastAddressUsed[ "shipping" ]:
event.shippingAddress.addressId;
[ "billing" ]: event.billingAddress.addressId
```

You can then refer to the most recently used shipping address using state.lastAddressUsed["shipping"], or the most recently used billing address using state.lastAddressUsed["billing"].

For an example of using event data fields as keys, you can use the same transaction event that contains a billing and a shipping address. You could write an expression to store the date and time that each address was last used in an event as follows:

```
state.addressSeen[ event.shippingAddress.addressId ]: event.eventTime;
[ event.billingAddress.addressId ]: event.eventTime
```

By using the @array or @set annotation when defining a dynamic map in a state or global expression, you can create a nested dynamic map for storing a separate collection of elements against each key in the map:

```
@eventType("transaction")
@array(7d)
state.merchantTransactions7d[event.merchantId]: event.amount.baseValue
```

You can then create a rule that looks up a key in the map and uses collection methods to extract data from the corresponding collection:

```
@eventType("transaction")
rule.excessivePaymentsSingleMerchant:
state.merchantTransactions7d[event.merchantId].size(1d) > 6 &&
state.merchantTransactions7d[event.merchantId].total(1d) +
event.amount.baseValue > 40000
```

For more information about using arrays and sets, see Collections.

Warning: To avoid performance issues, the default limit for the number of keys stored in each map is 1000. Once a map reaches this limit, each time a new key is added, the oldest key is removed. However even with this limit, performance issues may occur, particularly in the case of nested maps that store a collection of elements with each key. When creating a dynamic map, you should consider the number of unique keys for storage. Note that in a nested map, keys are not removed if the corresponding collection becomes empty. When necessary, use the @mapOptions annotation to set an appropriate limit on the number of stored keys. See below for more information.

3.15.3 Specifying a Key Limit

When defining a dynamic map using a state or global expression, you can use the @mapOptions annotation and the keyDuration or keySize arguments to specify a limit on the number of keys stored in the map.

```
@eventType("transaction")
@mapOptions(keyDuration=14d, keySize=500)
state.merchantTransactions7d[event.merchantId]: event.amount.baseValue
```

You can use the keyDuration argument to set a duration limit. Using this argument removes any keys that were last updated before this duration. You can use the keySize argument to set a size limit. Once the number of stored keys reaches the size limit, each time a new key is added, the key that was last updated longest ago is removed. If you use both arguments, keys are removed according to whichever limit is reached first.

3.16 Using Data Lists in AMDL

In AMDL, you can access data in a data list defined at the same level of the expression using the lists scopes, using the 'contains' (~#) and 'does not contain' (!#) operators for collections. You can also update a data list. To create a data list, you need to use the Settings menu in the Fraud Transaction Monitoring System. For more information, see Create a Data List in the *Thredd Fraud Transaction Monitoring Portal Guide)*. Data lists are displayed as a table, consisting of a mandatory unique ID column (_id) and one or more other optional columns.

3.16.1 Accessing a Data List

In AMDL, you can access data in a data list defined at the same level of the expression using the lists scopes, using the 'contains' (~#) and 'does not contain' (!#) operators for collections. For example, you could write a rule at the Solution level that checks whether a particular merchant ID is in the unique ID column of a data list with the name highRiskMerchantList defined for the same Solution:

```
@alert
@eventType(transaction)
rules.merchantOnHighRiskList: lists.highRiskMerchantList ~#
event.merchantId
```

Note that this rule refers to a data list that has already been created in the Data Lists section of the Thredd. Fraud Transaction Monitoring portal. As shown in the following error message, you cannot save a rule that contains a reference to a data list that has not yet been created.

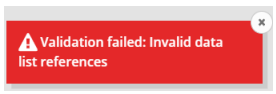


Figure 1: Validation error message

If your data list contains multiple columns, you can access the value in another column using AMDL map syntax. Using the unique ID as the key, you can select a row from the data list, and access columns as nested sub-maps using the column heading as the key.

For example, if a negative list of customers (called 'dataList') contains the column "mobileDeviceId":

```
rules.dataListCheck:
lists.dataList[event.consumerId]["mobileDeviceId"] == event.deviceId
```

3.16.2 Data List Size Limits

The following are the size limits on data lists:

Limit Description	Size	Effect
Number of rows in a single data list.	**Recommended Limit:** 60,000 **Hard Limit:** 600,000	If the number of rows exceeds the recommended limit, the processing of analytics that reference the data list is slowed down. In this situation, a warning is written to the Engine log. The warning threshold is configurable. When reaching the hard limit, the data list does allow any additional rows. A message is written to the engine log. This limit is configurable.
Number of rows across all data lists	**Recommended Limit:** 500,000 **Hard Limit:** 2 million	When reaching the hard limit, the data list does not allow any additional rows. An error is written to the data log. This limit is configurable.

Updating Data Lists Using AMDL

You can update data lists in the same way as for state and global variables, using a similar syntax and the lists scope. To add a unique ID to a data list, the syntax is the same as adding a value to a collection in a state or global expression, but using the lists scope. In this example, you can add a merchant ID to the high-risk merchant list if that merchant is included in a confirmed fraud report:

```
@eventType("fraud")
lists.highRiskMerchants: event.merchantId
```

Note that if this data list does not already exist, it will be created when this expression is evaluated for the first time.

If your data list has multiple columns, you can update the values in those columns as well, using the AMDL multiple values map update syntax. For example, to update a data list using the event data fields deviceId and ipAddress (adding these as columns headed "device" and "ip"):

```
lists.customerDevices[event.customerId]["device"]: event.deviceId;
["ip"]: event.ipAddress
```

As with updating a map, this update expression creates a new row in the data list, with the consumer ID from the event data as the value of the '_id' column, or overwrites such a row if it already exists. Note that the data list does not update if any of the update expressions fail to evaluate.

Testing Data Lists

The AMDL unit test functionality lets you test rules that refer to data lists. However, as unit tests do not have access to real data list data, you must create a simulated data list. This is in the same way where you create a simulated state when testing a rule that refers to a state or global expression.

To test a rule that refers to a single-column data list (one that only contains unique IDs), you can simulate the data list in the same way that an array (unordered collection) of strings is defined. For example, to test high-risk merchant ID rule (above), you can simulate a data list of merchant IDs by writing the following in the 'Initial State' test panel:

```
lists.highRiskMerchants: [
"M1056101",
"M3651540",
"M1120129",
"M9912832",
]
```

To test a rule that refers to a multi-column data list, you can simulate the list as an array of strings, where each string is a row of the data list. You simulate the rows using the following format: "_ id|column1Name:column1value|column2Name:column2Value...", where the string begins with the unique identifier for that row, followed by pairs of column name: value, separated by the pipe character (|).

For example, to test the multi-column data list (above), you can write the following in the 'Initial State' text panel:

```
lists.dataList: [
"1056101|mobileDeviceId:A01|ip:12.5.7.89",
"3651540|mobileDeviceId:D02|ip:11.5.7.89",
"1120129|mobileDeviceId:F11|ip:10.5.7.89",
"9912832|mobileDeviceId:Z76|ip:99.5.7.89",
]
```

3.17 Filtering Collections

You can filter collections using a predicate expression which evaluates to true or false. This returns a collection of all elements which meet the predicate, where it acts as a filter. The syntax for filtering collections is <scope>.<expression name/reference>[<predicate>]. The predicate expression uses $ to signify 'for each element in the collection'. For instance, consider a collection that contains the values of each transaction a particular customer has made within the last 24 hours:

```
state.transactionValues: [ 101, 99.99, 125, 45.99, 37.50, 48.96, 20, 10 ]
```

Using a predicate filter, you can write a rule to check how many transactions the customer has made with a value in excess of 100, containing the following condition:

```
state.transactionValues[ $ > 100 ].size() > 0
```

This condition is true if there are any elements in state.transactionValues with a value greater than 100 return a collection containing 2 elements, [101, 125]. The size of this collection is 2, therefore the condition is true.

The highlighted part of the expression is the predicate. The $ character means 'for each element' in the collection, and the predicate condition evaluates for each value in the array.

You can access properties of the values within the predicate by an extension of the $ syntax. For example, [$.currency == "GBP"] gets all objects in the collection that have a currency field equal to "GBP", Or you can enter the expression directly as [currency == "GBP"]).

The predicate filter is useful if your event contains an array of objects with properties, such as this array of items purchased:

```
{
"eventType": "transaction",
"items": [
{
"sku": "1234567",
"description": "ARIC™ Man™ action figure",
"unitCost": 22.99,
"quantity": 1,
"totalCost": 22.99
},
{
"sku": "9876543",
"description": "ARIC™ Man™ Fraud-fighter™ costume",
"unitCost": 8.99,
"quantity": 2,
"totalCost": 17.98
```

```
},
...
],
...
}
```

The following condition checks whether there are any items in the items array with a "sku" of "1234567", and evaluates to true.

```
event.items[ sku == "1234567" ].size() > 0
```

Note: The default limit for a collection is 1,000 elements.

3.18 Iterating over Elements in a Collection

AMDL expressions can iterate over all elements of a collection using the * operator. Referring to a collection using the [*] syntax performs a selector operation. For example, using the example event described in Filtering Collections above:

```
event.items[*].totalCost
```

This example returns a collection of all the values of totalCost for items within that array – in the case of the example event, [22.99, 17.98, ...]. If you have several nested layers, the selector is applied to everything after the [*]. For example, event.history [*].items.consumer.id returns a collection of all the items.consumer.id values nested within any elements of the collection event.history. A collection can also be nested if your event structure contains nested arrays. The expression below expands into all the values of all the field3 fields at the specified path:

```
event.data.array1[*].field1.field2.array2[*].field3
```

If a state update expression (key or value) contains the iteration operator, state variable updates with each value in the collection iteratively. The following expression results in the total costs for each item in the collection being added to the set individually:

```
@set(10d)
state.itemCosts: event.items[*].totalCost
```

Iterating on a map key results in each key being updated with the specified value:

```
state.itemTimes[ event.items[*].sku ]: event.eventTime
```

If both the key and the value contain the iterator, both collections iterate in sequence. The first key updates with the first value, and the second key with the second value, which continues.

```
state.itemCostsBySKU[ event.items[*].sku ]: event.items[*].totalCost
```

If the two collections do not have the same number of elements, the expression does not evaluate.

4 Customising Business Rules in AMDL

Business Rules AMDL lets you implement fraud and financial risk detection logic by breaking down the logic into units of AMDL expressions. The Getting Started section shows you how to create a basic rule and add state information. This section shows you how you can customise rules. The examples in this section use `transaction` and other event types (see Events for more details).

4.1 Suppressing Alerts and Tags

In some cases, it may be desirable to suppress the generation of alerts or tags for specific events – for example, to prevent customers on a VIP list from having their transactions held or blocked, for example a CEO or person of high net worth. You can add the `@suppressAlert` to a rule in order to prevent any alerts from being generated by AMDL rules or model risk thresholds. When a rule with this annotation triggers, all alerts are suppressed for that event/entity combination. The following example shows you how to prevent alerts being generated on transactions made by VIP customers:

```
@suppressAlert
@eventType("transaction")
rules.noAlertsForVIPs: event.customerSegment == "V"
```

Automated responses to rules (e.g. holding or declining a transaction) may also be based on tags added to an event. You can use the `@suppressTag` annotation to prevent specific tags from being added to the system response. You supply the tag, or lists of tags, for suppression as an argument to the annotation.

For example, this rule prevents VIP customers from having their transactions declined or routed via 3DS for declining deposits when the adding the `action="DENY"` tag to the output, and using the `via3DS="Y"` tag to route via 3D Secure authentication.

```
@suppressTag(action="DENY")
@suppressTag(via3DS="Y")
@eventType("transaction")
rules.noInconveniencesForVIPs: event.customerSegment == "V"
```

As with `@suppressAlert`, `@suppressTag` suppresses tag emission from all sources, including AMDL rules, models, risk score thresholds and aggregators. Suppressed tags are never shown in the UI, and are removed from the `outputTags` key in the JSON response. However, the models section of the response lists any tags assigned by models under tags.

Note that, because you define Business Rules for a specific entity type, alert and tag suppression only work within an entity type. If you write an expression with the `@alert` annotation for one entity type, and an expression with the `@suppressAlert` annotation against a different entity type, the second expression does not suppress the alert generated by the first. This is true even if both alerts trigger on the same event.

4.2 Adding Rule Scores

You can configure Rules to output a score using the `@score` annotation. Each rule supplies the score as an argument, where scores may be positive or negative. The total of all the scores from triggered rules for a specific event is output as the 'businessrules' model score. This score is rescaled in the same way that model scores are – a total score of 1 is equal to 100%. For example, for an event with an MCC of 5678 and an amount of £200, the following rules output a total score of 0.3 (30%):

```
@score(0.4)
rules.highTransactionValue: event.amount.baseValue > 150
```

```
@score(0.25)
rules.highRiskMCC:
[ "7999", "7995", "6051", "5912", "5933" ] ~#event.merchantCategoryCode
```

```
@score(-0.1)
rules.currencyIsGBP: event.amount.currency == "GBP"
```

If the 'businessrules' model score exceeds the overall risk score generated by any other model(s) for an event, it displays as the overall risk score in the Fraud Transaction Monitoring Portal for that event. These may result in risk score bars in excess of 100% or that are negative. While displayed appropriately in Fraud Transaction Monitoring System it may or may not be appropriate to assign events a risk above 100%.

var and rules Scope

The `@score` annotation exists in both the `var` and the `rules` scope. A scope allows you to write expressions that refer to another value (or operand), or provide definitions.

When added to the `var` expression, the score contributes the evaluated result of the expression to the risk score output of the 'businessrules' model. The following example scales the model down by 70% (represented as `models.externalModel1.score`) and adds the value to the 'businessrules' model score.

