Developing Apama Applications in Event Modeler

5.2.0

August 2014

This document applies to Apama 5.2.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: PAM-Developing_Apama_Applications_in_Event_Modeler-5.2.0-20140808@233876

Table of Contents

Preface **8**
About this documentation 8
How this book is organized 8
Documentation roadmap 9
Contacting customer support 11

Chapter 1: Overview of Using Event Modeler **12**
Event Modeler layout 12
About event flow states 14
How rules define scenario behavior 16
Description of rule conditions 17
Description of rule actions 19
Description of functions in rules 20
About rule evaluation 20
Basic view of rule processing 21
Expanded view of rule processing 22
Scenario monitoring stage 25
Summary of adding rules when a variable value changes 25
About scenario variables 25
Variable types 26
Auto-typing of variables 26
Variable properties 26
Variable constraints 27
User input and output 28
About blocks 28
Linking variables, block parameters, and block output fields 29

Chapter 2: Using Event Modeler **31**
Adding scenarios to projects 31
Creating the GlobalRuleExample project 32
Adding GlobalRuleExample.sdf to the GlobalRuleExample project 32
Adding a new scenario to the GlobalRuleExample project 32
Opening and viewing multiple scenarios 33
Selecting from the Scenario menu 33
The Event Modeler toolbar 34
Interacting with the tabs and panels 35
Working in the Event Flow panel 36
Interacting with states 36
Selecting a state 37
Resizing a state 37
Moving a state 37
Multiple selection 37
Adding a state 38
The finished status 38

Deleting a state....38
Labeling a state....38
Using cut/copy/paste with states....39
Interacting with transitions....39
Adding a transition....39
Selecting a transition....40
Changing end-points....40
Changing the shape of a transition....40
Labeling a transition....41
Deleting a transition....42
Using cut/copy/paste with transitions....42
Displaying global rule transitions....42
Working in the Rules panel....43
Adding a rule....43
About global rules....43
Selecting rules and rule elements....44
Re-ordering rules....44
Deleting a rule....44
Labeling a rule....45
Changing a rule's description....45
Minimizing and maximizing a rule....45
Cutting, copying, and pasting rules....46
Activating and deactivating rules....46
Specifying conditions....46
Interactive editing....47
Language elements....47
Selecting and replacing elements....48
Cascading alternative menus....50
Using functions in rules....51
Adding a condition to a rule....51
Specifying variable changes in conditions....52
Local rules and variable changes....54
Global rules and variable changes....54
Specifying actions....55
Adding action statements....55
Deleting action statements....56
Interactive editing....56
Using the keyboard to edit rules....56
Using the Variables tab....58
Adding a variable....59
Renaming a variable....60
Selecting a variable....60
Determining which states use a particular variable....60
Moving a variable....60
Deleting a variable....61
Changing a variable's properties....61
Setting a variable's value....62
Variable input and output....63

Linking a variable to a block output field....63
Conversion rules for variable types....65
Using the Catalogs tab....66
Adding a block template catalog....66
Selecting and inspecting a block template....67
Adding a block instance to the scenario....67
Using the Functions tab....68
Adding a function catalog....68
Selecting and inspecting a function....69
Using the Blocks tab....69
Interacting with a block instance....71
Selecting a parameter....71
Viewing a parameter's properties....72
Setting a parameter's initial value....72
Linking a parameter with a variable or output field....72
Switching blocks....73
Using the Block Wiring tab....74
Wiring block input feeds....74
Selecting, resizing, and moving block instances....75
Wiring two blocks together....75
Connecting feeds and specifying feed mapping....76
Wiring a scenario variable to a block....77
Mapping type conversions....77
Editing block wiring....78
Deleting a wiring....78
Deleting a block instance....78
Using older versions of blocks....78
Troubleshooting invalid scenarios....79
Setting preferences....79
Exporting scenarios as EPL....81
Exporting scenarios as block templates....81
Event Modeler command line options....81

Chapter 3: Working with Blocks Created from Scenarios....84
Terminology for using scenario blocks....85
Benefits of scenario blocks....86
Steps for using scenario blocks....86
Background for using scenario blocks....86
Saving scenarios as block templates....87
Incrementing scenario block version numbers....87
Adding a scenario block to a main scenario....88
Examining a scenario block's source scenario....88
Descriptions of scenario block parameters....88
Descriptions of scenario block operations....89
Descriptions of scenario block feeds....90
Setting parameters before creating sub-scenarios....93
Creating sub-scenarios....94
Deleting sub-scenarios....95

Unconditionally deleting a sub-scenario....96
Deleting all sub-scenarios....96
Modifying sub-scenario input variable values....96
Iterating through sub-scenarios....96
Obtaining variable values from sub-scenarios....98
Linking sub-scenarios with other blocks....98
Inheriting sub-scenarios....98
Description of inheritExternalInstances values....98
Notes for setting the inheritExternalInstances parameter....99
Example of inheriting sub-scenarios....99
Observing changes in sub-scenarios....100
Performing simple calculations across sub-scenarios....102

Chapter 4: Using Functions in Event Modeler....104
Reference information for provided functions....104
Date and time functions....104
Extended math functions on float types....106
IO functions....108
System value functions....109
Miscellaneous functions....110
Extended math functions on float types....114
About defining your own functions....116
Sample ABS function definition file....116
Sample function definition file with imports element....117
About function names....118

Chapter 5: Using Standard Blocks....120
A block's lifecycle....121
General analytic blocks....122
Change Notifier v2.0....122
Correlation Calculator v2.0....124
Data Distribution Calculator v2.0....125
Median and Mode Calculator v1.0....127
Moving Average v1.0....128
Spread Calculator v3.0....129
Statistics Calculator v1.0....130
Velocity Calculator v2.0....132
The Timer blocks....133
Schedule v3.0....133
Wait v3.0....136
The Utility blocks....137
Dictionary v2.0....137
File Reader v2.0....138
File Writer v2.0....140
History Logger v2.0....141
Input Merger v2.0....143
List v2.0....144
Scenario Terminator v2.0....145
Status v2.0....146

Variable Mapper v2.0 150
Database functionality—storage and retrieval 151
ADBC Storage v1.0 151
ADBC Retrieval v1.0 154
Blocks for working with scenario blocks 159
Change Observer v2.0 159
Filtered Summary v2.0 161

Preface

- About this documentation 8
- How this book is organized 8
- Documentation roadmap 9
- Contacting customer support 11

About this documentation

Developing Apama Application in Event Modeler provides information and instructions for defining independent, real-time, business strategies, referred to as scenarios. Each scenario can contain any number of states, and transitions between states happen according to rules that you define.

You use the Event Modeler to create scenarios. You inject completed scenarios into the correlator, and then use a dashboard to create and configure one or more instances of the scenario. Each scenario instance listens for particular events or sequences of events. When the scenario instance finds events or sequences of interest, it performs specified actions according to the rules defined in the scenario.

After you develop a scenario in Event Modeler, you use Dashboard Builder to create a graphical dashboard for the scenario. The dashboard lets end users create and interact with scenario instances through an intuitive and easy to manipulate graphical user interface, which is described in *Building Dashboards*.

It is assumed that you have read *Introduction to Apama*, which introduces scenario concepts, discusses the scenario development lifecycle, and covers Apama® architecture and other Apama concepts.

Preface

How this book is organized

The information in this book is organized as follows:

- "Overview of Using Event Modeler" on page 12 uses an example to show you the procedure for developing a scenario.
- "Using Event Modeler" on page 31 provides details about defining scenario states, rules, blocks, and variables.
- "Working with Blocks Created from Scenarios" on page 84 shows how to save a scenario as a Block and then use that scenario Block in some other scenario.
- "Using Functions in Event Modeler" on page 104describes the standard functions you can use in a scenario and provides instructions for defining your own function.
- "Using Standard Blocks" on page 120 provides details for using all Blocks provided with Apama.

Preface

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.2 > Apama Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write JMon programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.

Title	Contents
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: • Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Preface

Chapter 1: Overview of Using Event Modeler

- Event Modeler layout 12
- About event flow states 14
- How rules define scenario behavior 16
- Basic view of rule processing 21
- About scenario variables 25
- About blocks 28
- Linking variables, block parameters, and block output fields 29

This chapter introduces the concepts underlying the layout and functionality of the Apama Studio Event Modeler. It does not attempt to describe how to use the tool or how to interact with its various tabs and panels. That explanation is provided in "Using Event Modeler" on page 31, once the underlying concepts are understood.

Before using *Developing Apama Applications in Event Modeler*, we recommend that you take advantage of the Apama Studio Tutorials numbered 7, 8, and 9. These tutorials let you quickly start using Event Modeler by adding to a partially formed scenario. To access the tutorials, open Apama Studio and click Tutorials on the Welcome page.

It is assumed that you have read *Description of Event Modeler* and *Understanding scenarios and blocks* in *Introduction to Apama*, which introduces scenario concepts and discusses the scenario development lifecycle.

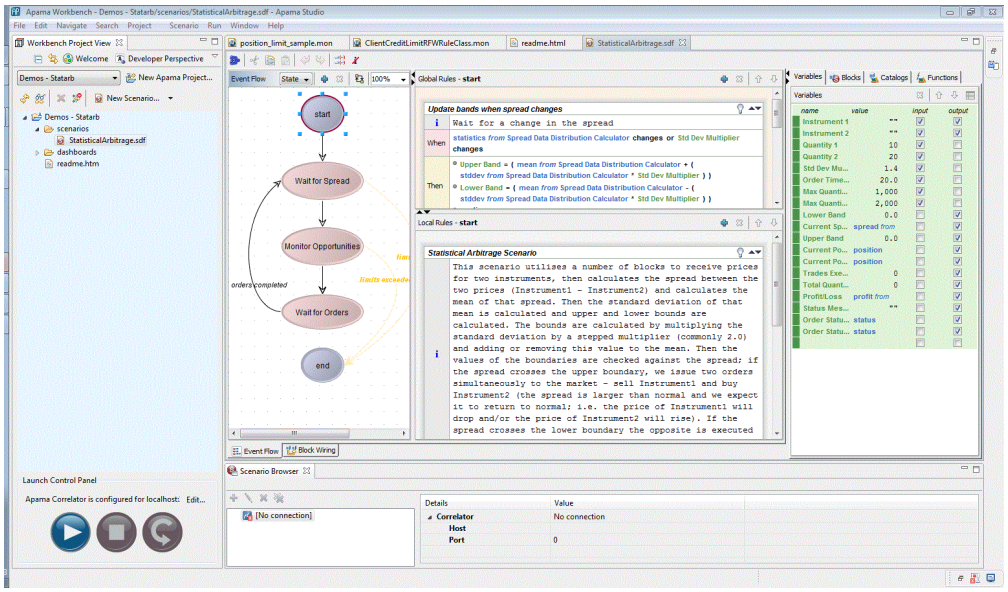
Event Modeler layout

To begin learning how to use Event Modeler, it is helpful to examine a sample scenario in Event Modeler. To do this:

1. Select Start > Programs > Software AG > Apama 5.2 > Apama Studio .
2. On the Apama Studio Welcome page, click Samples, then click Apama Samples.
3. In the list of demos that appears, click Statistical Arbitrage and click Open to open the demo application's project in Apama Studio.
4. In the Apama Studio Workbench Project View, expand Scenarios and double-click StatisticalArbitrage.sdf. This is the scenario definition file. When you double-click it, it opens in Apama Studio's Event Modeler editor.

The Event Modeler editor is divided into a number of areas. In the panel on the left (the Event Flow tab) click on the double-bordered oval shape marked start. Your display will now look as follows: xxx

Figure 1. Event Modeler editor layout



This is the default view. Event Modeler displays the following primary areas:

- Event Flow
- Global Rules and Local Rules
- Tabs for Variables, Blocks, Catalogs, and Functions.

At the bottom of Event Modeler, there are tabs for Event Flow and Block Wiring. When you click the Block Wiring tab, the Event Flow and Rules panels diappear and the Block Wiring tab appears.

During its lifetime, a scenario instance transits through a number of execution states, starting from the start state, and eventually ending at the end state (shown in the Event Flow tab). Event flows are described in "About event flow states" on page 14.

Each state consists of a list of rules that are executed in a particular sequence. Each has a condition that needs to be met for its embedded actions to be executed, and once those actions are complete, it can specify whether the following rules are to be processed next or the scenario should transit directly to another state. These rules appear in the Global Rules and Local Rules panels. Rules are examined in "How rules define scenario behavior" on page 16

The Variables tab lists any variables defined in the scenario. Scenario variables are placeholders for important information that needs to be referred to and modified during the scenario's execution. They also reflect what data can be collected from the user or sent back to be displayed to the user as results or progress updates. Variables will be described in "About scenario variables" on page 25.

The Blocks tab lists any blocks that are being used by this scenario. Blocks are pre-packaged modules that can be imported and used within scenarios. They can accept inputs, execute some logic of their own, and generate output. Like a scenario, blocks can themselves have configuration parameters as well as input and output feeds. Blocks can also carry out specialized operations. See "About blocks" on page 28 for details.

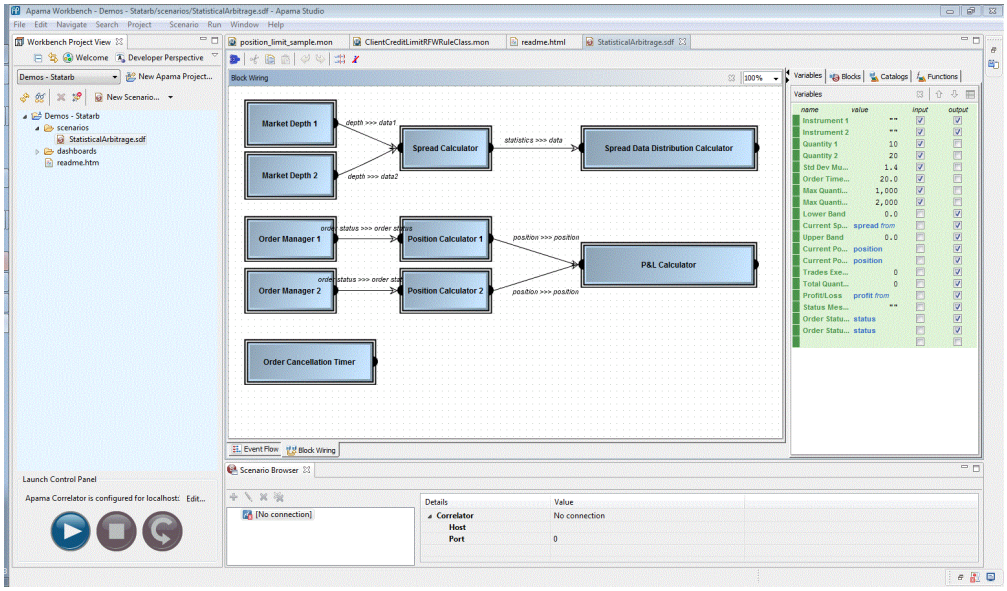
The Catalogs tab lists the reusable, ready packaged blocks that are available for use in this scenario. Event Modeler comes with a selection of standard blocks, and these are documented in "Using

Standard Blocks" on page 120. "Using the Catalogs tab" on page 66 describes usage of the Catalogs tab.

The Functions tab lists the functions that are available for use in this scenario. Event Modeler comes with a selection of standard functions, and these are documented in "Using the Functions tab" on page 68.

Minimize the panels that are not part of Event Modeler and then click the Block Wiring tab that appears below the Event Flow tab. The main view changes to show the Block Wiring tab. The Event Modeler display now looks like this:

Figure 2. Block Wiring tab



This tab shows the blocks that are being used within this scenario, and whether those blocks are wired together; that is, whether the outputs of one block are acting as the inputs of another. This functionality will be described in "Switching blocks" on page 73. The specific functionality of all the tabs will be covered in depth in "Using Event Modeler" on page 31.

Overview of Using Event Modeler

About event flow states

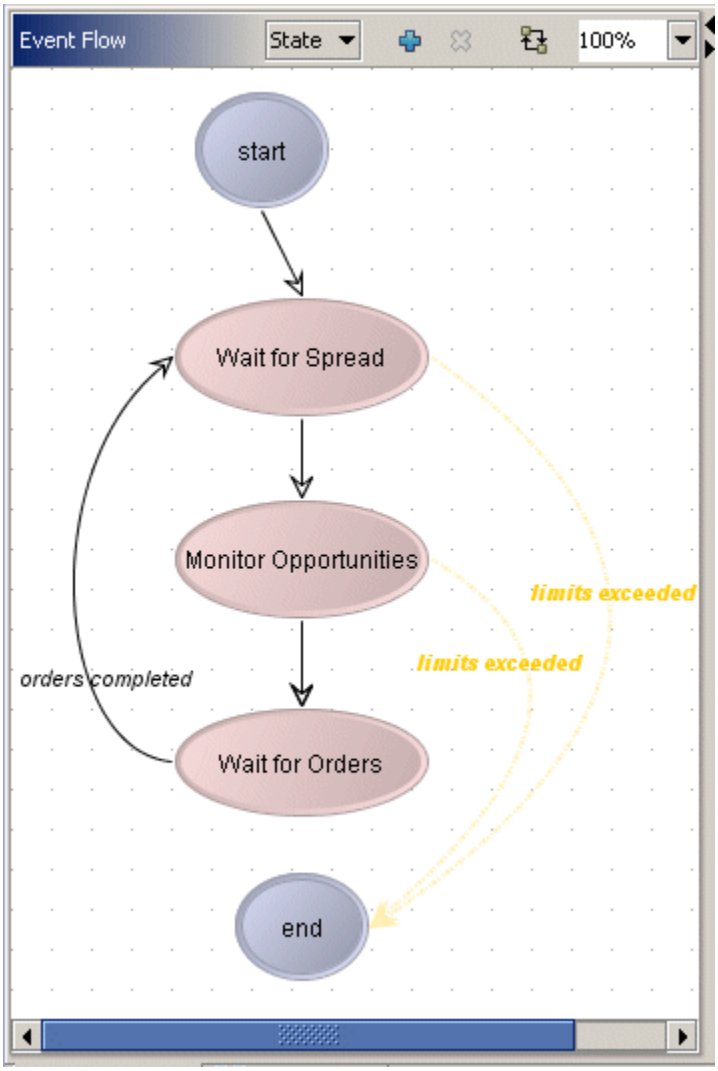
At any moment in a scenario instance's execution, it is said to be in a particular *state* in the event flow. The activities and actions that a scenario instance will be doing at any moment depend on its state, and are defined by that state's *rules*.

The execution of a scenario instance consists of progressing through a sequence of states, starting from the start state, and ending at the end state. For this reason, all scenarios must have a start and an end state.

A scenario instance can only ever be in one state, but there might be a choice of states it can advance to from that state. It is also possible for a scenario instance to move from a state back to the same state again. A scenario instance will continue executing until it reaches the end state, then it will terminate.

The Event Flow tab illustrates all the possible states that the scenario instance can be in while it is running inside the correlator. Note how when the Statistical Arbitrage sample is loaded the Event Flow tab is showing the following states (the arrows indicate possible transitions between states):

Figure 3. Event Flow tab



Using this scenario as an example, when the Statistical Arbitrage scenario is deployed to the correlator, it will start execution from the start state. From this state it can only transit to the Wait for Spread state. In Wait for Spread, however, it can go directly to the end state and terminate its execution (by means of a global rule - shown as an orange line; more on this later), or else transit to the Monitor Opportunities state by means of a local rule.

From the Monitor Opportunities state, the scenario can advance to the Wait for Orders state, or it can terminate execution and go to the end state. If execution does reach the Wait for Orders state, it can only transit back to the Wait for Spread state. What causes a scenario instance to change from one state to another state, and what it does while it is in a state, depends on its *rules*.

Overview of Using Event Modeler

How rules define scenario behavior

States matter because of the distinct behavior that the scenario instance will follow while in a particular state. And that is defined in each state's set of *rules*. A state can have one or more rules defined in it. Each rule has the following structure; "*if a condition is true then do the following …*".

The center panel has two parts — Global Rules and Local Rules. A global rule can apply to more than one state. A local rule can apply to exactly one state. When you select a state in the Event Flow tab, the rules defined for that state appear in the Rules panel.

Each rule has a *condition* part, denoted by When, and an *action* part, indicated by Then. The part indicated by the i symbol is just a descriptive comment that you can set to whatever you like. You can hide or show the comment by selecting in the Event Modeler toolbar. The start state illustrated in the previous topic has two local rules, including this rule:

Figure 4. Sample local rule

This is stating:

- *when* true, which means: always do this,
- *then* do the following:
  - carry out the `start` operation on the following block instances
    - `Market Depth 1`
    - `Market Depth 2`
    - `Spread Calculator`

    - `Spread Data Distribution Calculator`
    - `Position Calculator 1`
    - `Position Calculator 2`
    - `P&L Calculator`
- set the `StatusMessage` variable to "`Waiting for price data`"
- `continue`, that is, evaluate the next local rule

Variables: For now it is enough to know that, as in other programming environments, scenario variables are placeholders for useful information that the scenario needs to keep track of and perhaps modify during its execution. They also identify the information that will be required by a running instance of the scenario from the end-user in order to configure and start it off, as well as representing the information that will be sent back to be displayed to the user as progress updates or results.

- Variables are typed; each can be of type `text`, `number`, `choice` or `true/false`.
- Variables are described in "About scenario variables" on page 25.

Blocks: Likewise, blocks are ready-packaged modules that you can import and use within your scenarios. They can accept inputs, execute some logic of their own, and generate output. A block can consist of *Input feeds* (which contain one or more *input fields*), *Output feeds* (which contain one or more *output fields*), *Parameters*, and *Operations*. Block parameters and fields are typed; each can be of type `text`, `number`, `choice` or `true/false`. Blocks are described in "About blocks" on page 28.

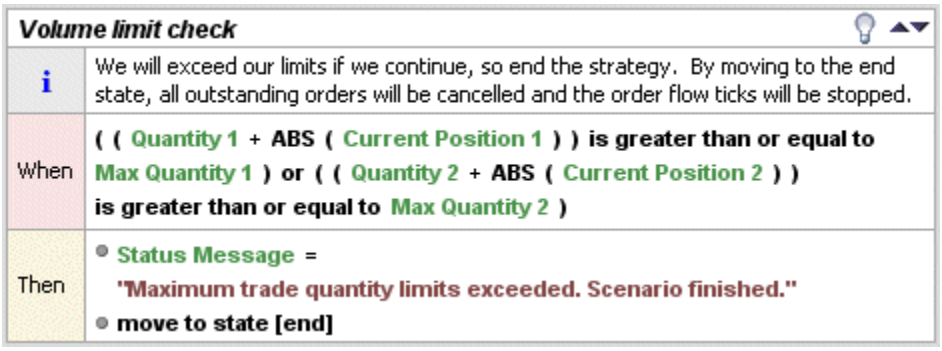
In addition to the standard blocks provided with Event Modeler, you can build custom blocks in Apama Studio.

Overview of Using Event Modeler

Description of rule conditions

The condition specified in a rule must be true for the action part to be executed. Conditions can be as straightforward as the example seen so far, such as a condition that specifies just `true (evaluated once)`. This condition causes the action part to execute whenever the rule is evaluated. However, more often a condition will specify a constraint on the value of a variable, field or parameter, for example, "*is a particular variable at present greater than this value*". It can also be a complex composition of various conditions defined using the operators `and` and `or`. For example:

1. Click on the Wait for Spread state.
2. In the global rules pane, scroll down to the last global rule, the one labeled `Volume limit check`.



Consider the condition for this first rule. This condition will be true if:

```
( Quantity 1 + ABS(Current Position 1) ) is greater than or equal to Max
Quantity 1
or
( Quantity 2 + ABS(Current Position 2) ) is greater than or equal to Max
Quantity 2
```

This condition contains two clauses:

- Whether the result of the variable `Quantity 1` being added to the absolute value of `Current Position 1` is greater than or equal to the variable `Max Quantity 1`.
- Whether the result of the variable `Quantity 2` being added to the absolute value of `Current Position 2` is greater than or equal to the variable `Max Quantity 2` .

As the two clauses are joined with an `or`, only one needs to be true for the condition to be true as a whole. Had the operator used been an `and`, then both of the clauses would have needed to be true for the condition as a whole to evaluate to true.

A condition needs to evaluate to the value `true` or `false`. Apart from the literal values `true` and `false` themselves, a condition can also consist of any of the following:

- The inverse of any other condition. This can be achieved by expressing `not` before that condition
- A variable (or block parameter or block output field) that is of type `True/False` (or *condition*)
- A check on whether a variable's value (or block parameter or block output field) has changed since the beginning of this state or since it was last checked by this rule

  For example, `Max Quantity changes`
- A function call whose result is either true or false

  For example, `isWeekday("Friday")`
- Any *numeric* expression being compared with another numeric expression. A numeric expression equates to a numeric value, and can be arrived at by any combination of arithmetic operations, functions and/or number variables. Numeric expressions can be compared to each other with `is less than`, `is less than or equal to`, `is greater than`, `is greater than or equal to`, `is equal to`, and `is not equal to`.

  For example, `Price is less than 20`

  or

  ```
  ((Price * 2) / Quantity) is greater than POW(Upper Limit, 5)
  ```
- Any *text* expression being compared with another text expression. A text expression is a string (that is, a word or phrase) and can be arrived at by any number of operations, functions and/or

text variables. Text expressions can be compared to each other with `is equal to`, `is not equal to`, and `contains`.

For example, `Name is equal to "Tom"`

or

`"Bookmark" contains "book"`

- Any choice variable being compared with a valid choice value. The latter can be another choice variable or a text expression. A choice variable is one whose valid values are limited to a particular selection of text values. The valid comparisons here are `is equal to`, and `is not equal to`.
- Any number of nested conditions joined with `and` or `or`

  For example, `Max Quantity changes and ( Price is less than 15 or Price is greater than or equal to 20 )`

Details on how to specify conditions in the Rules panel are given in "Working in the Rules panel" on page 43.

How rules define scenario behavior

Description of rule actions

If a condition evaluates to true, then the corresponding action part of that rule will be processed.

Actions consist of a number of *action statements*, and a *state transition statement*. The former are optional; it is possible to have an action that does not have any action statements. However, there must always be a state transition statement.

The state transition statement is straightforward; it will either be `continue`, or else `move to state [`one of the scenario's states`]`. It is important to note that the latter format could indicate a transition back to the same state, and that this is in fact different to stating `continue`. The distinction will be explained in "About rule evaluation" on page 20.