```
@score
var.ruleModelScore: models.externalModel1.score * 0.7
```

The score can only contribute to the evaluated result to the model score if it is:

- a single numeric value
- either positive or negative
- greater than 1

Thus, decimals are not allowed for the score value.

Annotating multiple var expressions with `@score` adds up the score. The score also includes any scores from `@score(<score>)` annotated rule expressions. The result is in an output of the total 'businesrules' model score.

When added to a rules expression, the annotation provides a static value to the expression that is returned if a specific rule is triggered. The value is added to any other rules triggered by the score annotation, and any other `var @score` value, resulting in an output of the total 'businesrules' model score.

The following example adds 0.4 as the static value to the 'businesrules' model score if the `event.amount.baseValue` is more than 150. The name of the business rule here is `highTransactionValue`.

```
@score(0.4)
rules.highTransactionValue: event.amount.baseValue > 150
```

Defining score generating expressions in the `var` scope is recommended as it enables both dynamic generation of the model at runtime and more complex logic in a single expression.

4.3 Limiting Events through Annotations

You can limit the events for evaluating a rule using the eventType annotation. For example, you can specify a rule that is only evaluated on the "transaction" event type:

```
@eventType("transaction")
rules.transactionGreaterThan100: event.amount.baseValue > 100
```

You can define a rule that is evaluated for multiple event types by adding additional eventType annotations. For example:

```
@eventType("transaction")
@eventType("accountTransfer")
rules.transactionOrAccountTransferGreaterThan100:
event.amount.baseValue > 100
```

If you do not specify event type annotations on a rule, the rule is evaluated on every event.

However, this rule does not produce any effect when triggered. In AMDL, you use annotations to cause a rule to generate an alert or add a tag to the output when it triggers.

You use the @alert annotation to create an alert in the Fraud Transaction Monitoring Portal if the rule triggers. An analyst can then review the incident and add the alert to the rule.

```
@alert
@eventType("transaction")
@eventType("accountTransfer")
rules.transactionOrAccountTransferGreaterThan100:
event.amount.baseValue > 100
```

It is important which entity type this rule is written against, because the alert is raised against the entity of that type in the evaluated event.

For example, if you write this rule against the 'customer' entity type, the following event generates an incident for the entity "Customer1":

```
{
"eventType": "transaction",
"eventTime": "2019-05-05T18:02:55Z",
"customerId": "Customer1",
"merchantId": "Merchant2",
"amount": {
"baseValue": 150,
...
}
```

However, if you write this rule against the 'merchant' entity type, the incident relates to "Merchant2".

You can also use the @tag annotation to add a tag to the output from the system when the rule triggers. For instance, you could add tags to the rule as follows:

```
@alert
@tag("High value transaction or account transfer")
@tag(action="BLOCK")
@eventType("transaction")
@eventType("accountTransfer")
rules.transactionOrAccountTransferGreaterThan100:
event.amount.baseValue > 100
```

This rule produces a tag with the namespace of action and value "BLOCK", and another which displays as the text "High value transaction or account transfer" in the Fraud Transaction Monitoring Portal.

The general form of the @tag annotation is @tag (<namespace>="<value>","<namespace>="<value>"). You can add more than tag by adding the @tag annotation multiple times. Or, you can add multiple tags using the same annotation:

```
@tag(<namespace1>="<value1>","<namespace2>="<value2>"…)
```

To add multiple tags in the same namespace, use this syntax:

```
@tag(<namespace1>="<value1>","<namespace1>="<value2>"…)
```

If you do not provide a namespace, you can use a default namespace (_tag). If the namespace is unimportant, you can write @tag ("<value>"). The portal displays this tag without a namespace.

4.4 Using State in a Rule

You can detect test transactions, which include a low value transaction to test the payment method used followed by a large transaction. To detect a low-value transaction followed by a high-value one, you need to be able to store some information about the customer's previous transaction.

To store the value of the most recent transaction each customer has made, you can write a state expression:

```
@eventType("transaction")
state.previousTransactionValue: event.amount.baseValue
```

Similar to rules, the `@eventType` annotation limits the execution of this expression to transaction events.

The entity type against which you write this expression is important, as the state is stored for the entity of this type. For the following event, if the above expression is written against the 'customer' entity type, this variable stores the value 100 in the state of the entity "Customer1":

```
{
"eventType": "transaction",
"eventTime": "2019-05-05T18:02:55Z",
"customerId": "Customer1",
"merchantId": "Merchant2",
"amount": {
"value": 100,
"currency": "GBP",
"baseValue": 100,
"baseCurrency": 100
},
...
}
```

You can then reference this state expression in a rule, provided that rule is written against the same entity type. This state expression is evaluated for each transaction event, and the transaction value stored against the relevant entity. However, this evaluation is done after the evaluation of rules and other expressions. Thus, when evaluating a rule, the value of the state expressions it references are the values as of the most recent event for that entity.

For example, this rule generates an alert if a customer makes a low-value transaction of less than 10 and their next transaction event is a high-value transaction of over 100:

```
@eventType("transaction")
rules.testTransaction: event.amount.baseValue > 100 &&
state.previousTransactionValue < 10
```

You can also write state expressions based on one event type, which are referenced in rules written against another event type. For example, you can store information about a customer when they register, and use that information in a rule.

For example, suppose a registration event contains the following data:

```
{
"eventId": "14b1f1e1a124c",
"eventType": "registration",
"eventTime": "2019-04-01T12:10:30Z",
"customerId": "3263827",
"customerSegment": "B",
"name": {
"firstName": "Exem"
"lastName": "Plar"
},
"email": "mrexample1@gmail.com",
"deviceData": {
"deviceId": "a85531c1-02d8-44ed-964f-0706155209c7",
"deviceType": "Android",
"OSversion": "10.0.1"
}
}
```

This data can be stored against the 'customer' entity type. For example, storing the customer's customer segment code determines whether they are a VIP customer when rules are executed:

```
@eventType("registration")
state.customerSegment: event.customerSegment
```

This rules ensures that a test transaction rule does not apply to VIP customers:

```
@eventType("transaction")
rules.testTransaction: event.amount.baseValue > 100 &&
state.previousTransactionValue < 10 &&
state.customerSegment != "V"
```

4.5 Applying Timestamps and Durations and Conditional State

You can apply timestamps to a rule and durations. In addition, you can apply conditional states.

Timestamps and Elapsed Time in Business Rules

In the previous example, the created rule triggers if a customer made a low-value transaction followed by a high-value one. However, that rule does not reference the time elapsed between those two transactions. With a typical test transaction, a high-value transaction is usually made shortly afterwards. A low value transaction is unlikely to be a test transaction if the following high value one takes place a week later. Times and durations (see Dates, Times and Durations) are therefore important in Business Rules.

To store the timestamp of each transaction for a customer, you can use a state expression:

```
@eventType("transaction")
state.previousTransactionTime: event.eventTime
```

You can then refer to the state expression in a rule, to determine the elapsed time since the previous transaction. This means you can modify the rule from example 2 to take into account the elapsed time since the low-value transaction. The following example is a rule triggered less than 2 hours since the previous transaction:

```
@eventType("transaction")
rules.testTransaction:
event.amount.baseValue > 100 &&
state.previousTransactionValue < 10 &&
event.eventTime - state.previousTransactionTime < 2h
```

state.previousTransactionTime and event.eventTime are both fields that contain date-times, and subtracting one from the other gives a duration representing the elapsed time between the two events.

Conditional State

In the example of Timestamps and Elapsed Time in Business Rules, you create a rule that checks the value and time of the last transaction where it triggers if a low-value transaction is followed by a high-value one within 2 hours. However, this rule could fail to detect a test transaction if a customer's card has been compromised and the card details are being used by both the criminal and the genuine cardholder.

Consider the following sequence of events:

Time	Cardholder or criminal?	Transaction value change	Time since previous transaction	Result
10:00 am	Criminal	Transaction Value: £ 5 Previous Transaction Value: N/A	N/A	No alert
10:30 am	Cardholder	Transaction Value: £ 90 Previous Transaction Value: £5	30 mins	No alert
10:45 am	Criminal	Transaction Value: £ 1000 Previous Transaction Value: £100	15 mins	No alert

There are no events in this sequence that match all the conditions in the rule. A better approach is to track the time since the last low-value transaction, and trigger the rule if you see a high-value transaction within 2 hours of a low-value one, regardless of whether there were any intervening transactions.

You can use a conditional assignment (see Conditional Assignment) to create state expressions that only store a value when a certain condition is true. This enables you to create an expression that stores the time of the last low-value transaction for each customer:

```
@eventType("transaction")
state.previousLowValueTransactionTime:
event.amount.baseValue <= 10 ?
event.eventTime
```

This expression stores the date and time only if the value of the transaction is less than or equal to 10. If the value is greater than 10 the state expression does not update. You can then create a version of the rule that refers to this state:

```
@eventType("transaction")
rules.testTransaction:
event.amount.baseValue > 100 &&
event.eventTime - state.previousLowValueTransactionTime < 2h
```

The rule now triggers according to the following sequence of events:

Time	Cardholder or criminal?	Transaction value change:	Time since previous transaction	Result
10:00 am	Criminal	Transaction Value: £ 5 Previous Transaction Value: N/A	N/A	No alert
10:30 am	Cardholder	Transaction Value: £ 90 Previous Transaction Value: £5	30 mins	No alert
10:45 am	Criminal	Transaction Value: £ 1000 Previous Transaction Value: £100	15 mins	**Alert**

The "N/A" annotation in the first row is because on the first transaction, the state expression has not yet been evaluated and returns a value of null. This causes the execution of the rule to stop when it reaches the reference to state.previousLowValueTransactionTime, meaning that the rule fails to execute and does not generate any output for this event. Whenever an expression refers to a state variable that has no value for the current entity, or references an event field that is not present in the current event, that undefined value can stop the execution of the expression.

Many state and other variable references return null.This is usually the intended behaviour as the test transaction rule does not trigger on the first transaction a customer makes, so when the rule execution stops this does not cause any issues.

However, in some cases, it is necessary to prevent references to state variables, event fields or other variables from returning null and causing the execution of the referring expression to halt. You can Specifying a Default Value or use the @defaultValue annotation (see @defaultValue) when defining a single-value state or global expressions. This ensures that, when an entity or entity type is first seen, the state variable already has a value.

4.6 Applying Global State and More Annotations

In this example, you create a rule which identifies unusually high spending over a given time period. For example, if normal behaviour indicated an average entity spend of about £100, you could specify a static threshold to trigger a rule if a transaction has a value over 500. However, this might cause problems during times when average spend tends to be higher, such as the run up to Christmas, Black Friday, or Singles' Day in China. It might be better in this case to track the average entity spending and compare the single transaction amount to this moving threshold.

To do this, you can define a global expression that stores state from all customers' transactions as in the following example:

```
@rollingAverage(24h)
@eventType("transaction")
```

```
globals.averageTransactionValue: event.amount.baseValue
```

You use the @rollingAverage annotation (see @rollingAverage) for calculating an average value instead of storing a value which gets overwritten each time a transaction is processed.

You can then define a rule that generates an alert for any transaction with a value more than five times the rolling average:

```
@alert
@eventType("transaction")
rules.highValueTransaction:
event.amount.baseValue > 5 * globals.averageTransactionValue
```

4.7 Using Collections

You can also create collections in AMDL expressions. Collections are useful for storing multiple values or comparing a field in the event data against a list of possible values. For example, you might want to generate an alert for a high-value transaction at a merchant within a high-risk category, as indicated by the Merchant Category Code (MCC):

```
@alert
@eventType("transaction")
rules.highValueTransactionHighRiskMCC:
event.amount.baseValue > 5 * globals.averageTransactionValue &&
[ "7999", "7995", "6051", "5912", "5933" ] ~# event.merchantCategoryCode
```

This rule generates an alert if the transaction value exceeds 5 times the global average, and the MCC is one of those listed in the array (using the "contains" operator, ~#). For more details, refer to the Collections section.

To use this list of high-risk MCCs in several expressions, or for separating this definition from the business logic of the rule itself, you could store it as a static value using the "values" scope (see Static Values):

```
values.highRiskMCCs: [ "7999", "7995", "6051", "5912", "5933" ]
```

You could then reference this static array in the rule to achieve the same effect as above:

```
@alert
@eventType("transaction")
rules.highValueTransaction_highRiskMCC:
event.amount.baseValue > 5 * globals.averageTransactionValue &&
values.highRiskMCCs ~# event.merchantCategoryCode
```

Note that the values expression above must be defined for the same entity type as this rule.

You can also create expressions that store collections in an entity or global state. For example, you can store a list of all the payment methods used with a DPAN/FPAN over the last year to trigger a rule whenever there is a large transaction using a payment method not seen before in that time. To store a list of payment methods, you can write a state expression using the @set annotation:

```
@set(365d)
@eventType("transaction")
state.paymentMethodsInLastYear: event.paymentMethod.methodId
```

You can then write a rule that checks whether the payment method of the current transaction event is contained in that set:

```
@alert
@eventType("transaction")
rules.highValueTransaction_newPaymentMethod:
event.amount.baseValue > 500 &&
state.paymentMethodsInLastYear !# event.paymentMethod.methodId
```

You use "does not contain" operator, !#, where the rule triggers if the value is over 500 and the payment method is not contained in the collection of payment methods for the current entity.

4.8 Applying Cross-Entity State References

In AMDL, each entity type has its own separate set of rules and state definitions. However, where more than one entity is present in an event, expressions defined for one entity type can refer to the state variables defined for other entities in the event. For example, if an event contains both a customer entity and a merchant entity, and the merchant has the following state definition:

```
@eventType("transaction")
@rollingAverage(1d)
state.avgTxnValue: event.amount.baseValue
```

A rule written against the customer entity type could refer to this state variable using the following syntax:

```
state.entities.<entity type>.<state variable>
```

This returns an array of the values of <state variable> for all entities of <entity type> in the event, even if there is only one entity of that type.

For example, an expression defined for the customer entity type could reference the merchant state defined above:

```
rules.transactionExceedsMerchantAverage:
event.amount.baseValue > state.entities.merchant.avgTxnValue.single()
```

The expression uses the single() method because the state.entities reference returns a collection, and this converts the collection into a single value.

You can test rules that use cross-entity in the AMDL unit test environment. You use the @entitytype annotation to specify the entity for the state expressions that are assigned values in the Initial State panel (see Entities in Unit Tests).

4.9 Rule References

An AMDL Business Rule expression can reference the result of a rule as a Boolean value.