An action statement can be:

- Assign the value of a numeric expression (that is, a number) to a numeric variable or block parameter. For example:

  ```
  Trades Executed = Trades Executed + 2
  ```

- Assign the value of a text expression (that is, a word or phrase) to a text variable or block parameter. For example:

  ```
  Status Message = "Both orders filled"
  ```

- Assign the value of a condition (true or false) to a conditional variable or block parameter. What constitutes a valid condition here is the same as listed in "Description of rule conditions" on page 17. For example:

  ```
  Active = ((Price * 2) / Quantity) is greater than POW(Upper Limit, 5)
  ```

- Assign the value of a text expression or choice variable to a choice variable or block parameter
- Invoke a block operation

See "Working in the Rules panel" on page 43 for details about specifying rules.

How rules define scenario behavior

Description of functions in rules

As you might have noticed from some of the examples used so far, functions are available in both conditions and actions.

Functions in Event Modeler take a fixed set of parameters, with each parameter being of a particular type. A function will return a single value of a particular type. The types available for both parameters and results are `text`, `number` and `True/False` (or condition).

Functions are each defined in a *function definition file* or `.fdf` file.

The bundled functions include commonly used arithmetic and string functions, like *abs* (the absolute value of a number), *ceil* (the whole number ceiling of a number), *floor* (the whole number floor of a number), *pow* (to the power of) and *concat* (concatenate). These functions are documented in "Using Functions in Event Modeler" on page 104.

Note that any `.fdf` files located in the folder `functions` are automatically picked up by the Event Modeler at startup time, and made available when defining rules.

How rules define scenario behavior

About rule evaluation

When scenario execution enters a state, the rules of that state are examined in the order they are defined. If there are global rules as well as local rules, Event Modeler evaluates the first global rule first.

The first rule's condition is checked to verify whether it is true or false.

If the condition is false, then execution moves on to the next rule, and the procedure is repeated in the same way for that rule. If there are global rules, the next rule is the next global rule. If there are no more global rules, the next rule is the first local rule. If Event Modeler processes all rules assigned to a state, the order is top to bottom in the combined Global and Local Rules panel.

If, on the other hand, the rule's condition is true, then its action part is processed. The action statements are executed, and then the state transition statement is examined. If it is `continue`, then execution moves on to the next rule. If, on the other hand it is `move to state [some state]` then the scenario will proceed directly to that state and ignore all other rules. Their conditions will not be reviewed and their action parts never processed. In the new state, the same procedure highlighted here is followed.

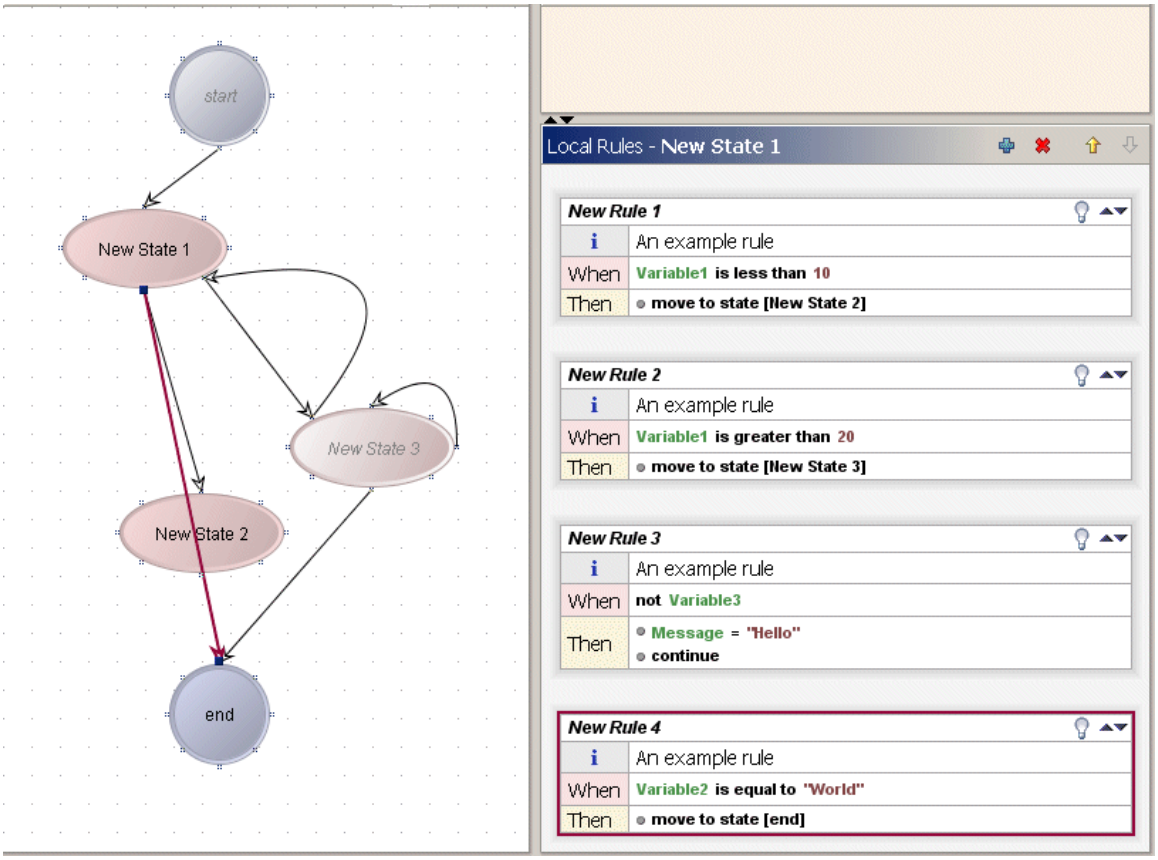
Note that as stated previously `continue`, and `move to state [this same state]` are different. The former causes execution to proceed to the next rule, while the latter causes the state's execution to restart from the first rule as if we had entered this state from a completely different state.

How rules define scenario behavior

Basic view of rule processing

Consider the set of rules shown in this screen.

Figure 5. Rule processing example one



This example scenario has four variables, called `Variable1` (number), `Variable2` (text), `Variable3` (condition) and `Message` (text). These variables could have any value when execution enters `NewState 1`. Their initial values would normally have been set by a user on creation of the scenario instance, or else they could have been set and modified by some rule in the `start` state.

Consider `NewState 1`, which specifies four rules. When the scenario instance's execution first enters `NewState1`, its rules will be processed as follows:

1. `New Rule 1` will be examined first.
2. If the value of `Variable1` is less than `10` then its condition will be true, its action part will be processed, and this will move the scenario's execution to `NewState 2` right away. `New Rule 2`, `New Rule 3` and `New Rule 4` will be ignored, and the rest of the steps outlined here would not apply.
3. If the value of `Variable1`, however, was greater than or equal to `10`, then the condition of `New Rule 1` will be false. In this case, `New Rule 2` will be examined.
4. In `New Rule 2`, if `Variable1` was actually greater than `20`, then the action part of `New Rule 2` gets processed, and this time the scenario moves to `NewState 3`. `New Rule 3` and `New Rule 4` will be ignored. No further steps apply.

5. On the other hand, in `New Rule 2`, if the condition was false, we move to `New Rule 3`.
6. In `New Rule 3`, if the value of `Variable3` was false, then `not Variable3` would be true, and the condition of `New Rule 3` would be true. In this case, `Message` would get set to the text "`Hello`". Since the state transition statement is `continue`, then `New Rule 4` will be processed.
7. Had the condition of `New Rule 3` been false, `Message` would not get set to the text "`Hello`". However, `New Rule4` would have been processed anyway.
8. The condition of `New Rule 4` checks whether `Variable2` contains the text "`World`". If so, execution proceeds to the `end` state. If not, then all rules would have been processed and the scenario would go into a monitoring stage. This will be described later.

This illustrates the way in which rules are processed, in order, from top to bottom.

Overview of Using Event Modeler

Expanded view of rule processing

While the previous top-to-bottom rule processing occurs in the majority of scenarios, the full picture of how rules are processed is more elaborate.

In practice, when execution enters a state, the rules of that state are placed on a queue in the order shown in the `Rules` panel — first global rules and then local rules. This queue is known as the *rule queue*. Rules are taken off the head of this queue and processed.

The sequence can differ if any of the action statements modify a scenario variable (or block parameter or block field) that is referenced by the condition of *any rule within that state*.

In that case, **all rules whose condition references that variable, and that are no longer on the queue, will be added to the end of the queue**. If those rules had already been on the queue waiting to be processed, then they would not be added again. For example, consider the following rules:

- R1: f(b): continue;
- R2: f(a): continue;
- R3: f(c): a=7; continue;
- R4: f(d): b=0; continue;
- R5: f(a,b): continue;

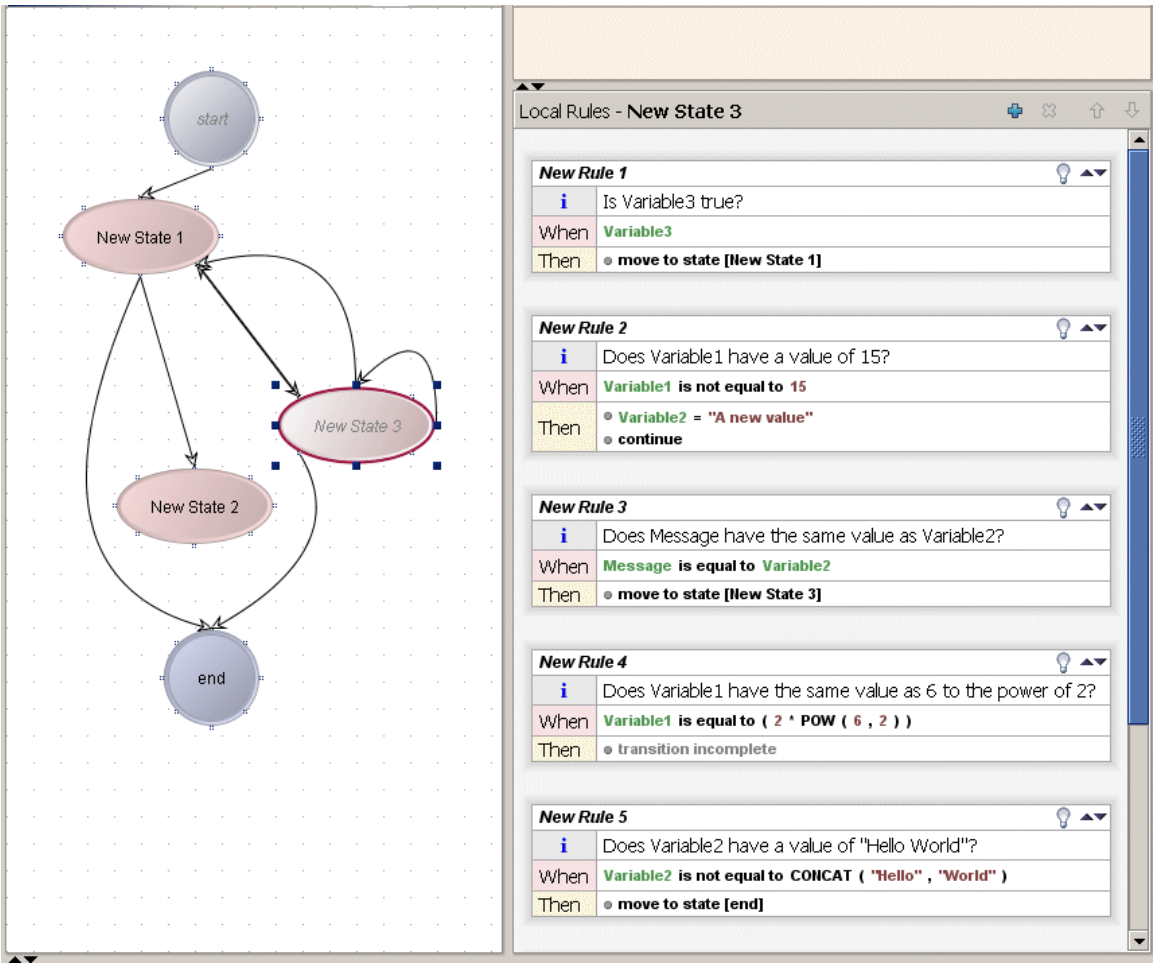
Suppose that a, b, c, and d are variables and f(a) means "some function of 'a'". Assume that f(c) and f(d) are both true. Event Modeler places the rules on the queue as follows:

R1 R2 R3 R4 R5 R2 R1

As you can see, when Event Modeler adds a rule to the queue, it always adds it to the end of the queue.

Consider the set of rules shown in the next screen:

Figure 6. Rule processing example two



1. When execution enters `NewState 3` any rules of the previous state are removed from the rule queue, and the following rules will be placed on it, in this order: `New Rule 1`, `New Rule 2`, `New Rule 3`, `New Rule 4` and `New Rule 5`.