```
rules.rule1AndHighValue: rules.rule1 && event.amount.baseValue > 100
```

Rule references in rule declarations cannot have any circular references. However, there is no restriction on rule references in state update expressions, because all rules are evaluated before state is updated.

Note that if a rule does not evaluate, any reference to that rule returns undefined. Default values can be assigned to rule outputs using the ?? operator.

For example, if rule1 in the example does not evaluate, it can be defaulted to 'false' as shown below:

```
rules.rule1AndHighValue: ( rules.rule1 ?? false ) && event.amount > 100
```

4.10 Outputting Values

IMPORTANT: Using the @output annotation to output a value is only permitted in Business Rules AMDL expressions.

You can use the @output annotation to output a calculated value (for example, the value of a transient variable), either as a tag (which is visible in the Fraud Transaction Monitoring Portal Incident Management page), or as part of the output produced by the fraud system in response to an event.

You can use @output in Business Rules expressions within the var or rules scope. When used on a var expression, the @output annotation causes output to be generated for any event for which the expression is evaluated. Thus, the output contains the value of the var expression for that event.

Output as a Tag

The default behaviour is to output the value as a tag, which is visible in the Fraud Transaction Monitoring Portal Incident Management page. You can enter an optional argument, which defines the outputted tag namespace. However, if you do not supply the argument, the namespace defaults to the name of the var expression.

For example, the expression below adds a tag with the namespace "Daily account position", containing the value of var.dailyPosition for that event, to all deposit and withdrawal events.

```
@eventType("deposit")
@eventType("withdrawal")
@output("Daily account position")
var.dailyPosition:
state.deposits24h.total() - state.withdrawals24h.total() +
event.amount.baseValue * ( event.eventType == "deposit" ? 1 : -1 )
```

You can use the @output annotation in a rule in the following manner:

- If the rule evaluates to true or false, the value output is true or false accordingly.
- If the rule does not evaluate, no output tag is generated.

However, using the annotation this way can generate a large numbers of tags, which can cause performance issues and may result in new tags no longer being stored or displayed for certain entities. This is because there is a configurable limit on the number of unique tag values stored for a given entity, and each distinct value output by an expression like the one above counts as a unique tag value. Once an entity has exceeded this limit, no further tags are displayed for this entity.

'Rules Output' Mode

Alternatively, you can use the 'rules output' mode of the `@output` annotation. This causes the expression to output its value in the output data from the Fraud Transaction Monitoring System, rather than as a tag. You can change the output mode by specifying a value for the mode argument. Valid options are `ruleoutput` and `tag` (for the tag output mode described above – this is the default).

Modifying the example above to use rule output mode gives:

```
@eventType("deposit")
@eventType("withdrawal")
@output(mode=ruleoutput)
var.dailyPosition:
state.deposits24h.total() - state.withdrawals24h.total() +
event.amount.baseValue * ( event.eventType == "deposit" ? 1 : -1 )
```

The output data is formatted as follows in the output from the Fraud Transaction Monitoring System (using the example above):

```
{
…"
models": [
{
"modelData": {
"dailyPosition": 568.25
}"
modelId": "businessrules",
…
},
…
],
…
}
```

Note that you cannot use this output mode for rules. You can only use it for transient variables.

5 Unit Tests for AMDL

The Analytics page in the Fraud Transaction Monitoring System enables you to use an offline tester to ensure AMDL expressions are behaving as expected before applying them to a live system. You can attach multiple tests to any AMDL expression for testing your Business Rules. You can create tests for any Rule Set in a Workflow for the AMDL expressions in the Rule Set Condition and in the main AMDL Block.

A test consists of a test event, optional tests on some initial state and the final state once the AMDL has been executed. The tests panel looks as follows:

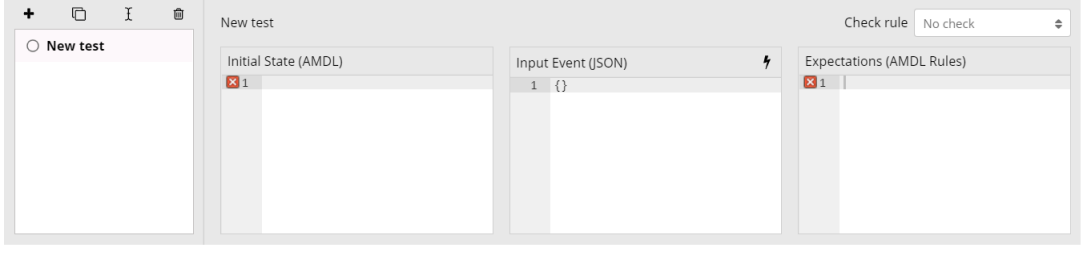


Figure 2: Unit Tests

The left-hand panel lists all of the tests that are attached to the AMDL expression or Rule Set. You enter the initial state, input event, and expectations in the middle left, middle right, and far right panels respectively. For more information on how to access the AMDL unit test panel, see the Fraud Transaction Monitoring Portal Guide.

You can create:

- **Positive test cases** – where an event should cause a rule to trigger, or a state variable or Support Variable to be set to a certain value.
- **Negative test cases** – where a rule should fail to trigger or a variable should not be set or updated.

Note: You can save tests alongside the expressions for which they are written, and export or import tests next to your Business Rules.

The Input Event

The most important component of any unit test is the input event. This is the event that your AMDL is evaluated against. The Input Event panel must contain JSON event data, and the event must have an `eventType` field and be of a type that is defined in your schema. However, the event is not validated against the schema, enabling you to use minimal events containing only the necessary fields, and even test potential schema changes.

The event generator tool that forms part of the AMDL unit tester lets you generate events that match your schema, and set the values of optional and mandatory fields to create appropriate test cases. Click **Event Generator** to generate a test event. This ensures that the test event closely approximates the real events that your AMDL will be evaluated against in production.

Expectations

Each test must also define the expected outcome(s) for comparing the result of the test evaluation with the expected result to determine a pass or a fail. These tests are set in different ways depending on whether you are testing rules or other kinds of expression in **AMDL Business Rules**. For more information, see Testing a Business Rule with a positive test case, Testing a Business Rule with a negative test case, and Testing state and global expressions in AMDL

Initial State

The Initial State panel allows you to define values for any other expressions referenced by the AMDL you are testing. Your unit tests have no access to real entities or a live state, or the values of any other AMDL expressions. Therefore, you must define values for any referenced state or global variables, transient variables, static values, data lists or support variables that you want to have a value in your test.

State and other variables are defined in exactly the same way they are in normal AMDL code. However, instead of an expression that evaluates to a value, you must set any variable to a fixed value in the Initial State panel. For example, you could set the value of a transient variable as follows:

```
var.exampleOfATransientVariableMyAMDLReferences: 12345
```

For static values (the "values" scope; see Static Values), you can simply copy and paste the values expression into the Initial State panel.

You can set the value of multiple different variables in various scopes by simply placing each expression on a separate line. For example:

```
var.exampleOfANumericalVariableMyAMDLReferences: 12345
var.exampleOfAStringVariableMyAMDLReferences: "my string"
values.exampleOfACollection: [ "a", "collection", "of", "strings" ]
```

Entities in Unit Tests

If you have expressions that refer to the state of multiple different entities or entity types, you can set values for these state expressions in the Initial State panel using an annotation specific to unit tests, `@entitytype`. This annotation takes two named arguments, the entity type (`type`) and the entity ID (`id`), although the entity ID is optional. Any state expressions defined below such an annotation in the "Initial State" panel belong to the entity specified in the `@entitytype` annotation.

For example, to specify the value of the state expression `avgTxValue` for a merchant entity with the entity ID "merchant1":

```
@entityType
(type="merchant", id="merchant1")
state.avgTxValue: 100
```

5.1 Testing a Business Rule with a Positive Test Case

Consider the following rule, which is similar to one you introduced in Example 4: Timestamps and Durations and Conditional State:

```
@alert
@eventType("transaction")
rules.testTransaction:
event.amount.baseValue > 100 &&
state.previousTransactionValue < 10 &&
event.eventTime - state.previousTransactionTime < 2h
```

There are a number of test cases you might want to create in order to ensure that this rule works correctly. The simplest is the positive case – a transaction event where the value is greater than 100, the previous transaction value was less than 10, and the previous transaction was less than 2 hours ago.

You first need to create the input event. For the positive test case, this means a transaction event where the value of `amount.baseValue` is greater than 100 (e.g. 150). The event is automatically generated with an `eventTime` equal to the current time. When creating the event, you can leave the placeholder values in all the other fields (similar to the following).

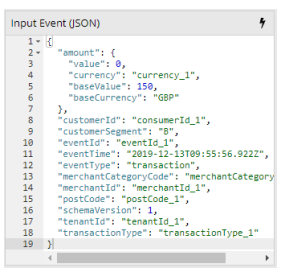


Figure 3: Positive Test Case

You also need to specify the values of the state expressions referenced in the rule for the positive test case. You can do this in the Initial State panel. The correct value of `state.previousTransactionValue` only needs to be less than 10. You also need to set `state.previousTransactionTime` to a date-time less than two hours ago by copying the date-time from the input event, which is 9:55 am on 13th Dec 2019, represented in ISO 8601 format as "2019-12-13T09:55:56.922Z". By reducing the hour by one, you can set the previous transaction time to 08:55 on the same day, and create a case in which the rule should trigger. This Initial State is shown below.

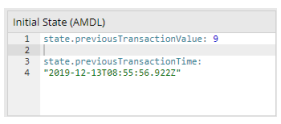


Figure 4: Positive Test Case

The final step in creating this test is to set the expected outcome. In this case, the rule should trigger, so you select "Check triggers" from the "Check rule" dropdown at the top right of the tests pane.

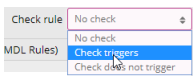


Figure 5: Check Triggers

Once the test is created, you can name this test and run it.

5.2 Testing a Business Rule with a Negative Test Case

While it's important to test that your rule triggers, you should also test when rule doesn't trigger in conditions where you anticipate it to generate an alert.

For the rule above, you can identify three cases to test:

- Transaction value is less than 100
- Previous transaction value is greater than 10
- Previous transaction time is more than 2 hours ago

To create a negative test case, you can duplicate the first test and then edit it (once you've created your first positive test case). By duplicating the positive test case, you can create each of the three negative test cases by:

- Changing the amount in the input event to 90
- Changing the value of state.previousTransactionValue to 11
- Changing the time of state.previousTransactionTime to 06:55 (i.e. change the date-time to "2019-12-13T09:55:56.922Z")

For each of these tests, you also need to change the "Check rule" setting to "Check does not trigger".

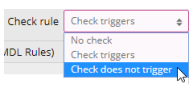


Figure 6: Check Does Not Trigger

You can test if the case where one or more of the state expressions referred to in the rule has no value (for example, when this is the first transaction for this particular entity). In the case of an optional field, you can test where the event data field referred to is not present.

You can create a simple test case of this kind by duplicating the initial positive test case, and deleting the expressions state.previousTransactionValue and state.previousTransactionTime from the Initial State panel. This action simulates the behaviour of the rule on the very first transaction for any given customer entity. You also need to change the "Check rule" setting to "Check does not trigger", as the rule should not trigger when these state variables are undefined.

Note that in this case, the rule stops executing when it reaches the reference to an undefined state. This results in a case where the rule doesn't trigger, and hence the test passes. However, the test results will also contain a warning that the rule did not execute. Whilst this is expected behaviour, a rule that fails to evaluate returns null when referred to in another expression (for details of the consequences of this, see Rule References).

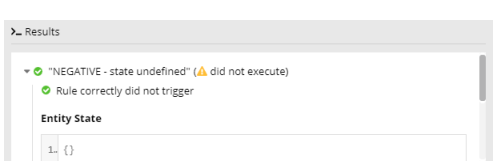


Figure 7: Rule not triggered

5.3 Testing State and Global expressions

You can also test that state and global expressions are updating as expected. For instance, you can test the state expression introduced in Example 4: Timestamps and Durations and Conditional State, which store the date-time of the most recent low-value transaction:

```
@eventType("transaction")
state.previousLowValueTransactionTime:
event.amount.baseValue < 10 ?
event.eventTime
```

There are two cases to test with this expression: Storage of the date-time of the transaction if the value is less than 10, and whether the stored value remain the same if the value is greater than or equal to 10,

As with rule testing, you provide some initial state and an event. When testing state and global expressions, however, rather than using the "Check rule" dropdown list, you instead use the Expectations panel. The Expectations panel must contain one or more AMDL rules, for evaluation after the input event is processed. These rules check that the final state is equal to the expected value. If all the rules in the Expectations panel trigger, the test passes. If any of these rules fail to trigger, the test fails. The names of these rules must be unique within the scope of a specific test, but can be repeated across multiple tests for the same entity.

For the first test case, you can set an initial value for the state you are testing, or simply leave the Initial State blank. The input event should be a transaction with a value less than 10.

```
state.previousLowValueTransactionTime:
"2019-10-15T12:34:56Z"
```

In the Expectations panel, you then write a rule that tests whether the final value of the state is what you expect it to be. In this case, the same as the date-time of the input event. You can copy the date-time directly from the input event JSON when creating the rule, giving something like this:

```
rules.newDateTimeSet:
state.previousLowValueTransactionTime == "2019-12-13T11:21:12.702Z"
```

Note that in the test results, the name of this rule is displayed, and if the rule triggers, a green check-mark is shown next to it. The final value of the state you are testing is also shown. Using a single expectation rule is by far the most common case, but it is possible to add additional rules to this panel to check other conditions to ensure the state was updated correctly.

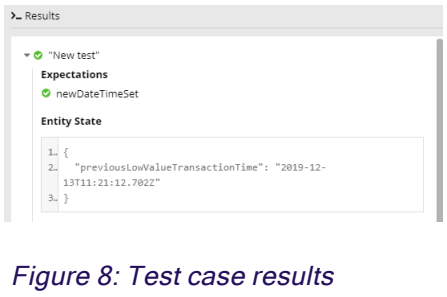


Figure 8: Test case results

You can also add another test to test the second case, where the transaction value exceeds 10. You can do this by duplicating the first test, changing the transaction value in the input event, and updating the expectation rule to check that the value of the state remains unchanged.

6 Appendices

This reference section provides information about AMDL including the:

- Data types used in AMDL (see Data Types in AMDL)
- Operators and their uses (see AMDL Operators)
- Scopes allowed in AMDL (see AMDL Scopes)
- Annotations and their descriptions (see AMDL Annotations)
- Methods associated with the variable types in AMDL (see AMDL Methods)

Further Information

For more information about:

- The syntax used by AMDL (see Getting Started with AMDL Syntax)
- Features that enable you to use advanced AMDL functionality and make rules easier to maintain (see Using AMDL in Advanced Expressions)
- How to use AMDL to write rules (see: Writing Business Rules in AMDL)
- How to write tests to check your AMDL rules (see Unit Tests for AMDL)

Appendix 1: Data Types in AMDL

All of the types that can be used within AMDL can be specified as fixed values. A full list is provided below.

Date Type	Examples
Arrays	values.array: [3, 1, 4, 1] These are ordered collections of values.
Booleans	values.testBool: true
Durations	values.days: 5d values.hours: 2h values.minutes: 15m values.seconds: 30s Durations can be specified as a certain number of days (d), hours (h), minutes (m), or seconds (s). Their main use is for comparison. For example, we may want to check whether the time between two events is less than one hour.
Numbers	values.testInt: 50 values.testDec: 50.365 Note that both integer and decimal numbers are permitted by the parser, and there is no difference in the way they are treated except when testing strict equality (see Type Coercion below).
Sets	values.allowedStrs: { "S1", "S2", "S3" } These are unordered collections of unique values.
Strings	values.testStr: "foo"

Type Coercion

AMDL is a weakly-typed language that performs coercion (implicit conversion) of data types where necessary. In practice, this means that you can write expressions like this and the AMDL interpreter will correctly convert variables from one type to another to ensure that comparisons work correctly:

```
// This evaluates to true
var.turnAStringIntoANumber:
"7.5" == 7.5 && "7.0" == 7 && "-7" == -7 && "7" + 1 == 8
```

Note that Booleans are coerced as shown below:

```
// This also evaluates to true
var.funWithBooleans:
"true" == true && "false" == false
&&
1 != true && 0 != false
```

However, note that the AMDL parser occasionally performs some unintuitive coercion. For example:

```
// This also evaluates to true
var.sevenIsNot7: "7" >= 7 && !( "7" == 7 )
```

The condition in parentheses is true because the string `"7"` is coerced to the integer `7`, but the integer representation in the AMDL code itself is coerced to the decimal number 7.0, which is not equal to the integer 7. The 'greater than or equal to' operator, however, triggers type coercion a second time, so the first condition is true.

Also, be aware that the type of a variable will not be coerced when applying a method. For example, execution of the following expression will fail:

```
var.numbersDontHaveAMethodCalled_isNumeric: 1.isNumeric()
```

Appendix 2: AMDL Operators

This section details the full list of operators within AMDL and their usage.

Operator Precedence

In the table below, all AMDL operators are listed in precedence order from highest to lowest. Operators in the same row of the table have the same precedence. Each operator is left, right, or non-associative as follows:

- Left-associative operators of the same precedence bind to the left: `e1 op e2 op e3` becomes `(e1 op e2) op e3`.
- Right-associative operators of the same precedence bind to the right: `e1 op e2 op e3` becomes `e1 op (e2 op e3)`.
- Non-associative operators do not appear in contexts where their binding is ambiguous.

Operator	Associativity
. ( )	left
[ ]	none
! ~	right
* /	left
+ - . . ~:	left
>
> =
<
< =	left
==
!=	left
Collection operators	right
&&	left
||	left
~?	right
??	right
? :	right

The following sections on operators describe what the above operators mean.

Mathematical Operators

Operators: `+ - * /`

As well as operating on numbers, + and - can also operate on times and durations. You can add or subtract durations, and add or subtract a duration to a time.

Example	Effect
`1 + 2`	Basic addition. This evaluates to 3.
`event.amount - 50`	Returns the value of the amount field in the event minus 50.
`event.amount.baseValue * state.storedValue`	Returns the `baseValue` field multiplied by some stored value.
`event.count /event.numberOfItems`	Returns a count divided by a number.

String Concatenation Operator

Operator: `..`

This operator concatenates any two strings, numbers, durations, or times as strings. For non-strings this is done in such a format that they can be coerced back to their original types.

Example	Effect
`"Hello " .. "World"`	Evaluates to "Hello World".
`event.firstName .. " " .. event.lastName`	Produces a full name from component parts.

Comparison and Equality Operators

These return a boolean value indicating the result of the comparison. Note that the first two operators can be applied to strings as well as numbers, durations, and times. However, the latter four cannot be used for strings.

Operator	Description	Example	Effect
==	equals	`1 == 1` `"foo" == "bar"`	This is always true. You can also check for equality amongst strings. The above is always false.
!=	does not equal	`event.phoneNumber != "999"`	Checking an event field is not equal to "999".
<	less than	`500 < event.amount.baseValue`	Checking an event field is more than a parameter value.
<=	or equal	`event.amount.baseValue <=param.thresholdValue`	Checking an event field does not exceed a parameter value.
>	greater than	`500 > event.amount.baseValue`	Checking an event field is less than a parameter value.
>=	greater than or equal	`500 >= event.amount.baseValue`	Checking an event field is equal to or less than a parameter value.

Note: Event fields can appear either side of the comparison operator.

Boolean Operators

These operators are used upon Boolean values.

Operator	Description	Example	Effect
&&	AND	`event.isFruit && event.isGreen`	This is only true if both fields are true.
||	OR	`state.isFruit || var.isGreen`	This is only false if both variables are false.
!	NOT	`!true`	This is always false.

Collection Membership Operators

These operators are used to test membership of a collection, and return a Boolean value. In usage, the left-hand operand must always be some form of collection, and the right-hand operand the element of the collection whose membership is being tested.

Operator	Description	Example	Effect
~#	contains	`event.transactionAmounts ~# 20` `values.declineStatusCodes ~# event.responseMessage`	Checking that a list in the event contains the value 20.00. Checking whether a string in the event is contained within a pre-defined collection of values.
!#	does not contain	`{ "GB", "US", "IS" } !# event.country`	Checking that the country field in the event is not one of "GB", "US" or "IS".

Collection Comparison Operators

These operators perform comparison operations upon all elements of a collection and return the Boolean combination of the results. The left operand must always be a collection and the right operand that which is to be compared.

Operator	Description	Example	Effect
==#	All equal to value	`[ 1, 1, 1, 1, 1 ] ==# 1`	All elements of the ordered collection are equal to 1, so this is true.
!=#	None equal to value	`{ "apple", "pear", "banana" } !=# "strawberry"`	None of the elements in the collection are equal to "strawberry", so this is true.
<#	All less than value	`event.transactionAmounts <# values.threshold`	Check that all values are less than to some pre-defined threshold.
<=#	All less than or equal to value	`event.transactionAmounts <=# values.threshold`	Check that all values are less than or equal to some pre-defined threshold.
>#	All greater than value	`event.transactionAmounts ># event.accountOpenDateTime`	Check that all values are greater than to some pre-defined datetime.
>=#	All greater than or equal to value	`event.transactionAmounts >=# values.threshold`	Check that all values are greater than or equal to some pre-defined threshold.

Ternary Operator

This operator allows one of two operations to be evaluated based upon a boolean condition. The operator is a useful control flow tool to use if, for instance, there are special cases that need treatment.

Operator	Description	Example	Effect
? :	Evaluate value x if boolean is true, else evaluate value y	`event.eventType == "registration" ? event.eventTime : event.regTime`	If the event type is "registration", this evaluates to the value of event.eventTime. However, if not, it evaluates to the event.regTime.
?		`event.eventType == "registration" ? event.eventTime`	The 'else' part of the ternary operation is optional. If the Boolean condition evaluates to false, expression evaluation stops.

Switch Case Operator

This operator allows one of many operations to be evaluated based upon a switch expression. This is useful if you have several special cases for the same variable, each of which needs different treatment.

Operator	Description	Example	Effect
~?	Evaluate multiple operations	`event.mcc ~? "7995": event.amount.baseValue > 150; "5912": event.amount.baseValue > 200; "4722": event.amount.baseValue > 5000; default: event.amount.baseValue > 500;`	Check the event amount against a different threshold based on the mcc field.
		`event.mcc ~? "7995": event.amount.baseValue > 150; "5912": event.amount.baseValue > 200; "4722": event.amount.baseValue > 5000;`	The 'default' case is optional. In this case, if the expression does not evaluate to one of the case labels, the expression evaluation stops.

Note: Case labels must be a fixed value or a 'default'. Currently, switching based upon variable expressions is not supported. Also note that every switch case must be terminated by a semicolon.

'Default Value' Operator

In AMDL, expression evaluation stops if a referenced variable does not exist. The ?? operator allows a default value to be specified if an expression cannot be evaluated.

Operator	Description	Example	Effect
??	This operator is used if the field does not exist (because data has not been populated). The operator is also used if the field contains null values.	`( event.amount.baseValue ?? 0 ) > 100`	If the event does not contain a field called amount.baseValue, this evaluates to 0 instead. This is particularly useful when dealing with optional fields that may or may not be present in the event. You can default to the value of another field or a

Operator	Description	Example	Effect
			state or other variable as well as a fixed value.
??	This operator is used if the field does not exist (because data has not been populated). The operator is also used if the field contains null values.	`( event.chargebackTime ?? event.eventTime ) < state.prevTime + 90d`	You can default to the value of another field or a state or other variable as well as a fixed value.

'Exists' Operator

This operator checks whether evaluation of a reference returns a non-null value.

Operator	Description	Example	Effect
~	Field Exists	`~event.optionalField`	Evaluates to true if the optionalField is present in the event being processed.
		`~state.previousTransactionValue`	Evaluates to true if the previousTransactionValue is defined in the state of the entity being processed.

Regular Expression Matches Operator

This operator checks whether the left string operand matches the regular expression contained in the right operand.

Note: This operator can only be applied to strings. For more information on regular expressions, see Outputting Values.

Operator	Description	Example	Effect
~=	Field matches regular expression	`event.address.postcode ~= "/^CB/"`	Checks whether the postcode starts with "CB".

Regular Expression Substitution Operator

This operator performs a substitution specified in the right operand on the left string operand.

Note: This operator can only be applied to strings. For more information on regular expressions, see Outputting Values.

Operator	Description	Example	Effect
~:	Substitute values in operand	`"1970.01.01" ~: "/-/./"`	Substitutes dots with dashes, resulting in "1970-01-01". Regexes can be quoted or unquoted.

Appendix 3: AMDL Scopes

The following is a list of allowed scopes in AMDL, detailing which are supported in Business Rules. A scope allows you to write expressions that refer to another value (or operand), or provide definitions. For example, the following is a scope as a reference `event.amount > 100` or `lists.negativeList ~# event.card`. The following shows how a scope is used for a definition, `rules.rule1: event.amount > 100` or `state.example: event.amount`). You can use certain variables for definition and reference, as highlighted in the following table.

Scope	Business Rules	
	Definition	Reference
event Event data fields		✓
rules Business rules	✓	✓
state Entity-level profile data	✓	✓
globals Population-level profile data	✓	✓
values Static values	✓	✓
features Feature extraction logic for third party models(see Features) This option is currently not available.	Only in Features Editor	✓
var Transient variables	✓	✓
lists Data lists	✓	✓
acglists Public/overridable data lists (MT)	✓	✓
external External service callouts This option is managed by Thredd.		✓
models Model outputs (including risk scores)		✓
param Workflow Parameters		
support Support Variables		
tests Unit test expressions		

* For more information, see Unit Tests for AMDL.

Appendix 4: AMDL Annotations

This section provides a complete list of annotations in AMDL, each of which is accompanied by a brief description.

@alert

This annotation must be applied to an AMDL rule to generate an alert when the rule triggers. If this is not applied to a rule, an alert is not generated and therefore the outcome is not be visible on the Incident List of the Fraud Transaction Monitoring System.

```
@alert
rules.dummyRule: 1 == 1
```

@array

- @array(<duration>)
- @array(<length>)
- @array(duration=<duration>, size=<length>)

This modifies the type of a state expression such that it captures an array, which in the context of AMDL refers to an ordered collection of objects. The size of an array must be limited by specifying a duration (in which case elements older than the specified duration will expire and be removed), or a fixed length (in which case the oldest element will be removed each time a new element is added, once the array reaches that length), or both (using named arguments, 'duration' and 'size').

If duration and length limits are both specified, the oldest element is removed if the array reaches its size limit when a new element is added. However, elements older than the duration are also be removed. A default size limit of 1000 elements is applied to all AMDL collections, including arrays. While this can be overridden by specifying a larger length limit, it is not recommended to do so, as this limit prevents performance issues due to large amounts of state data stored in a single collection.

```
@array(30d)
state.lastMonthsTransactions: event.amount.baseValue
```

@comment

- @comment(<comment>)

This annotation has no functionality, but can be used to add comments to AMDL expressions.

```
@comment
(
"place your comment here"
)
rules.dummyRule 1 == 1
```

@description

- @description(<description>)

This annotation provides a tool tip that appears when users hover over the Triggered Rules column in the Incident List.

```
@description("place your description here")
rules.dummyRule 1 == 1
```

@defaultValue

- @defaultValue(<defaultValue>)

Any single-value state or global expression can have a default value applied. However, the default value is referenced but does not yet exist using this annotation. If the state already exists, the default value is not applied.

Note: This annotation cannot be used with expressions that store collections of values, such as arrays, set or histograms.

```
@defaultValue
(
0)
state.myCount: event.amount.baseValue
```

@eventType

- @eventType(<eventType>)

An AMDL Business Rules expression accompanied by this annotation is only evaluated for events of the type specified within parentheses. Multiple such annotations can be used to allow evaluation on more than one defined events. If no `@eventType` annotation is provided for a Business Rules expression, it is evaluated for all event types by default.