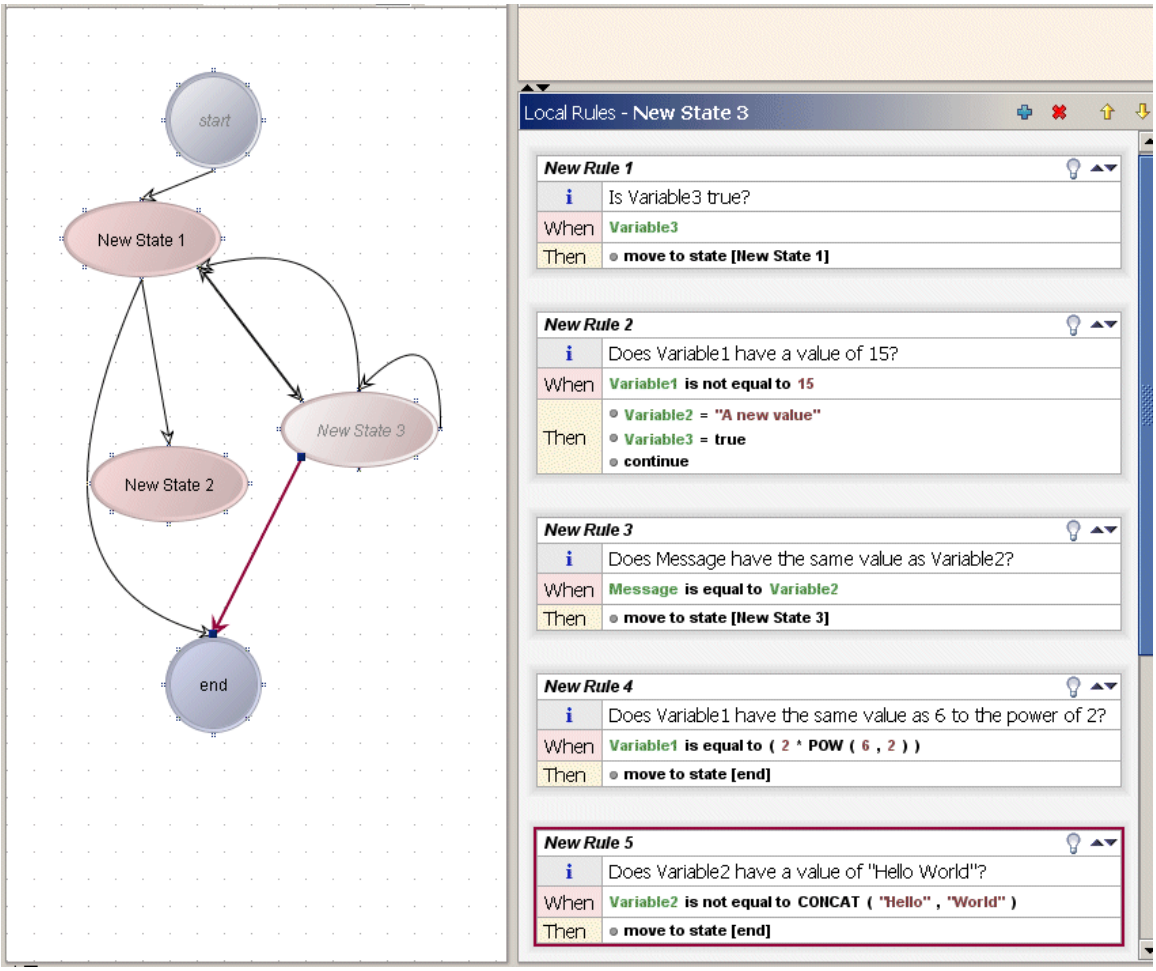
2. `New Rule 1` will be taken off the queue and its condition examined. If `Variable3` is true, then the scenario will move to `NewState1`. The rule queue will be emptied of all `New State 3` rules, and no further steps apply.

3. However, if `Variable3` is false, then `New Rule 2` is taken off the queue and its condition checked. Note that at this point the rule queue would contain `New Rule 3`, `New Rule 4` and `New Rule 5`. `New Rule 2's` condition states that if `Variable1` is not equal to `15` its action part must be processed. Let us assume that `Variable1` is indeed not equal to `15` and its single action statement changes `Variable2` to the value "`A new value`".

4. What happens next in this case depends on the state transition statement of `New Rule 2`. If it had caused a transition to another state, then the scenario would have emptied the rule queue, moved to that state, and then repopulated the queue with the rules from the new state. However, in this case the state transition statement is `continue`. Note that `Variable2` is referred to in the condition part of `New Rule 3` and `New Rule 5`, and that it has now been changed. Therefore, `New Rule 3` and `New Rule 5` must be added to the rule queue. However, they are already on the queue, so nothing happens. If either of these two rules had not been on the queue, they would have been added to the end of the queue.

Now, consider this slightly changed set of rules, specifically `New Rule 2`.

Figure 7. Rule processing example three



`New Rule 2` is now also changing `Variable3`. This time, starting with step 3 from the previous sequence, the following is what happens:

1. If `Variable3` is false, then `New Rule 2` is taken off the queue and its condition checked. Note that at this point the rule queue would contain `New Rule 3`, `New Rule 4` and `New Rule 5`. `New Rule 2`'s condition states that if `Variable1` is not equal to `15` its action part must be processed. Let us assume that `Variable1` is indeed not equal to `15` and action statements change `Variable2` to the value "`A new value`", and `Variable3` to `true`.
2. What happens next in this case depends on the state transition statement of `New Rule2`. If it had caused a transition to another state, then the scenario would have emptied the rule queue, moved to that state, and then repopulated the queue with the rules from the new state. However, in this case the state transition statement is `continue`. Note that `Variable2` is referred to in the condition part of `New Rule 3` and `New Rule 5`, and that it has now been changed. Therefore, `New Rule 3` and `New Rule 5` must be added to the rule queue. Also, `Variable3` is referred to in the condition part of `New Rule 1`, and it has also now been changed. Therefore, `New Rule 1` must be added to the rule queue. Now, `New Rule 3` and `New Rule 5` are already on the queue, so they are not added. `New Rule 1` is no longer on the queue, so it is added. Therefore, at the end of processing `New Rule 2`'s action part, the rule queue will now be: `New Rule 3`, `New Rule 4`, `New Rule 5` and `New Rule 1`.

Basic view of rule processing

Scenario monitoring stage

If all rules on the rule queue are processed and the queue becomes empty, the scenario instance goes into a monitoring stage.

The scenario instance stays in this state until some external source changes a variable, block parameter or block field that is referred to in any condition of any of its rules. This can occur because of a user sending in a scenario modification, or a block changing its properties in response to some external event feed.

If this occurs, then the affected rules are added to the rule queue and processed in the order as described previously.

This process of placing rules on the rule queue and processing them continues until a rule condition is true and the corresponding action requests a state transition to another state. After moving to the new state, Event Modeler places the new rules on the queue and evaluates them. Rule processing stops only when there are no rules left to be evaluated.

Basic view of rule processing

Summary of adding rules when a variable value changes

When a rule action or an external source changes a variable, block parameter or block field that is referred to in any condition of any rule in the current state, that rule is added to the current rule queue, unless it is already on the queue. If the queue was empty when the rule was added, then the rule is processed immediately. If multiple rules need to be added to the queue, they are added in the order they are listed, top to bottom.

Basic view of rule processing

About scenario variables

Typically, each scenario has a number of *variables*.

As in other programming environments, variables are placeholders for useful information that the scenario needs to keep track of and perhaps modify during its execution. They also indicate the information that will be required by a running instance of the scenario from the end-user in order to configure and start it off, as well as representing the information that will be sent back to be displayed to the user as progress updates or results.

The variables defined in a scenario are shown in the Variables tab. Each variable has a distinct *type*. If you click on the green box to the left of each variable you can examine its type and other properties.

Overview of Using Event Modeler

Variable types

Variables can be of four types in Event Modeler:

- `Text` (or *string*)
- `Number` (*integer* or *float* depending on constraint)
- `Choice` (or *enumeration*)
- `True/False` (or *conditional*, or *boolean*)

`Text` variables contain textual information, like words, phrases or sentences. An example of valid text is "`Hello World`", "`Monday`", "`ACME`" or "`Trading Strategy executed successfully`". Text values are normally shown in double quotes. If you want to have quotes in your text, you can escape them as follows: "`he said \"hello\" and left`".

`Number` variables can contain numbers. Valid examples are `1`, `25.0`, `-45.62`, or `8902e8`.

`Choice` variables are constrained so that they can only have values from a specific set of pre-defined values. For example, the choice variable `Day` could be constrained so that it can only have one of the values "`Monday`", "`Tuesday`", "`Wednesday`", "`Thursday`", "`Friday`", "`Saturday`"or"`Sunday`".

`True/False` variables, also known as *condition* variables, can only take the values `true` or `false`.

You can also specify constraints on variables according to their type. For example, you can specify maximum and minimum values for a `Number` variable.

About scenario variables

Auto-typing of variables

Variables are "auto-typed" by default. This means that the type is automatically inferred from the value assigned to the variable in the Variables tab . If such a variable is wired to another variable or a block field, it inherits the source's type.

If you subsequently change the wiring so that the auto-typed variable is then wired to another variable or block field, its previously inferred type will be changed to the type of the new source.

Note that this means that type mapping (as described in "Linking variables, block parameters, and block output fields" on page 29) will not be necessary for variables that are auto-typed.

About scenario variables

Variable properties

Variable properties only apply to, and are enforced by, dashboards. That is, they only apply when a variable is presented to, and is interacted with by, an end user of the scenario. By design, variable properties do not apply to scenario rules or variable wiring within the Variables tab.

Each variable has a mutability property, which can take the following values:

- `Mutable` – This property is of relevance to the dashboard. If set it means that the end-user should be able to set and change the value of this variable at any time, via a dashboard.
- `Immutable` – This property is of relevance to the dashboard. If set it means that the end-user should only be allowed to set the value of this variable upon creation of the scenario instance, and should not be able to modify it afterwards.
- `Fixed` – This means that this variable is a constant; it cannot be modified through a dashboard. If a variable is set as `Fixed` but no value is provided for it in the Variables tab, the Event Modeler will automatically set it to the default value for its type.

Furthermore, each variable can also be set to be `Unique`. This means that if multiple instances of a scenario are started concurrently, the value of this variable must be unique across all instances. The dashboard used to enter values for this variable will ascertain that this is the case before accepting the value from the user. Note that if a variable is set to be `Unique`, it must also be `Immutable`.

About scenario variables

Variable constraints

Depending on its type, each variable can also have *value constraints* set on it.

Variable constraints only apply to dashboards. That is, they only apply when a variable is presented to, and is interacted with by, an end user of the scenario. By design, variable constraints do not apply to scenario rules or variable wiring within the Variables tab.

For `Text` variables the possible constraints are:

- `Minimum length`: a whole number specifying the minimum acceptable length of the text string. Setting this constraint is optional.

  For example, if set to `5`, then "`book`" would not be valid, but "`library`" would.

- `Maximum length`: a whole number specifying the maximum acceptable length of the text string. Setting this constraint is optional.

  For example, if set to `8`, "`library`" would be a valid value, but "`librarian`" would not.

- One of `All Upper Case`, `All Lower Case` or `Mixed Case`. One of these constraints must be set, `Mixed Case` being set by default.

  For example, if set to `All Upper Case`, "`test`" and "`Test`" would be invalid, but "`TEST`" would be fine. Conversely, only "`test`" would have been valid if set to `All Lower Case`, but all three variants would be fine with the default `Mixed Case` setting.

- `Trim Whitespace`: If enabled, all leading and trailing white space characters (space, tabs, new line and other formatting characters) will be removed from the text string whenever its value is set. Note that if the minimum length and maximum length constraints were set, they would apply to the final 'trimmed' text string. The default is for this constraint to be disabled.

  For example, " `Hello World` " would be automatically changed to "`Hello World`" if `Trim Whitespace` were enabled.

For `Number` variables the possible constraints are:

- `Minimum`: a number specifying the minimum acceptable value of the variable. Setting this constraint is optional.

  For example, if set to `2` or `2.0`, then only numeric values greater than or equal to `2.0` would be valid.

- `Maximum`: a number specifying the maximum acceptable value of the variable. Setting this constraint is optional.

  For example, if set to `5` or `5.0`, then only numeric values less than or equal to 5.0 would be valid.

- `Whole Number`: If enabled, all values set for this variable will be changed to whole numbers by being *rounded down*. The default is for this setting to be disabled.

  For example, `3.1` would be automatically changed to `3`, as would `3.9736`., while `-3.1` would be changed to `-4`.

For `Choice` variables, the constraints specify the set of valid text values that this variable can take. These are distinct values, and choice variables can only take the values specified in their constraints.

For example, the choice variable `Day` should have its constraints set to the set of values "`Monday`", "`Tuesday`", "`Wednesday`", "`Thursday`", "`Friday`", "`Saturday`" and "`Sunday`".

No constraints are available for `True/False` (condition) variables.

About scenario variables

User input and output

Each scenario variable can be tagged as being an input variable, an output variable, or both.

Variables whose values can be collected directly from the user should be marked `input`. Those whose value can change during the execution of a scenario, and whose changing values may be of interest to the user, should be marked as `output`.

About scenario variables

About blocks

Blocks are ready packaged modules that you can use in your scenarios. They can accept inputs, execute some logic of their own, and generate output.

A block is defined in a *Block Definition File*, or `.bdf`. This XML file describes the functionality of the block and its implementation in Apama Event Processing Language (EPL), which is the new name of Apama MonitorScript. EPL is the native language of the correlator.

Note: Within the product, both EPL and MonitorScript are used and should be treated as synonymous.

A block can consist of:

- *Input feeds* – an input feed can be hooked up to a live stream of event data, like a price quote stream. Within it, an input feed will define one or more *input fields*, which can be mapped to data

in the stream. When event data arrives, the fields' values are updated. These fields are typed in the same way as scenario variables.

- *Output feeds* – an output feed is a stream of output data that can be generated by the block. Each output feed corresponds to an event that can be generated by the block, and embeds one or more *output fields*. The fields are updated as a result of operations carried out by the block. These fields are typed in the same way as scenario variables.
- *Parameters* – a block can have a number of parameters, which, when set, configure its behavior. Parameters differ from input fields, in that the latter are like work packages for the block to process. Typically, you use parameters to initialize the block or change its core behavior. Parameters are typed in the same way as scenario variables. Parameters are all provided at initialization time and can then be updated individually. Input fields are expected to change often and at any time.
- *Operations* – in addition to any standard behavior that is hard-wired into it, a block can also have a number of explicit operations that can be invoked by the scenario. For example, typical operations are `start` and `stop`, which cause the block to begin processing events or to cease. If an operation requires any configuration information, this is usually passed in through a block parameter.