```
@eventType(registration)
rules.dummyRule: 1 == 1
```

@firstValue

- @eventType(<eventType>)

Any state expression with this annotation is updated if it does not already exist. As such, it effectively captures the first value to which it is evaluated.

```
@firstValue
state.firstSeen: event.eventTime
```

@histogram

- @histogram(historyLength=<duration>, bucketSize=<bin width>)
- @histogram(historyLength=<duration>, bucketSize=<bin width>, timeField=<time field>)

This modifies the type of a state expression to a histogram, which in AMDL refers to a collection of accumulated values, bucketed by time. The `@histogram` annotation can take an optional argument, `timeField`, which allows you to define a non-default time field to use.

The following is an example of a global histogram that stores transactions over the last week as follows:

```
@histogram
(historyLength=7d, bucketSize=1d)
globals.txAmounts: event.amount.baseValue
```

For more information, see Using Histograms.

@initialContents

- @initialContents(<initialContents>)

Any state expression defined as an array or set can have initial contents – elements of the collection which are defined prior to the state expression being updated for the first time. If the state is referenced but does not exist yet, or when the state is updated for the first time, these values are assigned to the collection before the state expression is evaluated or updated.

```
@initialContents([0,0,0,0,0])
@array(5)
state.setTest: event.amount.baseValue
```

@mapOptions

- @mapOptions(keyDuration=<duration>, keySize=<size>)

When using a state or global expression to define a dynamic map, use this optional annotation to set a limit on the number of keys in the map.

Use the `keyDuration` argument to set a duration limit. The duration argument ensures that each time the map is updated, keys that were last updated before the specified duration are removed from the map.

Use the `keySize` argument to specify a fixed size limit. This ensures that once the number of keys stored in the map reaches this size, each time a new key is added, the key that was last updated longest ago is removed. If you use both arguments, keys are removed from the map according to whichever limit is reached first.

```
@eventType ("transaction")
@mapOptions (keyDuration=14d, keySize=100)
state.previousCurrencyAmount[ event.amount.currency ]:
event.amount.baseValue
```

A default size limit of 1000 keys is applied to all maps. While this can be overridden by specifying a larger size limit, it is not recommended to do so, as this can create performance issues due to large amounts of state data stored in a single map. This is particularly relevant for nested maps that store a collection of values with each key. For more information about maps, see Using Lookup Tables.

@output

- @output(<namespace>)
- @output(mode=ruleoutput)

This annotation modifies a var expression so that it assigns a tag with the value of that expression for each event for which it is calculated, or outputs the results in the Fraud Transaction Monitoring System output. If outputting as a tag, you can provide an optional namespace argument. If omitted, the tag generated will use the name of the var expression as its namespace.

This annotation modifies a var expression so that it assigns a tag with the value of that expression for each calculated event, or outputs the results in the fraud system output. If outputting as a tag, you can provide an optional namespace argument. If omitted, the generated tag uses the name of the var expression as its namespace.

```
@output("Twice the transaction amount")
var.twiceAmount: 2* event.amount.baseValue
```

@rollingAverage

- @rollingAverage(<duration>)

This modifies the type of a state expression to a rolling average. The rolling average provides the average of all values it is updated with, and is weighted so that the weight of each historical value decays with time.

```
@rollingAverage(7d)
globals.txAmounts: event.amount.baseValue
```

The rolling average is implemented internally as an exponentially decaying total and count value with a time period defined by the duration argument of the annotation. This means that expressions of this type do not require substantial storage space and therefore this is generally safe to use in high-throughput states such as globals.

The following equation shows the value of a rolling average for the i^{th} update of the expression (which happens at time t_i) is calculated as the ratio of the exponentially decayed total^T, and count^C :

$$A(t_i) = \frac{T(t_i)}{C(t_i)}$$

Both the exponentially decayed total and the exponentially decayed count for the rolling average are calculated from equations consisting of various coefficients. The exponentially decayed total is calculated as:

$$T(t_i) = \begin{cases} x_1 & i = 1 \\ x_i + \exp\left(-\frac{t_i - t_{i-1}}{\tau}\right) T(t_{i-1}) & i > 1 \end{cases}$$

The exponentially decayed count is calculated as:

$$C(t_i) = \begin{cases} 1 & i = 1 \\ 1 + \exp\left(-\frac{t_i - t_{i-1}}{\tau}\right) C(t_{i-1}) & i > 1 \end{cases}$$

T is the duration specified as an argument to the annotation, x_i is the value the update expression evaluates to (for example the transaction amount in the example above) and T_{i-1} signifies the time of the previous update of this expression.

@score

- @score(<score>)

This annotation enables rules to contribute to the overall risk score of a particular event or entity combination. A rule modified with this annotation adds the score specified in the argument to the risk score output by the 'businessrules' model. This model outputs the total of the scores assigned by all triggered rules.

```
@score(1.5)
rules.dummyRule: 1 == 1
```

@set

- @set(<duration>)
- @set(<length>)
- @set(duration=<duration>, size=<length>)

This annotation modifies the type of a state expression such that it captures a set, which in AMDL is an unordered collection of unique objects. Sets must be limited by specifying either a duration (in which case elements older than the specified duration will expire and be removed), or a fixed length (in which case the oldest element will be removed each time a new unique element is added, once the set reaches that length).

If a value which already exists is added to the set, the expiration timer upon that element will be reset. If duration and length limits are both specified, the oldest element will be removed if the set reaches its size limit when a new element is added, but elements older than the duration will also be removed. A default size limit of 1000 elements is applied to all AMDL collections, including sets.

```
@set(30d)
state.methodIdsLastMonth: event.methodId
```

@suppressAlert

This annotation modifies a rule where triggering for a particular event or entity combination, suppresses all alerts for that event or entity combination. This annotation is implemented as part of the implicit rules aggregator step, so applies to alerts generated by rules, models (risk thresholds), and aggregators.

```
@suppressAlert
rules.noAlertsOnWhitelist: lists.whitelist ~# event.customerId
```

@suppressTag

- @suppressTag(<namespace>="<tagValue>")

This annotation modifies a rule where triggering for a particular event or entity combination, suppresses a particular tag value (specified in the argument to this annotation). The namespace is optional. If no namespace is given, this annotation suppresses the tag with the default namespace and the specified value. This annotation is implemented as part of the implicit rules aggregator step, so applies to tags assigned by rules, models (risk thresholds), and aggregators.

```
@suppressTag
(action="deny")
rules.noDeclinesOnWhiteList: lists.whitelist ~# event.customerId
```

@tag

- @tag(<namespace>="<tagValue>")

This annotation is only applicable to rule expressions. If a rule triggers, the tag is appended to the response. The namespace is an optional argument. If no namespace is given (`@tag("<tagValue>")`), then this annotation is assigned to the default namespace (`_tag`). Multiple tag annotations can form part of any AMDL rule.

```
@tag(action="deny")
rules.dummyRule: 1 == 1
```

Appendix 5: AMDL Methods

Several of the variable types within AMDL have methods attached to them. All of the methods described below are case insensitive. For example, `"some string".endsWithIgnoreCase("ING")` is the same as `"some string".endswithIgnorecase("ING")` (see String.endsWithIgnoreCase(suffix)).

String Methods

The following group of methods can be applied to strings. If applied to something which isn't a string, the AMDL expression stops the evaluation. Implementation of these methods mostly follows the Apache commons-lang `StringUtils` public class (see https://commons.apache.org > StringUtils). Where details are not given, refer to the `StringUtils` documentation. Terms such as "lowercase" and "uppercase" are used according to the definitions given by the Unicode consortium (see https://docs.oracle.com > Characters).

This section is divided into:

- General string methods (below);
- Date-time methods for manipulating dates and date-times (see Date-time methods);
- String similarity methods for calculating quantitative measures of string similarity (see String similarity);
- Password methods for storing and checking hashed passwords (see Passwords)
- Compression methods for compressing and decompressing strings (see Compression).

String.abbreviate(maxWidth)

Where there is a requirement to abbreviate long strings in code, strings can be of a length less than or equal to the provided `maxWidth`. If abbreviated, the method uses ellipses instead of the `maxWidth`.

```
rules.alwaysTrue:
"some string".abbreviate(11) == "some string" &&
"some string".abbreviate(7) == "some..."
```

String.capitalize()

If the first letter is alphabetical, it is capitalised, otherwise the method leaves the string alone. This is useful, for example, for matching text that is capitalised/not capitalised in a data feed.

```
rules.alwaysTrue:
"some_string".capitalize() == "Some_string" &&
"1some_string".capitalize() == "1some_string" &&
"Some_string".capitalize() == "Some_string"
```

String.center(size)

Centres a string inside a larger string of a `size`. Extra characters are evaluated by the method as spaces " ".

```
rules.alwaysTrue: "some string".center(15) == " some string "
```

String.charAt(n)

Returns the character at position n in the string (first character is at index 0).

```
rules.alwaysTrue: "some string".charat(5) == "s"
```

String.chomp()

Removes one newline from the end of the string if one exists, otherwise the method leaves the string alone.

```
rules.alwaysTrue:
"Some string
".chomp() == "Some string".chomp()
```

String.contains(substr)

Checks whether the substring `substr` is present within the string.

```
rules.alwaysTrue:
"some string".contains("str") == true &&
```

```
"some string".contains("foo") == false
```

String.containsIgnoreCase(substr)

Checks whether the substring substr is present within the string, irrespective of case.

```
rules.alwaysTrue:
"some string".containsIgnoreCase("STR") == true &&
"some string".containsIgnoreCase("FOO") == false
```

String.containsAnyChars(chars)

Checks whether any of the characters in the chars string are present in the string.

```
rules.alwaysTrue:
"some string".containsAnyChars("sr") == true &&
"some string".containsAnyChars("x1") == false
```

String.containsNoneChars(chars)

Checks that none of the characters in the chars string are present in the string. This method is the opposite of String.containsanychars (chars).

```
rules.alwaysTrue:
"some string".containsNoneChars("x1") == true &&
"some string".containsNoneChars("sr") == false
```

String.countMatches(substr)

Counts how many times the substring substr appears in the string.

```
rules.alwaysTrue: "some strange string".countMatches("tr") == 2
```

String.difference(string2)

Compares the string with string2 and returns the remainder of string2 from the point where the two strings differ.

```
rules.alwaysTrue:
"some string".difference("something else") == "thing else" &&
"some string".difference("string") == "tring" &&
"some string".difference("other") == "other"
```

String.endsWith(suffix)

Checks whether the string ends with the provided suffix.

```
rules.alwaysTrue: "some string".endsWith("ing") == true
```

String.endsWithIgnoreCase(suffix)

Checks whether the string ends with the provided suffix, irrespective of case.

```
rules.alwaysTrue: "some string".endsWithIgnoreCase("ING") == true
```

String.entropy()

Returns the ideal Shannon entropy of the string. The Shannon entropy is an information theory to measure the uncertainty of a random process.

```
rules.alwaysTrue: "fooBAR123".entropy() > 1
```

```
rules.alwaysTrue: "foofoofoo".entropy() < "fooBAR123".entropy()
```

String.equals(str)

Compares the string with the argument string, returning true if they represent identical sequences of characters.

```
rules.alwaysTrue: "some string".equals("some string") == true
```

String.equalsIgnoreCase(str)

Compares the string with the argument string, returning true if they represent equal sequences of characters, irrespective of case.

```
rules.alwaysTrue: "some string".equalsIgnoreCase("Some String") == true
```

String.format(args)

Requires the string to be a valid format string. The method then uses the arguments `args` in the format string according to `Java.lang.String.format()` (standard Java formatting, see https://docs.oracle.com: Java Development Kit version 11 API Specification).

If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero.

```
rules.alwaysTrue:
"a: %s, b: %s, c: %s".format("example",1,"???") == "a: example, b: 1, c:
???"
```

String.geodistance(lat1, long1, lat2, long2)

Returns the distance (in km) between two points on the Earth's surface specified by latitude and longitude (in degrees). This is where `lat1` and `long1` are the latitude and longitude of point 1, and `lat2` and `long2` are the latitude and longitude of point 2. The string that this method operates on is not used (you may simply use an empty string).

```
rules.alwaysTrue: "".geodistance(90, 0, -90, 0) == 20015.086796020572
```

String.isAllLowercase()

Checks if the string contains only lowercase letters.

```
rules.alwaysTrue:
"somestring".isAllLowercase() == true &&
"someString".isAllLowercase() == false &&
"some string".isAllLowercase() == false &&
"somestring1".isAllLowercase() == false
```

String.isAllUppercase()

Checks if the string contains only uppercase letters.

```
rules.alwaysTrue:
"SOMESTRING".isAllUppercase() == true &&
"SomeString".isAllUppercase() == false &&
"SOME STRING".isAllUppercase() == false &&
"SOMESTRING1".isAllUppercase() == false
```

String.isAlpha()

Checks if the string contains only Unicode letters.

```
rules.alwaysTrue: "SomeString".isAlpha() == true &&
"Some String1".isAlpha() == false
```

String.isAlphanumeric()

Checks if the string contains only Unicode letters or digits.

```
rules.alwaysTrue:
"SomeString1".isAlphanumeric() == true &&
"Some String1".isAlphanumeric() == false
```

String.isAlphanumericSpace()

Checks if the string contains only Unicode letters, digits, or spaces.

```
rules.alwaysTrue:
"Some String1".isAlphanumericSpace() == true &&
"Some String1&".isAlphanumericSpace() == false
```

String.isAlphaSpace()

Checks if the string contains only Unicode letters or spaces. This does not check for any digits in the string (unlike `String.isAlphaSpace()`).

```
rules.alwaysTrue:
"Some String".isAlphaSpace() == true &&
"Some String1".isAlphaSpace() == false
```

String.isAsciiPrintable()

Checks if the string contains only ASCII printable characters. ASCII printable characters are the 85 characters in ASCII format that can printed on paper or displayed on a screen.

```
rules.alwaysTrue: "Some ASCII".isAsciiPrintable() == true
```

String.isBlank()

Checks if the string is whitespace or empty. Note that, unlike StringUtils, null evaluation of the string results in non-evaluation of the rule.

```
rules.alwaysTrue:
"".isBlank() == true &&
" ".isBlank() == true &&
" some string".isBlank() == false
```

String.isEmpty()

Checks if the string is empty.

```
rules.alwaysTrue:
"".isEmpty() == true &&
" ".isEmpty() == false
```

String.isNotBlank()

Checks if the string is not empty and not whitespace only. This method is the opposite of String.isblank ().

```
rules.alwaysTrue:
" some string".isNotBlank() == true &&
" ".isNotBlank() == false &&
"".isNotBlank() == false
```

String.isNotEmpty()

Checks if the string is not empty. This method is the opposite of String.isempty().

```
rules.alwaysTrue:
" ".isNotEmpty() == true &&
"".isNotEmpty() == false
```

String.isNumeric()

Checks if the string contains only Unicode digits.

```
rules.alwaysTrue:
"123456".isNumeric() == true &&
"string123".isNumeric() == false
```

String.isNumericSpace()

Checks if the string contains only Unicode digits or spaces.

```
rules.alwaysTrue:
"123 456".isNumericSpace() == true &&
"string 123".isNumericSpace() == false
```

String.isWhitespace()

Checks if the string contains only whitespace.

```
rules.alwaysTrue:
"".isWhitespace() == true &&
" ".isWhitespace() == true &&
"some string".isWhitespace() == false
```

String.left(len)

Returns the leftmost len characters of the string.

```
rules.alwaysTrue: "some string".left(7) == "some st"
```

String.leftPad(n)

Left pad the string with spaces, up to a total length `n`.

```
rules.alwaysTrue: "abc".leftPad(6) = "  abc"
```

String.length()

Returns the number of characters in the string.

```
rules.alwaysTrue: "some_string".length() == 11
```

String.lowercase()

Converts alphabetical characters in the string to lower case.

```
rules.alwaysTrue: "Mr. Smith".lowercase() == "mr. smith"
```

String.md5()

Returns the MD5 checksum of the string.

```
rules.alwaysTrue: "some str".md5() == "e05679f1d1deca304db99ce2cc19a7c9"
```

String.ngram(n, accepted_characters)

Implicitly normalises the string as described in String.normaliseChars(accepted_ characters), before returning a collection of possible n-grams contained within the normalised string.

```
rules.alwaysTrue: "FOOBAR123".ngram(3, "fba") ~# "fba"
```

```
rules.alwaysTrue:
"AbCdEaBcDe".ngram(2, "ace") == ["ac", "ce", "ea", "ac", "ce"]
```

String.normaliseChars(accepted_characters)

Converts alphabetical characters in the string to lower case. The method then removes any characters that are not in the argument string accepted_characters.

```
rules.alwaysTrue: "FOOBAR123".normaliseChars("for") == "foor"
```

```
rules.alwaysTrue: "AbCdE12345aBcDe".normaliseChars("abd31") == "abd13abd"
```

String.remove(substr)

Removes all appearances of the substring substr from the string.

```
rules.alwaysTrue: "some strange string".remove("tr") == "some sange sing"
```

String.removeEnd (substr)

Removes substring substr from the end of the string if it appears at the end.

```
rules.alwaysTrue: "starting string".removeEnd("ing") ==
"starting str"
```

String.removeEndIgnoreCase(substr)

Removes substring substr from the end of the string if it appears at the end, irrespective of case.

```
rules.alwaysTrue:
"starting string".removeEndIgnoreCase("ING") == "starting str"
```

String.removePattern(patt)

Removes each substring of the string that matches the given regular expression patt using dotall mode (matches any character, including line terminators).

```
rules.alwaysTrue: "some string".removePattern("[sgi]") == "ome
trn"
```

String.removePunctuation()

Removes punctuation characters from the string.

```
rules.alwaysTrue:
"!some.,punc{tu'ation?".removePunctuation() == "somepunctuation"
```

String.removeStart(substr)

Removes substring substr from the start of the string if it appears at the start.

```
rules.alwaysTrue: "starting string".removeStart("st") ==
"arting string"
```

String.removeStartIgnoreCase(substr)

Removes substring substr from the start of the string if it appears at the start, irrespective of case.

```
rules.alwaysTrue:
"starting string".removeStartIgnoreCase("ST") == "arting string"
```

String.repeat(n)

Repeats the string n times as in a new string.

```
rules.alwaysTrue: "abc".repeat(2) == "abcabc"
```

String.replace(a, b)

Replaces all occurrences of string a with string b.

```
rules.alwaysTrue: "abc".repeat(2) == "abcabc"
```

String.replacePattern(pat, rep)

Replaces each substring substr of the string that matches regular expression pat with the replacement rep using dotall mode (matches any character, including line terminators).

```
rules.alwaysTrue:
"some string".replacePattern("ing$", "obe") == "some strobe"
```

String.reverse()

Reverses the order of the characters in the string.

```
rules.alwaysTrue: "abcdefg".reverse() == "gfedcba"
```

String.reverseDelimited(delimiter)

Reverses a string separated by delimiter delimiter. The strings between the delimiters are not reversed.

```
rules.alwaysTrue: "a.b.c.d".reverseDelimited(".") == "d.c.b.a"
```

String.right(len)

Returns the rightmost len characters of the string.

```
rules.alwaysTrue: "some string".right(4) == "ring"
```

String.rightPad(n)

Right pad the string with spaces, up to a total length of n.

```
rules.alwaysTrue: "abc".rightPad(6) == "abc "
```

String.sha256()

Returns the SHA-256 hash of the string.

```
rules.alwaysTrue:
"some str".sha256() ==
"4ad27ac64e74640fbe5f24e205abdbf30effb67ddcced744ba05d8455cb7eb8a"
```

String.sequenceProbability(probabilities, accepted_characters)

Given a NxN probability matrix (such as a 2D array) and an accepted_characters string of length N, this method returns the Markov transition probability from the string. The NxN probability matrix is used for calculating outcomes based on the positions of a table. The Markov transition probability calculates future states based on transition from the current state.

```
rules.alwaysTrue:
"fooBAR123".sequenceProbability([[0.3, 0.7], [0.2, 0.4]], "fr") > 0.1
```

String.split(delimiter)

Splits the string by the delimiter delimiter into an ordered collection of strings.

```
rules.alwaysTrue: "some string".split("st") == [ "some,", "ring" ]
```

String.splitByChars(chars)

Splits the string by any of the specified characters chars into an ordered collection of strings.

```
rules.alwaysTrue:
"some string".splitByChars("si") == [ "ome ", "tr", "ng" ]
```

String.splitByCharacterType()

Splits the string by character type into an ordered collection of strings.

```
rules.alwaysTrue:
"string123!".splitByCharacterType() == [ "string", "123", "!" ]
```

String.splitByCharacterTypeCamelCase()

Splits the string by character type (including camel case words) into an ordered collection of strings.

```
rules.alwaysTrue:
"SStringString123".splitByCharacterTypeCamelCase() == [ "S", "String",
"String", "123" ]
```

String.startsWith(prefix)

Checks whether the string starts with the provided prefix.

```
rules.alwaysTrue: "some string".startsWith("som") == true
```

String.startsWithIgnoreCase(prefix)

Checks whether the string starts with the provided prefix, irrespective of case.

```
rules.alwaysTrue: "some string".startsWithIgnoreCase("SO") ==
true
```

String.strip()

Removes whitespace from the start and end of the string.

```
rules.alwaysTrue: " some string ".strip() == "some string"
```

String.stripAccents()

Removes diacritics from the string.

```
rules.alwaysTrue: "Et ça sera sa moitié.".stripAccents() == "Et
ca sera sa moitie."
```

String.stripCharsStart(chars)

Strips any of the provided characters chars from the start of the string.

```
rules.alwaysTrue: "some string".stripCharsStart("os") == "me
string"
```

String.stripCharsEnd(chars)

Strips any of the provided characters chars from the end of the string.

```
rules.alwaysTrue: "some string".stripCharsStart("os") == "me
string"
```

String.substring(start)

Returns the substring `substr` after index start.

```
rules.alwaysTrue: "abcdefghi".substring(3) == "defghi"
```

String.substring(start, end)

Returns the substring `substr` between indices `start` and `end`.

```
rules.alwaysTrue: "abcdefghi".substring(3, 7) == "defg"
```

String.substringAfter(sep)

Returns the substring `substr` after the first occurrence of the separator `sep`.

```
rules.alwaysTrue: "a.b.c".substringAfter(".") == "b.c"
```

String.substringAfterLast(sep)

Returns the substring `substr` after the last occurrence of the separator `sep`.

```
rules.alwaysTrue: "a.b.c".substringAfterLast(".") == "c"
```

String.substringBefore(sep)

Returns the substring `substr` before the first occurrence of the separator `sep`.

```
rules.alwaysTrue: "a.b.c".substringBefore(".") == "a"
```

String.substringBeforeLast(sep)

Returns the substring `substr` before the last occurrence of the separator `sep`.

```
rules.alwaysTrue: "a.b.c".substringBeforeLast(".") == "a.b"
```

String.substringBetween(arg1)

Returns the substring `substr` nested between the first two instances of `arg1`.

```
rules.alwaysTrue: "abcdefabcdef".substringBetween("ab") ==
"cdef"
```

String.substringBetween(arg1, arg2)

Returns the substring `substr` nested between the first instances of `arg1` and `arg2`.

```
rules.alwaysTrue: "abcdefghi".substringBetween("bc", "h") ==
"defg"
```

String.swapCase()

Swaps the case of each string character, changing lower case characters to upper case and vice versa.

```
rules.alwaysTrue:
"a string".swapCase() == "A STRING" &&
"A STRING".swapCase() == "a string" &&
"a StRiNg".swapCase() == "A sTrInG"
```

String.trim()

Removes start and end characters below ASCII code 32 from the string.

```
rules.alwaysTrue: "some string ".trim() == "some string"
```

String.uncapitalize()

If the first letter of the string is alphabetical, it is changed to lower case, otherwise it is left alone.

```
rules.alwaysTrue: "Some string".uncapitalize() == "some string"
```

String.uppercase()

Converts alphabetical characters in the string to upper case.

```
rules.alwaysTrue: "Some string".uppercase() == "SOME STRING"
```

Date-time Methods

These methods can be used on strings which represent a timestamp. The methods accept an optional format argument, which defines the format of the string the method is acting on (and any date-time arguments) and how it should be parsed, as well as the format of any date-time output. Without this, ISO-8601 format and UTC is assumed. The format argument can be any of the following:

- A format string using Java's DataTimeFormatter syntax (see https://docs.oracle.com: Java API Specification > DateTimeFormatter).
- "iso8601" for ISO-8601 format.
- "epoch" for a timestamp expressed as a number of milliseconds since the Unix epoch (midnight UTC on 1st Jan 1970).
- *"epochsec"* for a timestamp expressed as a number of seconds since the Unix epoch.

String.isTimeAfter(time)

Returns true if the timestamp string is after the date-time time. The method otherwise returns false.

```
rules.alwaysTrue:
"2024-01-01T01:00:00Z".isTimeAfter( "2024-01-01T00:00:00Z" ) == true
```

```
rules.alwaysTrue:
"2024/01/01 01:00".isTimeAfter( "2024/01/01 00:00", "yyyy/MM/dd HH:mm" )
== true
```

String.isTimeBefore(time)

Returns true if the timestamp string is before the date-time time. The method otherwise returns false.

```
rules.alwaysTrue:
"2024-01-01T00:00:00Z".isTimeBefore( "2024-01-01T00:30:00Z" ) == true
```

```
rules.alwaysTrue:
"2024/01/01 00:00".isTimeBefore( "2024/01/01 00:10", "yyyy/MM/dd HH:mm" )
== true
```

String.timeAddMilli(ms)

Adds a number of millisecond to the timestamp string.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeAddMili(100, "dd/MM/yy HH:mm:ss:SSS") == "01/01/24 00:00:00:100"
```

String.timeAddSec(s)

Adds a number of seconds to the timestamp string.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeAddSec(10, "dd/MM/yy HH:mm:ss") == "01/01/24 00:00:10"
```

String.timeAddMin(mins)

Adds a number of minutes to the timestamp string.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeAddMin(50) == "2024-01-01T01:50:00Z"
```

String.timeAddHour(hours)

Adds a number of hours to the timestamp string.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeAddHour(12) == "2024-01-01T12:00:00Z"
```

String.timeAddDay(days)

Adds a number of days to the timestamp string.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeAddDay(10, "dd/MM/yy HH:mm") == "11/01/24 00:00"
```

String.timeDiffMilli(time)

Returns the time between the timestamp string and the argument, in the units specified (rounded down to the nearest integer). This method is for miliseconds.

```
rules.alwaysTrue:
"01/01/24 00:00:00".timeDiffMili("01/01/24 00:00:00:100", "dd/MM/yy HH:mm:ss:SSS")
== 100
```

String.timeDiffSec(time)

Returns the time between the timestamp string and the argument, in the units specified (rounded down to the nearest integer). This method is for seconds.

```
rules.alwaysTrue:
"01/01/24 00:00:00".timeDiffSec("01/01/24 00:10:30", "dd/MM/yy HH:mm:ss")
== 630
```

String.timeDiffMin(time)

Returns the time between the timestamp string and the argument, in the units specified (rounded down to the nearest integer). This method is for minutes.

```
rules.alwaysTrue:
"01/01/24 00:00:00".timeDiffMin("01/01/24 00:05:00", "dd/MM/yy HH:mm:ss")
== 5
```

String.timeDiffHour(time)

Returns the time between the timestamp string and the argument, in the units specified (rounded down to the nearest integer). This method is for hours.

```
rules.alwaysTrue:
"2024-01-01T00:00:00Z".timeDiffhour("2024-01-05T12:00:00Z") == 108
```

String.timeDiffDay(time)