Apama provides a library of useful blocks, which can be viewed and selected from the Catalogs tab. For information about provided blocks, see "Using Standard Blocks" on page 120.

There is no restriction on the number of block instances that can be added to a scenario. The Blocks tab shows the blocks that have been added to a scenario. When you add a block to a scenario you are effectively specifying that instances of that scenario should create an instance of that block running within them. Whether the block instance then starts executing some activity immediately or waits for some operation on it to be called depends entirely on how the block itself was written.

It is possible to add multiple instances of the same block to a scenario. Each instance will have its operations, parameters and fields clearly tagged by its unique name to ensure there is no conflict.

If there is no standard block that meets your needs, you can create a custom block. There are several ways to do this:

- Use the Apama Studio block editor to create a block by defining its parameters, operations, input feeds and output feeds.
- Use the Apama Studio block editor to create a block from an event definition.
- Save a scenario as a block. This lets you create composite scenarios when you use such blocks in other scenarios. However, you cannot save a scenario as a block if you mark that scenario as parallel. Nor can you save a non-parallel scenario as a block and then mark the block as parallel-aware. For details, see "Working with Blocks Created from Scenarios" on page 84.

For more information on the structure of a block and for instructions on how to create your own blocks, see "Creating blocks" in *Using Apama Studio*.

Overview of Using Event Modeler

Linking variables, block parameters, and block output fields

One of the facilities provided by the Event Modeler is the linking of:

- Block output fields to scenario variables

  This creates a relationship between an output field of a block and a scenario variable. Once set up, Event Modeler automatically updates the value of the variable to the value of the output field. If the output field changes, the variable's value immediately reflects the new value of the block output field.

  If the field and the variable are not of the same type, Event Modeler converts the field's value to the type of the variable before it updates the variable. If the conversion is not possible, Event Modeler assigns a default value to the variable. See "Conversion rules for variable types" on page 65 for more information.

  If the variable is of auto-type, it inherits the type of the block output field.

  After you link a block output field to a scenario variable, you can still explicitly modify the value of the scenario variable. If you do, keep in mind that Event Modeler will continue to update the value of the scenario variable each time the value of the linked block output field changes. Consequently, after you link a block output field to a scenario variable, the recommendation is that you do not explicitly modify the value of that scenario variable.

- *Scenario variables* or *block output fields to block parameters*

  This creates a relationship between a scenario variable or block output field and a block parameter. Once set up, Event Modeler automatically updates the value of the block parameter to the value of the scenario variable. If the value of the scenario variable or block output field changes, the value of the linked block parameter immediately changes to reflect the new value.

  If the variable or field and the parameter are of different types, Event Modeler converts the variable's value or the output field's value to the type of the parameter before updating the value of the parameter. If the conversion is not possible, Event Modeler assigns a default value. See "Conversion rules for variable types" on page 65 for more information.

  After you link a scenario variable or block output field to a block parameter, you can still explicitly modify the value of the block parameter. If you do, keep in mind that Event Modeler will continue to update the value of the block parameter each time the value of the linked scenario variable or block output field changes. Consequently, after you link a scenario variable or block output field to a block parameter, the recommendation is that you do not explicitly modify the value of that block parameter.

Overview of Using Event Modeler

Chapter 2: Using Event Modeler

- Adding scenarios to projects 31
- Opening and viewing multiple scenarios 33
- Selecting from the Scenario menu 33
- The Event Modeler toolbar 34
- Interacting with the tabs and panels 35
- Working in the Event Flow panel 36
- Working in the Rules panel 43
- Using the Variables tab 58
- Using the Catalogs tab 66
- Using the Functions tab 68
- Using the Blocks tab 69
- Switching blocks 73
- Using the Block Wiring tab 74
- Troubleshooting invalid scenarios 79
- Setting preferences 79
- Exporting scenarios as EPL 81
- Exporting scenarios as block templates 81
- Event Modeler command line options 81

Now that the important concepts underlying the definition of a scenario have been introduced, this section will illustrate how to use the Event Modeler's interactive functionality.

This section will describe each of the tabs available in Event Modeler and how to use them effectively.

Adding scenarios to projects

To open or create a scenario, the scenario must belong to an Apama project. This section uses an example to show you how to create a project, create a new scenario, and add a scenario to a project:

- "Creating the GlobalRuleExample project" on page 32
- "Adding GlobalRuleExample.sdf to the GlobalRuleExample project" on page 32
- "Adding a new scenario to the GlobalRuleExample project" on page 32

Using Event Modeler

Creating the GlobalRuleExample project

The following steps provide an example of how to create an Apama project. To create the GlobalRuleExample project:

1. Ensure that Apama Workbench appears in the Apama Studio window title bar. If it does not, from the Apama Studio menu, select Window > Open Perspective > Apama Workbench.
2. From the Apama Studio menu, select File > New > Apama Project to display the **New Apama Project** dialog.
3. In the **New Apama Project** dialog, specify GlobalRuleExample for the project name, accept the default project location, and click Next.
4. In the list of standard bundles that appears, select Scenario Service (required by all Scenario-based applications), and click Finish.

   Bundles are packages of Apama objects such as EPL files, event definition files, and event files or adapter configuration files that are required for specific types of applications.

   Apama Studio displays your new project in the **Workbench Project View** pane on the left of the perspective.

Adding scenarios to projects

Adding GlobalRuleExample.sdf to the GlobalRuleExample project

To add `GlobalRuleExample.sdf` (an existing scenario) to the GlobalRuleExample project:

1. From the Apama Studio menu, select File > Import.
2. Expand General, click File System, and then Next.
3. Click Browse and then navigate to and select *`your_Apama_install_directory`*`\samples\scenarios`, and click OK.
4. In the **Import** dialog, select GlobalRuleExample.sdf and click Finish.
5. In the Workbench Project View pane, expand scenarios, and double-click GlobalRuleExample.sdf to open it in Event Modeler.

Adding scenarios to projects

Adding a new scenario to the GlobalRuleExample project

To add a new scenario to the GlobalRuleExample project:

1. From the Apama Studio menu, select File > New > Scenario.

2. Enter a name for the new scenario and click Finish.

Adding scenarios to projects

Opening and viewing multiple scenarios

In the Apama Developer perspective, Event Modeler can open multiple scenarios concurrently, but only one can be on active display; that is *on view* at any one time. You can tell which scenario is currently on view by examining the contents of the window title bar, as this lists the scenario's name and the location of its corresponding `.sdf` file.

At the top of the Event Flow/Rules display, there is a tab for each opened scenario. The last opened scenario always becomes the scenario on view. So depending on the sequence in which you open scenarios, one will be on view and the other will still be loaded. You can switch from one to the other by clicking its tab.

It is also possible to open multiple Apama Event Modeler windows and view different scenarios (or the same, for that matter) in each. This can be carried out from the Window menu on the menu bar, and is not the same as actually starting another instance of Apama Studio. There should never be any need to do the latter.

To open a window for each scenario:

1. From the Apama Studio Window menu, select New Window.

   Another Apama Studio window appears.

2. In this second Apama Studio window, you can open the same scenario or a different scenario.

Notice how the title bars reflect which scenario is on view in each window.

If you have multiple windows open showing the same scenario, any *edits* done in one will be immediately reflected in the other if applicable. Selections and view changes are not reflected in this manner; so if in one window you are viewing the start state while in another you are editing the rules of another state, you will not see your edits in the first window until you select the edited state there.

If you close a window, the scenario on view in that window remains loaded in the Event Modeler and no changes are lost. If you close all the windows in Event Modeler, you have effectively exited the Event Modeler. You will be prompted with a warning dialog if you try to exit Event Modeler while there are modified (unsaved) scenarios open.

Using Event Modeler

Selecting from the Scenario menu

When Event Modeler is open, the Apama Studio menubar includes Scenario. The nested options from Scenario are as follows:

- Generate Debug Code — When this is checked, Apama Studio injects the scenario in debug mode when it runs your project.
- Generate Block — When this is checked, Apama Studio saves your scenario as a block template when it saves and/or builds your project. Apama Studio puts the block template in the Generated scenario blocks catalog in the `catalogs` directory of your project. You can use the block template in

other scenarios. This option is available only if all of the scenario's states, and by consequence, all their rules' conditions and actions, are finished. You cannot mark a scenario as parallel and then export it as a block.

- Toggle Block Field Feed Name Display— In the Block Wiring tab, toggles the display of block field feed names.
- Toggle Rule Comment Display — In the Rules panel, toggles the display of the comments that can be associated with each rule.
- Global Rule Arc Visibility — Determines the Event Flow tab display of transitions controlled by global rules. Choices are:
  - Emphasize All Global Rule Arcs — All global transitions appear in a bright orange color.
  - Emphasize State Global Rule Arcs — The global transitions for only the selected state appear in bright orange. Other global transitions are in a very light orange.
  - Deemphasize All Global Rule Arcs — All global transitions appear in a very light orange color.

When you save a scenario, Event Modeler first tries to save a copy of the previously saved version of that scenario to create a backup. If Event Modeler is unable to make the backup, it displays a dialog that lets you know. You can save the scenario anyway or cancel and try to find out why the backup could not be made.

Using Event Modeler

The Event Modeler toolbar

The Event Modeler toolbar contains a number of icons that correspond to commonly used operations:

Table 1. Event Modeler toolbar

Toolbar icon	Operation
	Enable/Disable parallel execution — Indicate that the instances of the scenario will be run in parallel. This selection is a toggle. A scenario that runs in parallel executes each scenario instance in a separate context. Contexts let Apama organize work into threads that the correlator can concurrently execute. For a scenario to run in parallel, each block that it uses must be parallel-aware. If a scenario uses one of the standard blocks provided with Apama, the scenario must use the latest version of the block. If a scenario uses a custom block, you must have created it in Callback or Callback (DEBUG) mode, or converted it to Callback or Callback (DEBUG) mode. You cannot create a block from a scenario that can run in parallel. Also, you cannot create a block from a non-parallel scenario and then mark that block as parallel-aware.
	Cut the currently selected element to the clipboard (that is, copy it and then delete it)

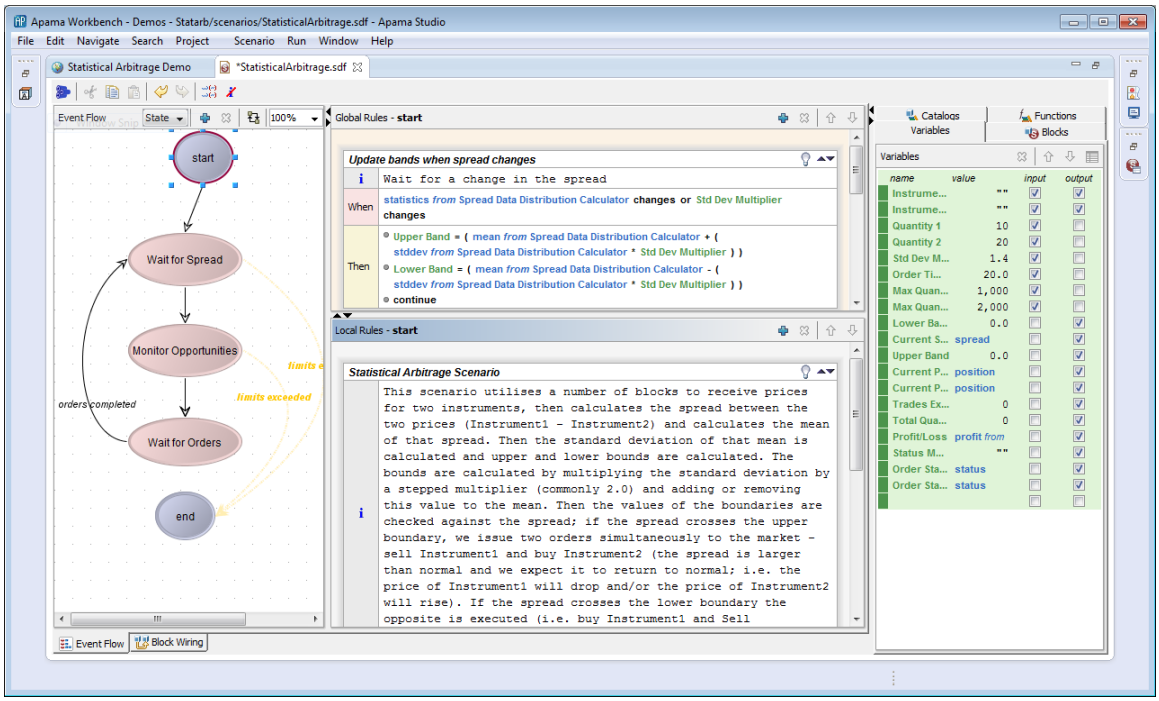
Toolbar icon	Operation
	Copy the currently selected element to the clipboard.
	Paste the current contents of the clipboard to the current selected location. This will not be available if the clipboard is empty or if its contents are not suitable for the current location. For example, you cannot paste a state in the Variables tab.
	Undo the last action.
	Redo the last action which was undone.
	Show feed names for block fields.
	Toggle display of rule comments.

Using Event Modeler

Interacting with the tabs and panels

Certain operations require you to highlight or select one of the panels first. You can do this by clicking somewhere within the desired panel or on its title bar. When a panel is highlighted, its title bar changes color as shown below. The Local Rules panel's title bar is highlighted because it is the selected panel.