Returns the time between the timestamp string and the argument, in the units specified (rounded down to the nearest integer). This method is for days.

```
rules.alwaysTrue:
"2024/01/01 01:00".timeDiffDay("2024/01/10 01:00") == 10
```

String.timeFormat(format)

Returns the timestamp string, formatted according to the format string specified. If a second format string is specified, this represents the format of the timestamp string to be converted. Note that this format string has the same possible values as the optional format string that can be added to any of these date-time methods.

```
rules.alwaysTrue:
"2024-04-02T01:23:00Z".timeFormat("yyyy MM dd HH:mm") == "2024 Apr 2
01:23"
```

```
rules.alwaysTrue:
"2024/04/02 01:23".timeFormat("utc", "yyyy/MM/dd HH:mm") == "2024-04-
02T01:23:00Z"
```

String.timeGetMilli()

Gets the timestamp string in miliseconds.

```
rules.alwaysTrue:
"03:11:54:100 1 Jan 24".timeGetMili("HH:mm:ss:SSS d MMM yy") == 100
```

String.timeGetSec()

Gets the timestamp string in seconds.

```
rules.alwaysTrue:
"02:50:40 1 Jan 24".timeGetSec("HH:mm:ss d MMM yy") == 40
```

String.timeGetMin()

Gets the timestamp string in minutes.

```
rules.alwaysTrue:
"01:30:25 1 Jan 24".timeGetMin("HH:mm:ss d MMM yy") == 30
```

String.timeGetHour()

Gets the timestamp string in hours.

```
rules.alwaysTrue:
"2024-01-01T01:30:25Z".timeGetHour() == 1
```

String.timeGetDay()

Returns the millisecond, second, minute, hour and day part of the string timestamp.

```
rules.alwaysTrue:
"2024-01-11T01:30:25Z".timeGetDay() == 11
```

String.timeGetField()

Returns any of the standard set of Java temporal fields (see docs.oracle.com: Java API Specification > ChronoField) from the string timestamp.

```
rules.alwaysTrue:
"2024-04-02T01:23:45".timeGetField("DAY_OF_YEAR") == 92
```

```
rules.alwaysTrue:
"2024/04/02 01:23".timeGetField("DAY_OF_YEAR", "yyyy/MM/dd HH:mm") == 92
```

String.timeNow()

Returns the current time (using the format specified by the argument if provided –defaulting to ISO-8601). The string operated on has no effect on the return value, so the empty string can be used.

```
rules.trueOn10thApril2023: "".timeNow("d MMM yyyy") == "10 Apr
2023"
```

String.timeZoneChange(zone)

Returns the timestamp string, but with the time zone changed to the time zone zone.

```
rules.alwaysTrue:
"2023-04-02T01:23:00Z".timeZoneChange("+05:00") == "2023-04-
02T01:23:00+05:00"
```

String Similarity

String.cosineDistance(str)

Returns the cosine distance cosineDistance between the string and str. This is calculated as 1 which is the cosine similarity of the two strings, with each string represented as a set of 3-shingles (trigrams).

```
rules.alwaysTrue:
"Some string".cosineDistance("Another string") > 0.5 &&
"some string".cosineDistance("Some other string") < 0.32
```

String.cosineDistanceIgnoreCase(str)

Returns the cosine distance cosineDistance between the string and str, ignoring case.

```
rules.alwaysTrue:
"Some string".cosineDistanceIgnoreCase("Another string") > 0.5 &&
"Some string".cosineDistanceIgnoreCase("Some other string") < 0.32
```

String.jaccardIndexDistance(str)

Returns the Jaccard index distance jaccardIndexDistance between the string and str, calculated as 1 which is the Jaccard index for the two strings. Each string represented as a set of 3-shingles (trigrams).

```
rules.alwaysTrue:
"Some string".jaccardIndexDistance("Another string") > 0.68 &&
"Some string".jaccardIndexDistance("Some other string") <= 0.5
```

String.jaccardIndexDistanceIgnoreCase(str)

Returns the Jaccard index distance jaccardIndexDistance between the string and str, ignoring case. The Jaccard index is a statistic used for gauging the similarity and diversity of sample sets.

```
rules.alwaysTrue:
"SOME STRING".jaccardIndexDistanceIgnoreCase("Another string") > 0.68 &&
"SOME STRING".jaccardIndexDistanceIgnoreCase("Some other string") <= 0.5
```

String.jaroWinklerDistance(str)

Returns the Jaro-Winkler distance jaroWinklerDistance between the string and str, calculated as 1 which is the Jaro-Winkler similarity for the two strings. Jaro-Winkler is a string distance function used in statistics.

```
rules.alwaysTrue:
"Some string".jaroWinklerDistance("Another string") > 0.25 &&
"Some string".jaroWinklerDistance("Some other string") <= 0.25
```

String.jaroWinklerDistanceIgnoreCase(str)

Returns the Jaro-Winkler distance jaroWinklerDistance between the string and str, ignoring case.

```
rules.alwaysTrue:
"SOME STRING".jaroWinklerDistanceIgnoreCase("Another string") > 0.25 &&
"SOME STRING".jaroWinklerDistanceIgnoreCase("Some other string") <= 0.25
```

String.levenshtein(str)

Returns the Levenshtein levenshtein distance between the string and str.

```
rules.alwaysTrue: "Some String!".levenshtein("same sprung") == 6
rules.alwaysTrue: "Some String!".levenshtein("same sprung", true) == 0.5
```

String.levenshtein(str,normalised)

If normalised is true, returns the Levenshtein levenshtein distance between the string and str, normalised to [0-1] by dividing by the length of the longer of the two strings (the maximum possible value of the Levenshtein distance).

```
rules.alwaysTrue: "Some String!".levenshtein("same sprung") == 6
rules.alwaysTrue: "Some String!".levenshtein("same sprung", true) == 0.5
```

String.levenshteinIgnoreCase(str)

Returns the Levenshtein levenshtein distance between the string and str, ignoring character case. Levenshtein is a string distance function used for measuring the difference between two sequences.

```
rules.alwaysTrue: "Some String!".levenshteinIgnoreCase("same sprung!") == 3
rules.alwaysTrue: "Some String!".levenshteinIgnoreCase("same sprung!",
true) == 0.25
```

String.levenshteinIgnoreCase(str,normalised)

If normalised is true, returns the Levenshtein levenshtein distance between the string and str, ignoring character case and normalised to [0-1] as above.

```
rules.alwaysTrue: "Some String!".levenshteinIgnoreCase("same sprung!") == 3
rules.alwaysTrue: "Some String!".levenshteinIgnoreCase("same sprung!",
true) == 0.25
```

String.ratcliffObershelpDistance(str)

Returns the Ratcliff/Obershelp ratcliffObershelpDistance between the string and str, calculated as 1 which is the Ratcliff/Obershelp similarity of the two strings. Ratcliff/Obershelp is a string-matching algorithm for determining the similarity of two strings

```
rules.alwaysTrue:
"Some string".ratcliffObershelpDistance("Another string") > 0.25 &&
"Some string".ratcliffObershelpDistance("Some other string") <= 0.25
```

String.ratcliffObershelpDistanceIgnoreCase(str)

Returns the Ratcliff/Obershelp ratcliffObershelpDistance distance between the string and str, ignoring case.

```
rules.alwaysTrue:
"SOME STRING".ratcliffObershelpDistanceIgnoreCase("Another string") > 0.25
&&
"SOME STRING".ratcliffObershelpDistanceIgnoreCase("The string") <= 0.25
```

String.sorensenDiceDistance(str)

Returns the Sørensen-Dice distance sørensenDiceDistance between the string and str, calculated as 1, which is the Sørensen-Dice coefficient for the two strings. Each string represented as a set of 3- shingles (trigrams). Sorensen-Dice is a statistic used to gauge the similarity of two samples.

```
rules.alwaysTrue:
"Some string".sorensenDiceDistance("Another string") > 0.5 &&
"Some string".sorensenDiceDistance("Some other string") <= 0.35
```

String.sorensenDiceDistanceIgnoreCase(str)

Returns the Sørensen-Dice sørensenDiceDistance distance between the string and str, ignoring case.

```
rules.alwaysTrue:
"SOME STRING".sorensenDiceDistanceIgnoreCase("Another string") > 0.5 &&
"SOME STRING".sorensenDiceDistanceIgnoreCase("Some other string") <= 0.35
```

Passwords

String.hashPassword()

Returns the PBKDF2 hash of the string. Note that the result of this is different every time.

```
rules.alwaysTrue:
"mypassword".hashpassword() ==
"D7CV2UeshEN8C9Lw9U7GVw$q814S6B77dNj5DfhUeU10D4FQKR0ZkaVJw_cNN4oUz8"
```

String.checkPassword(arg)

Validates the supplied password against the hash argument.

```
rules.alwaysTrue:
"mypassword".checkpassword
("D7CV2UeshEN8C9Lw9U7GVw$q814S6B77dNj5DfhUeU10D4FQKR0ZkaVJw_cNN4oUz8") ==
true
```

Compression

String.compress()

Compresses the string using Java Deflator.

```
rules.alwaysTrue:
"long string [...]".compress() == "y8nPS1coLinKBFLRenp6sQA"
```

String.decompress()

Decompresses the string using Java Deflator.

```
rules.alwaysTrue:
"y8nPS1coLinKBFLRenp6sQA".decompress() == "long string [...]"
```

Number Methods

The following group of methods can be applied to numbers. If the method subject is not a number, expression evaluation will be stopped. Implementation of these methods mostly follows the Java Math library (see docs.oracle.com: Java Specifications > Math).

Number.abs()

Returns the absolute value of the number.

```
rules.alwaysTrue: -3.abs() == 3
```

Number.acos()

Returns the arc cosine of the value. The returned angle is in the range 0 - π.

```
rules.alwaysTrue: 1.acos() == 0
```

Number.asin()

Returns the arc sine of the value. The returned angle is in the range - π/2 - π /2.

```
rules.alwaysTrue: 0.asin() == 0
```

Number.atan()

Returns the arc tangent of a value. The returned angle is in the range -π /2 - π /2.

```
rules.alwaysTrue: 0.atan() == 0
```

Number.cbrt()

Returns the cube root of the number.

```
rules.alwaysTrue: 8.cbrt() == 2
```

Number.ceil()

Returns the smallest (closest to negative infinity) integer value that is greater than or equal to the number.

```
rules.alwaysTrue: 2.718.ceil() == 3
```

Number.cos()

Returns the trigonometric cosine of the value.

```
rules.alwaysTrue: 0.cos() == 1
```

Number.cosh()

Returns the hyperbolic cosine of the value.

```
rules.alwaysTrue: 0.cosh() == 1
```

Number.exp()

Returns Euler's number e raised to the power of the subject number.

```
rules.alwaysTrue: 0.exp() == 1
```

Number.expm1()

Returns Euler's number e raised to the power of the subject number minus one.

```
rules.alwaysTrue: 0.expm1() == 0
```

Number.floor()

Returns the largest (closest to positive infinity) integer value that is less than or equal to the number.

```
rules.alwaysTrue: 2.718.floor() == 2
```

Number.log()

Returns the natural logarithm of the number.

```
rules.alwaysTrue: 1.log() == 0
```

Number.log10()

Returns the logarithm (base 10) of the number.

```
rules.alwaysTrue: 100.log10() == 2
```

Number.max(n)

Returns the greater of the subject number and the argument `n`.

```
rules.alwaysTrue: 8.max(5) == 8
```

Number.min(n)

Returns the smaller of the subject number and the argument `n`.

```
rules.alwaysTrue: 8.min(5) == 5
```

Number.mod(n)

Returns the remainder when the subject number is divided by the argument `n`.

```
rules.alwaysTrue: 27.mod(5) == 2
```

Number.pow(n)

Raises the number to the power of `n`.

```
rules.alwaysTrue: 6.pow(2) == 36
```

Number.random()

IMPORTANT: Not recommended for use in production (see below).

Returns a random number greater than or equal to 0.0 and less than the subject number. The subject number must be positive.

```
rules.alwaysTrue: 5.random() < 5
```

Number.randomInt()

IMPORTANT: Not recommended for use in production (see below).

Returns a random integer greater than or equal to 0 and less than the subject number. The subject number must be positive.

```
rules.alwaysTrue: 5.randomint() < 5
```

IMPORTANT: Methods such as random() and randomint() that give non-deterministic results should only be used for testing, and not in production Business Rules. Non-deterministic methods can result in discrepancies between the real-time and asynchronous responses. This is because the output from the method is different when the event is processed synchronously to produce the real-time response, and when it is processed asynchronously by the fraud system.

Number.round(n)

Rounds the number to `n` decimal places. If no argument is specified, the method rounds to the nearest integer, with ties rounding up. If the number if already specified to less than n decimal places, the method returns the original number.

```
rules.alwaysTrue: 2.718.round() == 3
```

Number.signum()

Returns the signum function of the value, where:

$$\text{signum}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

```
rules.alwaysTrue: 6.signum() == 1
```

Number.sin()

Returns the trigonometric sine of the value.

```
rules.alwaysTrue: 0.sin() == 0
```

Number.sinh()

Returns the hyperbolic sine of the value.

```
rules.alwaysTrue: 0.sinh() == 0
```

Number.sqrt()

Returns the square root of the number.

```
rules.alwaysTrue: 25.sqrt() == 5
```

Number.tan()

Returns the trigonometric tangent of the value.

```
rules.alwaysTrue: 1.57.tan() > 1000
```

Number.tanh()

Returns the hyperbolic tangent of the value.

```
rules.alwaysTrue: 0.tanh() == 0
```

Number.toDegrees()

Converts the number from radians to degrees.

```
rules.alwaysTrue: 1.67.toDegrees() > 89
```

Number.toRadians()

Converts the number from degrees to radians.

```
rules.alwaysTrue: 89.toRadians() > 1.55
```

Collection Methods

AMDL supports some common collection methods as well as statistical methods:

- https://commons.apache.org: CollectionUtils
- https://commons.apache.org: SummaryStatistics
- https://commons.apache.org: DescriptiveStatistics

Note that statistical methods cause the expression execution to stop if the collection contains non-numerical values.

In addition, there are some methods which can only be applied to ordered collections (see Ordered Collection Methods).

Common Collection Methods

Collection.concat(another_collection)

Returns an ordered collection consisting of the original collection with all elements of another_collection appended.

```
rules.alwaysTrue: {"method1", "method2"}.concat({"method2",>
"method3"}) == ["method1", "method2", "method2", "method3"]
```

Collection.difference(another_collection)

Returns an unordered collection consisting of all elements in the first collection not present in another_collection.

```
rules.alwaysTrue: {"method1", "method2"}.difference({"method2", "method3"})
== {"method1"}
```

Collection.intersection(another_collection)

Returns an unordered collection consisting of all elements present in both collections.

```
rules.alwaysTrue: {"method1", "method2"}.intersection({"method2",
"method3"}) == {"method2"}
```

Collection.isEmpty()

Checks whether there are any values in the collection.

```
rules.alwaysTrue: [ "something", "here" ].isempty() == false
```

Collection.join(delimiter)

Concatenates all values in the collection into a string. If an argument is provided, it uses this as a delimiter. If no argument is provided, no delimiter is used.

```
rules.alwaysTrue:
[ "this", "is", "a", "collection" ].join (" ") == "this is a collection"
[ "this", "is", "a", "collection" ].join () == "thissacollection"
```

Collection.single()

Concatenates all values in the collection into a string. If an argument is provided, it uses this as a delimiter. If no argument is provided, no delimiter is used.

```
rules.alwaysTrue:
[ "this", "is", "a", "collection" ].join (" ") == "this is a collection"
[ "this", "is", "a", "collection" ].join () == "thissacollection"
```

Collection.size()

Returns the number of elements in the collection.

```
rules.alwaysTrue: [ "this", "is", "a", "collection" ].size() == 4
```

Collection.sorted( )

Returns an ordered collection in sorted order. If the values are strings, alphabetical sorting is used. If, however, the values are numbers, numerical sorting is used.

```
rules.alwaysTrue: [ 2, 1, 3 ].sorted() == [ 1, 2, 3 ]
```

Collection.symmetricDifference(another_collection)

Returns an unordered collection consisting of all elements present in the current collection and not in another_collection. The method also returns all elements present in another_collection and not the current collection.

```
rules.alwaysTrue: {"method1", "method2"}.symmetricDifference
({"method2", "method3"}) == {"method1", "method3"}
```

Collection.total()

Returns the sum of the values in the collection. If any non-numeric values are present in the collection, expression execution stops.

```
rules.alwaysTrue: [ 1, 3, 9 ].total() == 13
```

Collection.union(another_collection)

Returns an unordered set consisting of all the unique values in either collection.

```
rules.alwaysTrue: {"method1", "method2"}.union({"method2", "method3"}) ==
["method1", "method2", "method3"]
```

Statistical Methods

Collection.geometricMean()

Returns the geometric mean geometricMean of the values in the collection. If any non-number values are present or the collection is empty, the method returns null. Note that this does not return an exact amount for integer solutions.

```
rules.alwaysTrue: [ 1, 3, 9 ].geometricmean() > 2.9999
&& [ 1, 3, 9 ].geometricmean() < 3.0001
```

Collection.kurtosis()

Returns the kurtosis of the values in the collection.

```
rules.alwaysTrue: [ 1, 3, 9, 3, 1 ].kurtosis() > 3.25
&& [ 1, 3, 9, 3, 1 ].kurtosis() < 3.26
```

Collection.max()

Returns the maximum value max in the collection.

```
rules.alwaysTrue: [ 1, 3, 9 ].max() == 9
```

Collection.mean()

Returns the arithmetic mean of the values in the collection.

```
rules.alwaysTrue: [ 10, 20, 30 ].mean() == 20
```

Collection.median()

Returns the median of the values in the collection.

```
rules.alwaysTrue: [ 1, 2, 3, 4, 1000, 2000, 3000 ].median() == 4
```

Collection.min()

Returns the minimum value in the collection.

```
rules.alwaysTrue: [ 1, 3, 9 ].min() == 1
```

Collection.percentile(p)

Returns an estimate for the pth percentile of the values in the collection.

```
rules.alwaysTrue: [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 3 ].percentile(20) ==
1
```

Collection.populationVariance()

Returns the population variance populationVariance of the values in the collection.

```
rules.alwaysTrue: [ 1, 1, 4 ].populationvariance() == 1
```

Collection.quadraticMean()

Returns the quadratic mean quadraticmean (also known as root-mean-square) of the values in the collection.

```
rules.alwaysTrue: [ 1, 3, 9 ].quadraticmean() > 5
[ 1, 3, 9 ].quadraticmean() <6
```

Collection.secondMoment()

Returns the second central moment secondMoment of the values in the collection (the sum of squared deviations from the sample mean).

```
rules.alwaysTrue: [ 1, 3, 9 ].secondmoment() == 2
```

Collection.skewness()

Returns the skewness of the values in the collection, using the following definition:

$$\text{skewness} = \frac{n}{(n-1)(n-2)\sigma^3}\sum_{i=1}^{n}(x_i-\mu)^3$$

This definition consists of n as the number of elements, xi, as the collection, μ as the mean and σ as the standard deviation.

```
rules.alwaysTrue: [ 1, 3, 9, 3, 1 ].skewness() > 1.73
[ 1, 3, 9, 3, 1 ].skewness() < 1.74
```

Collection.stdDev()

Returns the standard deviation stddev of the values in the collection, based upon the sample variance.

```
rules.alwaysTrue: [ 3, 4, 5 ].stddev() == 1
```

Collection.sumOfLogs()

Returns the sum of the natural logs of the values in the collection.

```
rules.alwaysTrue: [ 2.718, 2.718, 2.718 ].sumoflogs() > 2.999
&& [ 2.718, 2.718, 2.718 ].sumoflogs() < 3.001
```

Collection.sumOfSquares()

Returns the sum of the squares of the values in the collection.

```
rules.alwaysTrue: [ 1, 3, 9 ].sumofsquares() == 91
```

Collection.variance()

Returns the (sample) variance of the values in the collection. This method returns the bias-corrected sample variance (using n - 1 in the denominator).

```
rules.alwaysTrue: [ 1, 1, 4 ].variance() == 3
```

Ordered Collection Methods

OrderedCollection.reverse()

Returns the collection in reversed order.

```
rules.alwaysTrue: [ "a", "c", "b", "d"].reverse() == [ "d",
"b", "c", "a" ]
```

OrderedCollection.shuffle()

Randomly permutes the collection using a default source of randomness.

```
rules.alwaysTrue: [[ "a", "b" ], [ "b", "a" ]] ~# [ "a", "b" ].shuffle()
```

OrderedCollection.sublist(a)

Returns the sublist of all elements with an index greater than or equal to the parameter `a`.

```
rules.alwaysTrue: [ 1, 2, 3, 4, 5 ].sublist(2) == [ 3, 4, 5 ]
```

OrderedCollection.sublist(a, b)

Returns the sublist of all elements with an index greater than or equal to the parameter `a` and less than the parameter `b`.

```
rules.alwaysTrue: [ 1, 2, 3, 4, 5 ].sublist(2, 4) == [ 3, 4 ]
```

Glossary

This page provides a list of glossary terms used in this guide.

A

Aggregator

An aggregator is a type of analytic that can combine and use the outputs of multiple rules and models to generate alerts.

Alert

The Fraud Transaction Monitoring System can flag up high-risk events for alert reviews. A flagged event is said to have generated an alert. The system's analytics rules, models and aggregators) can all generate alerts.

Alert Review

This is where analysts review alerts generated by the Thredd Fraud Transaction Monioring System. They can classify alerts as ‘Risk’ or ‘No Risk’, refer them to other users, or put them aside for further monitoring or to await additional information.

AMDL

AMDL (ARIC Modelling Data Language) is a language for specifying rules and logic within the Fraud Transaction Monitoring System. It is a declarative language for specifying state updates and executions on each event that passes through the system. An example of an event is an account registration or a transaction. Every event contains a reference (for example, an ID field) to one or more entities of different types, such as a merchant and a consumer. You can use AMDL to create Business Rules for the detection of fraud.

C

Chargeback

Where a cardholder disputes a transaction on their account and is unable to resolve directly with the merchant, they can raise a chargeback with their card issuer. The chargeback must be for a legitimate reason, such as goods and services not received, faulty goods, or a fraudulent transaction. For more information, see the Payments Dispute Management Guide.

E

Entity

Events happen to entities. An entity represents a unique individual or object, and every event is associated with at least one entity. For example, if a customer makes a card transaction, that event can be associated with the customer entity, the card entity, or both.

Entity ID

Each entity is identified by a unique entity ID in the event data for example, a 16-digit token.

Entity State

Every entity has a state – a combination of information about the entity that the system has accumulated over time. This is also called a behavioral profile. Every event processed by the system has the potential to update an entity’s state, adding more information or updating information that the system can use to build a behavioral profile of a customer or card for example.

Event

The Fraud Transaction Monitoring System recognizes potential fraud and financial crime by monitoring events. An event could be a customer transaction, a new customer application, or a merchant attempting to process a payment – these are all examples of event types. Each event is associated with one or more entities and one or more solutions.

I

Incident

In the Fraud Transaction Monitoring System, alerts are grouped into incidents. Each incident contains all the unreviewed alerts related to a particular entity.

Issuer (BIN sponsor)

The card issuer, typically a financial organisation authorised to issue cards. The issuer has a direct relationship with the relevant card scheme (payment network). For more information, see the Key Concepts Guide.

L

Label Events

Label events are types of event that contain ground truth information. They are used to label other events as 'risk' (i.e. confirmed fraud, financial crime, etc.) or 'no risk' (i.e. genuine). Alert reviews are one common form of label event, but your portal may also use other kinds of label event, such as chargebacks or manual fraud reports. Labels are used by Adaptive Behavioral Analytics models to learn to better identify high-risk events. They are also used to quantify and report on the performance of models.

M

Mastercard Fraud and Loss Database

A Mastercard repository of fraud transactions submitted by issuers. It is used for reporting, monitoring, and combating card fraud. Previously know as: System to Avoid Fraud Effectively (SAFE).

MasterCom API

MasterCom API offers Mastercard customers the ability to create and manage dispute claims in MasterCom. MasterCom is a system for dispute management. All activities for any given dispute can be tracked within a single claim using Mastercom, including Retrieval Request and Fulfilment, First Chargeback, Second Presentment, Fraud reporting, Case Filing, and Fee Collection requests. All activities for any given dispute throughout its lifecycle can be tracked within a single claim.

Model

A model in the Thredd Fraud Transaction Monitoring System is a predictive model that processes events and generates a risk score for certain event types, for example, authorisations.

P

PAN

The Primary Account Number (PAN) is the card identifier found on payment cards, such as credit/debit/prepaid cards, as well as stored-value cards, gift cards and other similar cards. The card's 16-digit PAN is typically embossed on a physical card. For more information, see the Key Concepts Guide.

R

Real time/near-real time events

Every event is processed by analytics in the Featurespace fraud monitoring system engine. This processing happens in strict chronological order, so that no event is ever processed out of sequence. This is asynchronous processing, and happens to all events. However, some events, such as authorisations, require a real-time response (within a few hundredths of a second). These must be processed in a way that prioritizes low latency (such as a fast response), rather than chronological order. This kind of event is called a real-time event, and is processed by the portal synchronous response generator (as well as the portal Engine). Events that do not require a real-time response (asynchronous events), are only processed by the engine, for example, chargebacks, address or phone number updates

Rule

A rule defines some simple logic – rules take in information from events, entity states, and other data, and output a simple true/false response. Rules are written in the business logic definition language, AMDL.

Rule Set

Each Analytical Workflow is divided into a series of Rule Sets. Each Rule Set contains a number of expressions written in AMDL, and one or more Scorecards which contain conditions that determine what effects the Workflow triggers (e.g. generating an alert, adding a tag, outputting a risk score). Each Rule Set may also have a condition that determines whether or not that Rule Set is executed for an event.

S

Single Sign-On (SSO)

An identification method that enables users to log in to multiple applications and websites with one set of credentials.

Smart Client

Smart Client is Thredd's user interface for managing your account on the Thredd system. Smart Client is installed as a desktop application and requires a VPN connection to Thredd systems in order to be able to access your account. For more information, see the Smart Client Guide.

Solution

Multiple product Solutions may be configured in your portal deployment. Each Solution provides a combination of UI configurations, data enrichment and analytics for detecting a specific type of risk. For example, you may have a Solution for application fraud and another for inbound/outbound payments, subject to your programs set up with Thredd and Featurespace. The same event may trigger separate alerts in different Solutions.

Solution ID

Each Solution is uniquely identified by a Solution ID in the event data.

Solution UI

The fraud system user interface that users access when they open the relevant Solution. The Solution UI is mainly used for reviewing incidents that are specific to that Solution, and can be customized for detecting the relevant type of financial risk.

State

Every entity has a state – a combination of information about the entity that the systems has accumulated over time. This is also called a behavioral profile. Every event processed by the system has the potential to update an entity's state, adding more information or updating information that the system can use to build a behavioral profile of a customer or card for example.

T

Tag

Rules, aggregators and models can add tags to alerts, to give analysts more information or to automate a response in a downstream system, such as declining a transaction.

Token

Displays the unique token linked to the card PAN on which the transaction was made.

Document History

Version	Date	Description	Revised by
1.0	10/05/2024	Added details on the var and rules scope for rule scores using the @score annotation.	KD
	26/04/2024	Updates to content and graphics to align with taxonomy updates on our Documentation Portal.	KD
	31/01/2024	First version	KD

Contact Us

Please contact us if you have queries relating to this document. Our contact details are provided below.

Thredd Ltd.

Support Email: occ@thredd.com

Support Phone: +44 (0) 203 740 9682

Our Head Office

6th Floor,

Victoria House,

Bloomsbury Square,

London,

WC1B 4DA

Telephone: +44 (0)330 088 8761

Technical Publications

If you want to contact our technical publications team directly, for queries or feedback related to this guide, you can email us at: docs@thredd.com.