Figure 8. Event Modeler with Local Rules panel highlighted



Using Event Modeler

Working in the Event Flow panel

The Event Flow panel graphically illustrates the states that a scenario instance can be in during execution, and how it can transit from one state to another. The states are depicted as circles, and possible transitions are shown as a line between the two states, with the arrow head indicating the direction of the transition.

Upon creation, a new scenario has two states, marked start and end, with a single transition going from the start state to the end state. User-defined states have a single border, while mandatory states, the start and end states, have a double border. Mandatory states are also shown in a different color (pale blue) instead of rose. The name of an unfinished state appears in red italics. In a newly created scenario, the start state is unfinished because you have not yet defined any rules to indicate how the scenario can transit from the start state to the end state.

Figure 9. Mandatory start and end states

Note that all colors used in the Event Modeler can be changed from the Preferences dialog. Select Window > Preferences , expand Apama, and select Scenarios.

You can zoom the view in and out within the Event Flow panel by changing the zoom value from the pull down selector available on the panel's toolbar. You can adjust the zoom level from 25% to 400%, with 100% being the default setting. Alternatively you can just type the zoom value you would like and press Enter.

Using Event Modeler

Interacting with states

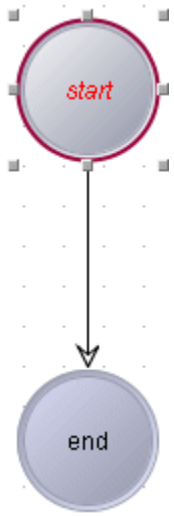
You can interact with states in the Event Flow panel in a variety of ways.

Working in the Event Flow panel

Selecting a state

If you click on a state you will notice that it becomes highlighted. This is indicated by the border changing color and eight *drag handles* appearing around the state.

Figure 10. Selected start state



If a state is selected its rules will be displayed in the adjoining Rules panel if this is viewable. When the title of the state is in red italic the state is unfinished. When the title of the state is in black the state is finished. See "The finished status" on page 38.

Interacting with states

Resizing a state

The drag handles allow you to resize the state in any of eight directions. Press and hold the left mouse button while pointing to any of the drag handles to resize while dragging. Notice how the mouse cursor changes to indicate that a directional resize is available.

If you hold down the Shift key while doing this, you will restore and preserve the aspect ratio of the circle.

Interacting with states

Moving a state

You can move a state around by pressing the left mouse button while pointing to it, and then dragging it around while holding down the mouse button.

If a state is selected its rules will be displayed in the adjoining Rules panel if this is viewable.

Interacting with states

Multiple selection

You can select multiple states concurrently by holding down the Shift key and clicking on multiple states; all will be selected. You can then drag them together by pressing and holding down the left

mouse button while pointing to any of them. If more than one state is selected, only the rules for the first one will be displayed in the Rules panel.

You can also drag and select a rectangle around multiple states and transitions.

Interacting with states

Adding a state

To add a state, click on the button on the Event Flow panel's toolbar. A new state will appear in the upper left corner of the Event Flow tab from where you can move it to a suitable location. This new state will be selected by default.

Interacting with states

The finished status

To inject a scenario into the correlator, or for the `Export EPL` functionality to be available, all its states must be *finished*.

For a state to be finished, all its rules must be properly defined. This means that they need to have valid fully specified conditions, and if any action statements have been added to them, those also need to be fully specified.

You can ascertain visually whether a state is finished or not by how its name is displayed in the Event Flow panel. If the name is in regular black font, then the state is finished. On the other hand, a red italic font for the name indicates that the state is unfinished, that is one or more of its nested rules are not fully defined.

Note also that if the scenario has changed since the last time it was saved to a file, it must be saved again before you can export it.

Interacting with states

Deleting a state

To delete a state, select it and then press the Del key, or click the button on the Event Modeler toolbar. If you selected multiple states, each of these actions deletes all selected states.

When you delete a state, if there are any rules with transitions to the deleted state, Event Modeler changes the transition section of those rules to transition incomplete. This makes the state that contains this rule incomplete. Event Modeler cannot generate EPL for this scenario until you complete the transition for this rule.

Interacting with states

Labeling a state

To change the label on a state, double click on the state. Type the new name of the state, and press Enter when done. While typing, you can press Esc to undo the edit.

You can label a state with any name you want. Note that state names do not have to be unique although it is recommended that you make them so. Otherwise it could be confusing to pick the correct one when defining the target for a transition from the list of available states.

Interacting with states

Using cut/copy/paste with states

You can *cut*, *copy*, and *paste* states.

For reference, recall that *cut* will copy the current selection into the clipboard and delete it from the scenario, while *copy* only places a copy of it in the clipboard.

To cut or copy a state, right-click it to display a context menu and select the operation you want. Alternatively, you can select it, and then do one of the following:

- Press the Control X and Control C shortcut keys
- Click the  or  buttons on the main toolbar, respectively.
- Select Cut or Copy from the Edit menu.

To paste a state, the Event Flow panel must be highlighted. You can do this by clicking somewhere within the Event Flow panel so that its toolbar is highlighted. If you right-click, you can select Paste from the popup context menu. Alternatively, press CTRL+ V, or click the  button on the Event Modeler toolbar. The newly pasted state is renamed to `Copy of previous_name` if there is still a state with the same name. For example, if you copy a state and then paste it back in, the newly pasted state will be renamed.

Note also that all rule transitions in the newly pasted state will be reset to `continue`. You can then manually change them to your intended transitions.

You cannot cut the start or end states.

If you want to make a copy of a state that retains all its transitions, you should use the Shift key to first select the state and then select each of the transitions you want to retain. Copy the entire selection into the clipboard, and paste it to obtain a copy of the state with the rule transitions' destinations preserved.

Interacting with states

Interacting with transitions

Once you have created your scenario's states, you can define transitions between them. The state where the transition starts is the *source* state, and the state where the transition ends is the *destination* state.

A transition in the Event Flow panel is the same as the action statement that defines it in the Rules panel. Any interaction with one affects the other; for example, deleting the transition link on the graph changes the rule's action statement to transition incomplete.

Working in the Event Flow panel

Adding a transition

You can add a transition in a number of ways:

- Having selected the source state, you can add a rule to it and then change the state transition statement for that rule so that it causes a transition to the destination state. This will automatically add the transition between the states in the Event Flow panel.

Adding rules will be described in "Working in the Rules panel" on page 43.

- Alternatively, click the icon on the Event Flow panel's toolbar to activate *Connect* mode. Small pale red squares, or *connectors*, appear around the border of all the states except the end state. Point to a connector on the source state and note how the cursor changes. Press the left mouse button, and while still holding it, move to another connector on the destination state. Release the mouse button to create a transition between the two connectors, and thus the two states.

  If you select the source state you will notice that a rule has been created in it that embodies the state transition you have just created. You can repeat this to create more transitions.

  Click on the icon again to deactivate *Connect* mode when done.

Interacting with transitions

Selecting a transition

In order to select a transition, click on it with the left mouse button. The transition changes color to a bold red to indicate it is selected. The corresponding rule is also highlighted in the Rules panel.

You can select multiple transitions by holding down the Shift key while clicking on them

Interacting with transitions

Changing end-points

If a transition is selected and *Connect* mode is not enabled, you can change the end-points of the transition.

Point to one of the end-points of the selected transition. The mouse cursor will change. Press the left mouse button and drag along the border of the state until another connector appears. Release to move the end-point of the transition to this connector, or keep on dragging to locate another connector.

There are eight such connectors around the border of each state.

You can also use this to drag the source or destination to another state. This will move the state or change the transition statement (for the target).

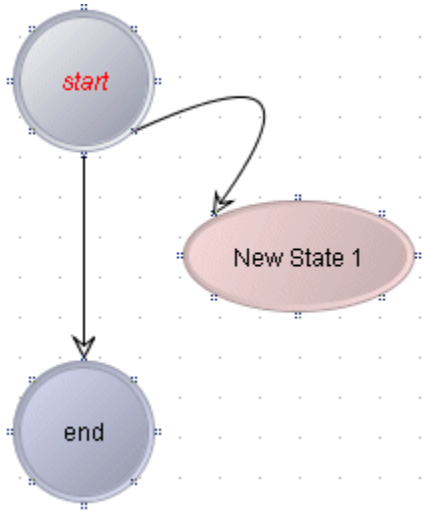
Interacting with transitions

Changing the shape of a transition

By default a transition will be a straight line between one state and another. You can change this into a curve if you wish.

Select the transition you wish to modify. Right click somewhere along the transition, ideally close to the centre of the line. A drag handle will appear on it. As before, press and hold the left mouse button while pointing to the drag handle, and drag to turn the line into a curve. You can do this at multiple points along the line to further shape the curve, and if you change your mind, you can delete each curve point by right clicking on its drag handle.

Figure 11. Changing transition shape



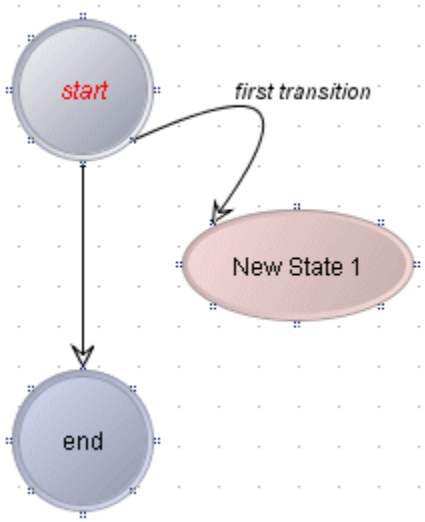
Interacting with transitions

Labeling a transition

To add a label to a transition, double click on the transition. A text entry box will appear in the middle of the transition. Type the text you want to use for its label, and press Enter when done. If, while typing you press Esc, the edit will be undone. The label will appear at the center of the transition line.

If you want to move the label, point to it with your mouse. Notice how the mouse cursor changes. Simply drag the label to the new position.

Figure 12. Moving transition label



Interacting with transitions

Deleting a transition

To delete a transition, select it and then either press the Del key, or click the button on the Event Flow toolbar.

When a transition is deleted, the action statement that defined that transition will be deleted.

If you selected multiple transitions, or even a selection of states and transitions, a delete operation deletes all selected entities at the same time.

Interacting with transitions

Using cut/copy/paste with transitions

You can *Cut*, *Copy* and *Paste* transitions, although note that this is identical to doing this with the associated rules.

To cut or copy a transition, right-click it and select the desired operation from the popup context menu. Alternatively, you can select it in the Event Flow panel, and then press the Control X and Control C shortcut keys, or click the or buttons on the Event Modeler toolbar, respectively. This is the same as cutting or copying the transition's associated rule from the Rules panel.

To paste a transition, the Rules panel must be highlighted. You can do this by clicking somewhere within the Rules panel so that its toolbar is highlighted.

Then you can press Control V, or click the button on the Event Modeler toolbar. The newly pasted rule is renamed to `Copy of its_previous_name` if there is still a rule with the same name within that state. For example, if you copy a transition or rule and then paste it back into the same state, the newly pasted one will be renamed. The transition's destination state will be preserved provided that the destination state still exists. If not, it will revert to `continue`.

Interacting with transitions

Displaying global rule transitions

Global rule transitions are dotted orange lines. You can choose to have them appear in a very light shade so they do not clutter the Event Flow panel. At the top of the Event Flow panel, click State to display the drop-down menu.

- All — Displays all global rule transitions in bright orange.
- State — Displays in bright orange the global rule transitions for only the selected state.
- None — De-emphasizes all global rule transitions. They appear as a very light orange.

The current selection always appears in the Event Flow panel toolbar.

Interacting with transitions

Working in the Rules panel

The contents of the Rules panel change whenever a state is selected in the Event Flow panel. It then lists those rules that a scenario must process when it enters the selected state. Global rules apply to two or more states; a local rule applies to only one state.

A new state does not have any rules defined in it.

Using Event Modeler

Adding a rule

To add a global rule, click the button on the Global Rules panel toolbar. The Event Modeler adds this new rule to every state except the end state.

Local rules can be added in the following ways:

- Select the state to add the rule, and then click on the button on the Local Rules panel toolbar.
- In the Event Flow panel, in *Connect* mode, manually add a transition between two states. This creates a new local rule with that transition defined in it within the source state.

The new rule is added to the bottom of the list of local rules.

A new rule will have the default title, "`New Rule n`", no description, an *unfinished* condition indicated by the red font of the rule name, and an action containing only a state transition statement.

You cannot add a rule to the end state. After a scenario enters its end state, nothing more can execute. If you want to do some cleanup before you terminate a scenario, add a cleanup state that comes just before the end state.

Working in the Rules panel

About global rules

When a state has both global and local rules, Event Modeler starts processing with the first global rule. If Event Modeler processes all of a state's global and local rules, it starts at the top, works through the global rules, and then works through the local rules.

To create a global rule, click the Add a New Global Rule button in the right part of the title bar of the Global Rules panel. This adds the new global rule to every state except the end state. If you add a new state after you create a global rule, Event Modeler automatically adds any global rules to the new state.

If you do not want a global rule to apply to a particular state, select that state, and then click the Activate/Deactivate button in the top right corner of the global rule. This toggles whether the selected rule is processed for the selected state. See "Activating and deactivating rules" on page 46 for more information.

To determine which states a global rule applies to, click the global rule to select it. All states that this rule applies to have dashed orange borders. If a global rule is unfinished the title of the rule appears in red italics and the titles of all states that the global rule applies to appear in red italics in the Event Flow pane. The Problems view displays information about any unfinished global rules.

There is an example of a scenario that uses a global rule in the `scenarios\samples` directory of your Apama installation directory.

Working in the Rules panel

Selecting rules and rule elements

To select a rule so that you can carry out operations on it, click on any empty space within it. The rule will become highlighted, with its border turning to a bold red. If the rule selected defined a state transition (that is, not `continue`) the corresponding transition will be highlighted in the Event Flow panel.

You can select multiple rules by holding down the Shift key while clicking on the rules to select.

To select a rule element, left-click it. To select multiple, contiguous rule elements, move the cursor over one of the elements, hold down the left mouse button, and drag the cursor over the other elements.

To select a rule element and display a popup selection menu for that element's position, right-click the element. This version of the selection menu also has the Cut/Copy/Paste options at the bottom. To display a more narrow selection menu for an element, hold down the Shift key and right-click the element. To select multiple, contiguous, rule elements and display a selection menu, move the cursor over one of the elements, hold down the right mouse button, and drag the cursor over the other elements.

Working in the Rules panel

Re-ordering rules

A rule's position in the listing of rules in the Rules panel is important because of the rule evaluation procedure described in "About rule evaluation" on page 20. Rules are always added to the rule queue in the top-to-bottom order shown in the Rules panel.

You can change a rule's position by selecting it, and then using the  and  icons on the Rules panel toolbar to move the rule upwards or downwards, respectively. You can use the Ctrl and Shift keys to select multiple rules at the same time and move them as a group.

The icons are only available when a rule is selected and their function is available for that rule. For example, you cannot move the first rule further upwards.

Working in the Rules panel

Deleting a rule

To delete a rule, select it and click the icon in the Rules panel toolbar. You can also press the Del key to achieve the same effect if you are not editing the rule's title or description.

If the rule has a state transition defined in its action part, the corresponding transition in the Event Flow panel will be deleted.

If you have multiple rules selected, any of the above variants will delete all of them in one step.

Working in the Rules panel

Labeling a rule

The first visual element of a rule is its title. Its function is just to assist you in visually identifying rules and is not pertinent to rule processing. The rule title is, however, included in logging information when debug mode is enabled, and therefore constitutes a very useful diagnostic tool. It is therefore recommended that you name rules. The title does not have to be unique, and by default all new rules are titled "`New Rule n`".

Double click with your left mouse button on the title of a rule to be able to edit it. You must press Enter when you are done to save the new title. If you press Esc your edits will be cancelled.

Working in the Rules panel

Changing a rule's description

The next visual element, indicated by the symbol , is an optional description of the rule's purpose. You can hide or show rule descriptions by clicking in the Event Modeler toolbar, or by selecting Scenario > Toggle Rule Comment Display in the Apama Studio menu.

It is advisable to set a description that explains what condition the rule is checking, what actions it is carrying out, and its effect within the scope of the overall scenario's logic. This helps when reviewing states and rules at a later stage, more so if another person other than the scenario's author is doing the reviewing.

Working in the Rules panel

Minimizing and maximizing a rule

Note the two icons to the right of the rule's title: and . If you click on once, the rule will be minimized to just its title, its description if it was showing, and the When section. If you click on it again, only the title and the rule description will be left showing. If you click to hide the comments, only the rule title appears.

You can then use to revert it back to either the title and condition, or the entire rule with title, condition and action. If necessary, click to display the rule's description.

Working in the Rules panel

Cutting, copying, and pasting rules

You can *Cut*, *Copy* and *Paste* rules.

To cut or copy a rule, right-click it and select the desired operation from the popup context menu. Alternatively you can press the Control X and Control C shortcut keys, or click the  and  buttons on the main toolbar, respectively.

To paste a rule, the Rules panel must be highlighted. You can do this by clicking somewhere within the Rules panel so that its toolbar is highlighted. You can also right-click in the Rules panel and select Paste from the popup menu. Alternatively, press Control V, or click the  button on the Event Modeler toolbar. Note that the newly pasted rule will be renamed to `Copy of previous_name` if there is still a rule with the same name. For example, if you copy a rule and then paste it back into the same state, the newly pasted one will be renamed. The rule transition's destination state will be preserved provided that the destination state still exists. If not, it will revert to `continue`.

You can also use Cut/Copy/Paste with rule elements. For example, you can copy a text variable element from a "`variablechanges`" statement and paste it into a text expression element.

You can also drag and drop rule elements to copy them. To do this, first select the rule element. Then hold down the mouse button and drag the element to the location to which you want to copy it. Not all elements can be copied to every other rule element. For example, you cannot copy a number expression and paste it into a condition expression. When you drag an element over its intended target, Event Modeler highlights the target in green if the copy is allowed and in red if the copy is not allowed.

Working in the Rules panel

Activating and deactivating rules

A deactivated rule is excluded from the EPL code generation and deployment. The

 button in the top right corner of each rule acts as a toggle to activate and deactivate the selected rule. Deactivated rules have a grey background to distinguish them from active rules which have normal white backgrounds. Also, if a rule has a transition associated with it, it will not appear on the state graph when its rule is deactivated.

An invalid rule prevents Event Modeler from exporting EPL for the scenario. If the rest of your scenario is valid and you want to export it as EPL, you can deactivate an invalid rule to generate the EPL. The EPL generator ignores deactivated rules.

Working in the Rules panel

Specifying conditions

In the Event Modeler Rules panel, the condition part of a rule is denoted by When. Every rule specifies a condition that must evaluate to true or false. When the condition evaluates to true, Event Modeler executes the action part of the rule, which is denoted by Then.

Working in the Rules panel

Interactive editing

As described in "How rules define scenario behavior" on page 16, there is a rich syntax available for defining conditions. Traditionally, you would expect to have to learn the language for defining conditions, specify a condition in a rule, and then have some facility that will check your input and inform you whether or not it is valid.

Event Modeler takes a different approach, in that it provides for graphical programming. With graphical programming, you assemble the condition by selecting from a number of options, gradually piecing it together. The advantage of this approach is that you do not necessarily need to know the intricacies of the language in any great detail and will be unable to make syntactic mistakes. With a little practice you can rapidly become as fast as someone who is typing in the condition.

Specifying conditions

Language elements

The interactive editing function is provided by the Condition Editor. Using the Condition Editor is very straightforward, but some terminology should be clear in order to assist with explanation.

Text in the condition part consists of a number of *elements*, which can be one of two types:

- *non-terminals* — elements that are not yet fully defined and are acting as placeholders to be replaced with further elements
- *terminals* — elements that are fully defined, and actually constitute the proper text of the condition.

An example will make the distinction clear. If it is not already open, open the Limit Order scenario as a template for exploring the Condition Editor features. See "Adding scenarios to projects" on page 31.

Ensure you do not save any changes as this might render the sample unusable. It is recommended that you make a backup copy of the `.sdf` file. After the scenario is open:

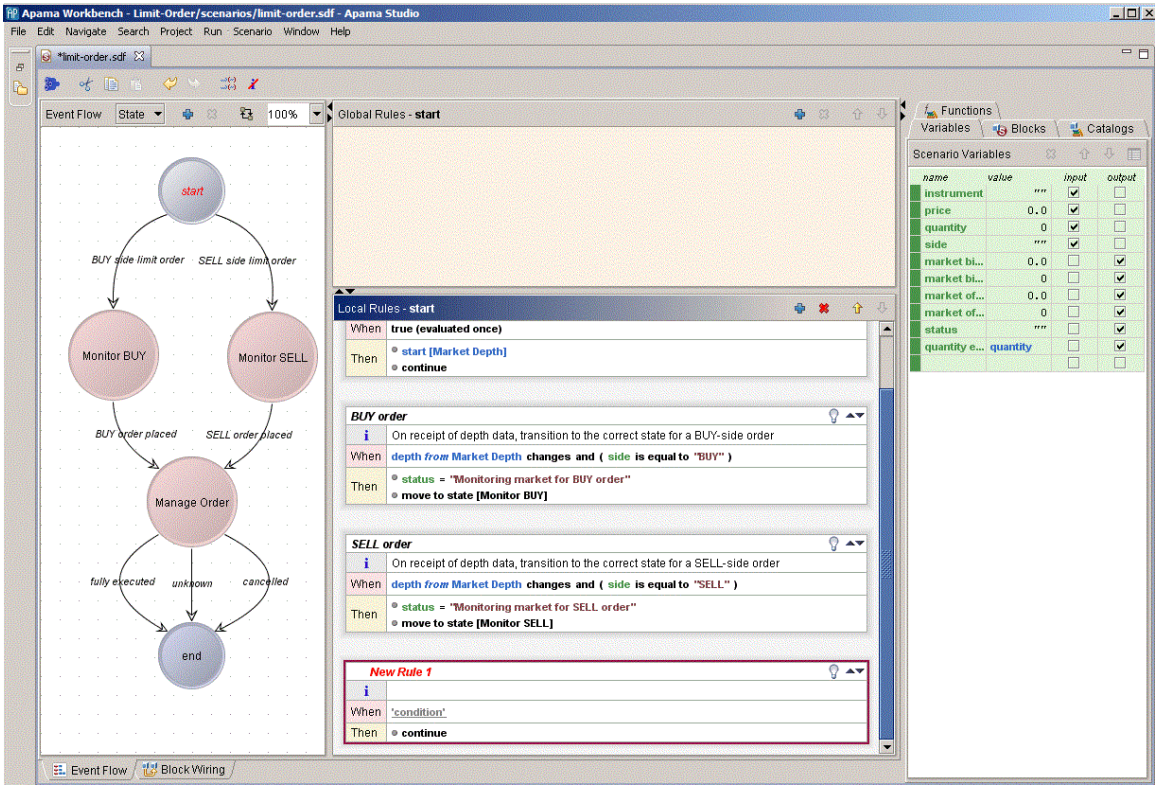
1. Click the start state to select it.
2. Click the  symbol in the Local Rules panel toolbar to add a new rule to the start state.

   The new rule is added to the bottom.

A new rule starts off with the condition part containing the text `'condition'`. Note that the word `'condition'` is in quotes and also underlined. Both quotes and underlining indicate that this is a non-terminal, that is, it still needs to be replaced with more precise text for the condition to be *finished*. Because the rule is unfinished its name appears in red italics.

The Event Modeler window will look as follows:

Figure 13. Event Modeler with new rule



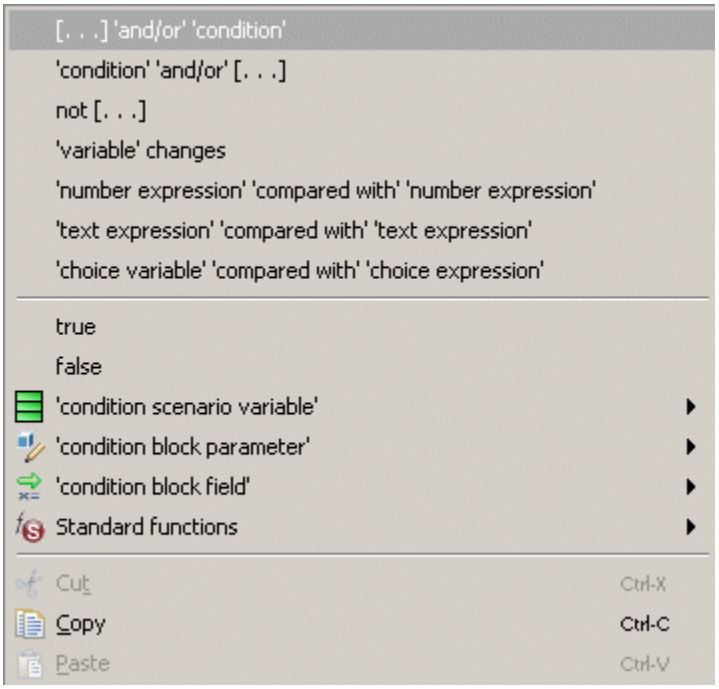
Because this condition is unfinished, the rule, the state and indeed the entire scenario are now unfinished. You can observe that a state is unfinished by the fact that its title is displayed in red italics text, as with the start state in this case.

Specifying conditions

Selecting and replacing elements

1. Right click on the `'condition'` non-terminal to see what it can be replaced with.

   A pop-up menu with several *alternatives* will appear.



Note that some of the alternatives themselves have elements with quotes to indicate that they are non-terminals that would need to be replaced in turn. The alternatives shown are always those with which the current *selection* can be replaced. There can be a distinction between what's selected and what's highlighted, as will be shown shortly.

2. Choose either of the first two alternatives:

   `[...] 'and/or' 'condition'`

   `'condition' 'and/or' [...]`

   The condition editor replaces `[...]` with the selected text.

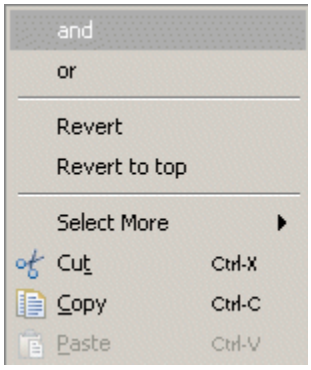
   The text inside the condition part changes from `'condition'` to `'condition' 'and/or' 'condition'`. All selections will be reset.

3. Right-click on the middle non-terminal, `'and/or'`, to see what its available alternatives are.

   They are `and` and `or`.

4. Choose `and`.

5. Now right-click on the new `and` terminal to see its alternatives.



Although it is a finished element, you can change it to `or`. Also, you can select Revert to set it back to what it was before the previous operation. Or, you can select Revert to top, which sets the value back to the value it had as far as possible in the hierarchy of changes. In this example, Revert and Revert to top have the same result.

Choosing Select More lets you select more of the condition statement. In this case, you can select the whole statement.

Working in the Rules panel

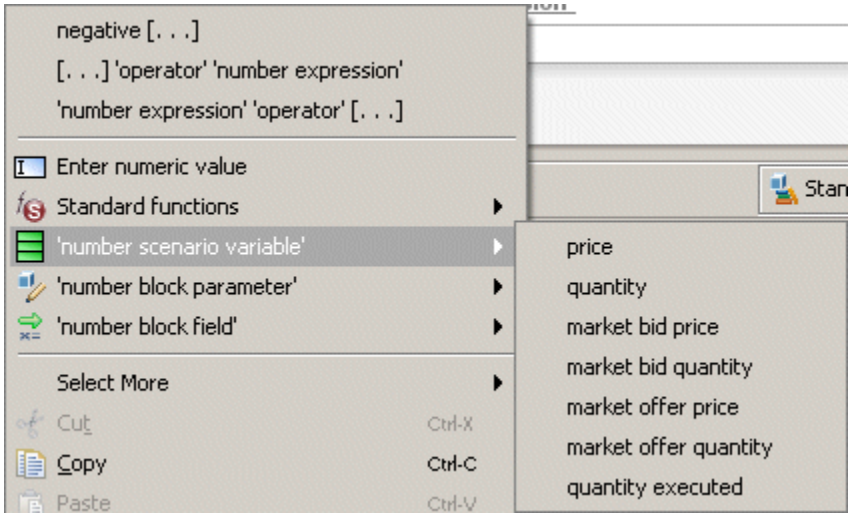
Cascading alternative menus

To fully define the condition:

1. Move the cursor over either instance of `'condition'`, hold the right mouse button down and drag to select the whole condition statement. This displays a popup menu that lists the elements that can replace the selected elements.
2. Choose `'number expression' 'compared with' 'number expression'.`

   All three highlighted elements will be replaced with `'number expression' 'compared with' 'number expression'.`
3. Right-click the first `'number expression'` to display its alternatives.

   When an alternative consists of a single non-terminal, the Condition Editor looks ahead to see what it could be replaced with in turn, and provides those choices in a further cascading menu. This accelerates the process of defining a condition. This is recursive.
4. Point to `'number scenario variable'` to be shown which scenario variables of number type are available.



5. Choose `price`.
6. Right-click `'compared with'` and select `'is less than or equal to'.`
7. Right-click the remaining `'number expression'` and choose either of these alternatives:

   `[...]'operator' 'number expression'.`

   `'number expression' 'operator' [...]`

Notice how the editor has added brackets around these latest replacement elements to improve clarity when a condition starts to get complex:

```
price is less than or equal to (‘number expression’ ‘operator’ ‘number expression’)
```

8. Right-click the first `‘number expression’` and select `‘Enter numeric value’`.

   A dialog will appear in which you can supply a number. The dialog indicates the expected format for your locale.

9. Enter a number, like `25.36`, and click OK to accept it.

10. In a similar fashion replace `‘operator’` with `*`.

Working in the Rules panel

Using functions in rules

To use a function in a rule:

1. Right-click the remaining `‘number expression’`, and from the alternatives in the context menu, point to `Standard functions`.

   This displays a listing of all the functions available in the Event Modeler that return a number as a result.

2. Choose `ABS ( ‘number’ value )`.

   A function is selected slightly differently to other elements. If you click on the function name you will select the function itself, and can thus replace it. If you click on any of its parameters (if it has any), then you can replace just the parameter. Click the Functions tab to display information about available functions; see "Using the Functions tab" on page 68.

3. Select the `‘number expression’` parameter, and replace it with the scenario variable `quantity`, by choosing `‘number scenario variable’, quantity`.

The condition is now complete.

There are no unfinished elements, or non-terminals, in it. No elements have quotes or are underlined.

And if you glance over at the start state in the Event Flow panel, you will notice that the name of the state is now back to regular black font.

Working in the Rules panel

Adding a condition to a rule

Suppose that when you have finished the condition defined in "Using functions in rules" on page 51 you realize that you only want it to evaluate to true if a condition scenario variable is also true. So you want to add an `and` with another condition clause to the end of the condition you have already specified, without having to revert it all and start all over again.

You can do this as follows,

1. Select the entire condition by moving the cursor over the condition, holding down the right mouse button, and dragging until all elements are highlighted. This displays a popup menu of alternatives for the selected elements.

   Now, remember these two alternatives:

   ```
   [...]‘and/or’ ‘condition’
   ‘condition’ ‘and/or’ [...]
   ```

   What this means is that if you select one of those alternatives, because the selection you are replacing is already a `‘condition’` in itself, it will not be thrown away but will be retained within the new replacement in place of the `[...]`.

   So, if you choose the `[...] ‘and/or’ ‘condition’` alternative, the current selection will be retained and will replace `[...]`.

2. Do that to see this result:

   ```
   (price is less than or equal to (25.36 * ABS (quantity)))
   ‘and/or’ ‘condition’
   ```

   If you had chosen `‘condition’ ‘and/or’ [...]`, then `‘condition’ ‘and/or’` would have been added to the front of your previous elements, not after.

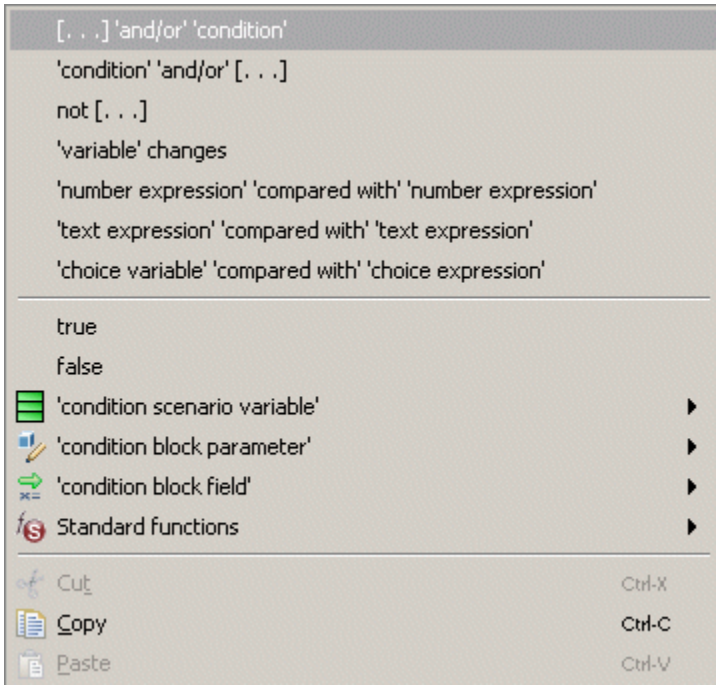
This replacement mechanism is automatically provided wherever an alternative for the current selection embeds an element of the same type as the selection itself.

Working in the Rules panel

Specifying variable changes in conditions

When you define a rule’s condition, you can choose 'variable'changes from the condition popup menu. For example:

1. Add a new rule.
2. In the new rule, right-click `'condition'`, which displays this popup menu:



3. Select `'variable' changes`. This replaces `'condition'` with `'variable' changes`.
4. Right-click `'variable'`, which displays a menu of the variables you can specify. As you can see, this menu lists the scenario variables, and it then lists the blocks that the scenario uses. If you select a block, you can then select the variables in that block. The variable in the `'variable' changes` expression can be one of the following:
   - Scenario variable
   - Block output feed
   - Field in a block output feed
   - Block parameter

When you select 'variable' changes, it can be the entire condition, or it can be an expression in a condition. Following are a few examples of specifying 'variable' changes in a condition:

- `When  quantity changes`
- `When  quantity changes or price changes`
- `When  quantity is greater than 20 and price changes`

A changes expression can become true as follows:

- When the variable in the changes expression is a block feed, any update that causes the block to send that output feed changes the condition to true. It does not matter whether or not the values of any fields in the output feed actually change.
- When the variable in the changes expression is a scenario variable, a block field, or a block parameter, a change in the value of that variable causes the 'variable' changes expression to be true. For example, if you assign the value `5` to the quantity scenario variable and the quantity scenario variable already has the value `5`, then there is no change and the 'variable' changes expression remains false.

Suppose that a 'variable' changes expression in a condition becomes true and the entire condition becomes true. When this happens, Event Modeler does two things:

- Executes the rule's action.
- Resets the value of the 'variable' changes expression to false. This ensures that two rules that specify the same variable in a changes expression can each trigger their action as a result of the same change.

Beyond this, the behavior of a 'variable' changes expression varies according to whether the condition appears in a global rule or a local rule.

Working in the Rules panel

Local rules and variable changes

When there is a transition to a state, any 'variable' changes expressions in local rules are initially false. Any changes made in previous states do not affect any changes expressions in the new state. For a changes expression to become true, the specified change must occur in the state to which the rule, which specifies the changes expression, applies.

Specifying variable changes in conditions

Global rules and variable changes

When there is a transition to a state, a 'variable' changes expression in a global rule can be initially true or false.

In a global rule, the 'variable' changes expression is initially true when all of the following are true:

- In a previous active state, the 'variable' changes expression became true but there was a transition to another state before the associated rule was triggered.
- Since the 'variable' changes expression became true, it has not triggered execution of an action.
- The scenario has not passed through a state for which this global rule was deactivated.

Remember that when a true 'variable' changes expression triggers a rule, the Event Modeler resets the value of the 'variable' changes expression to false.

In a global rule, the 'variable' changes expression is initially false in each of the following situations:

- The active state is the first state during scenario execution for which the global rule is activated.
- The global rule was not activated in a previous state and since that state was active the variable of interest has not changed.
- The global rule was triggered in a previous state and since that state was active the variable of interest has not changed.

For example, suppose states `1`, `2`, and `3` each define global rule `x`, which specifies `price changes` as its condition. There is a transition to state `1`. Initially, the `price changes` expression is false, but while state `1` is active the `price` variable changes and the `price changes` expression becomes true. However, there is a transition to state `2` before execution triggers global rule `x`. Global rule `x` is activated for state `2` but there is a transition to state `3` before execution triggers global rule `x` in state `2`. In state `3`, the `price changes` expression is still true. Execution triggers global rule `x`, performs the associated action, and resets the `price changes` expression to false. If global rule `x` has not been activated for state `2`, or if

global rule `x` has been triggered in state `2`, then the `price changes` expression would have been false when state `3` become active.

Specifying variable changes in conditions

Specifying actions

The second important part of a rule is its action part.

The action part of a rule is denoted by Then and consists of a number of *action statements* and a *state transition statement*.

When a rule is first created it has no action statements set.

The state transition statement

The state transition statement, already introduced elsewhere, specifies whether scenario execution should transit to another state if the rule's condition is true and once its actions are fully executed.

It can be `continue`, the default setting, which specifies that no transition is to occur, or be `move to state [a state]`.

You can modify the state transition statement by pointing to it and right-clicking. A pop-up menu will appear listing all the possible settings for the statement.

If you select any of the states, the state transition arrow will be set to `move to state [that_state]`. A corresponding transition will also appear in the Event Flow panel.

If a state transition starts and ends within the same state, a transition will still be added from that state to itself in the Event Flow panel. If you ensure that the rule is highlighted, the transition will be highlighted as well, and you will then be able to change its connectors and turn it into a curve. This will make it more visible.

Note that if you click on the state transition statement, i.e. with the left mouse button, and it is set to `move to state [a_state]`, you will be taken to that state. That is, the target state will be selected in the Event Flow panel, and the Rules panel will change to show the rules of that state.

Working in the Rules panel

Adding action statements

To add an action statement, *left* click the ● symbol to the left of the state transition statement.

1. Click the ● symbol to add an action statement.

   New action statements consist of the text `'action statement'` preceded by a ● symbol.

   In general, when you left-click a ● symbol Event Modeler adds an action statement before the line containing the symbol.

2. Click the ● symbol preceding the new action statement to add another action statement before it.

Specifying actions

Deleting action statements

To delete an action statement, *right* click the ●symbol to the left of the statement you want to delete. Note that you cannot delete the transition statement.

Click the ●symbol for the first action statement to delete it.

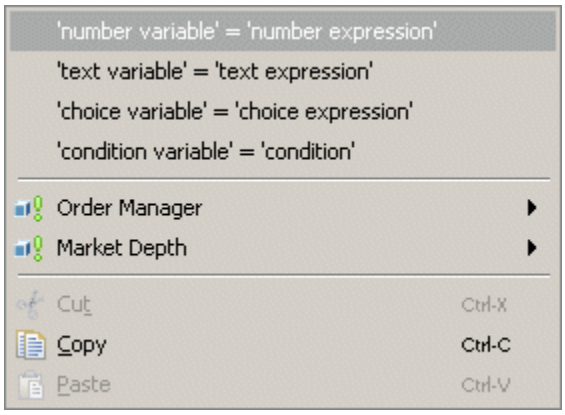
Specifying actions

Interactive editing

Once you have added an action statement, you need to specify the desired action using the *Action Editor*.

The Action Editor works on the same principles as the Condition Editor. See "Specifying conditions" on page 46.

Right click the non-terminal `'action statement'` to see its replacement alternatives.



As you can see from the alternatives available, the main difference is that action statements can either be assignments to variables or invocations of block operations.

There is a separate Action Editor for each action statement, and like the condition, all statements need to be *finished* for the rule to be finished. Feel free to explore the language elements and replacements available in action statements.

Specifying actions

Using the keyboard to edit rules

Instead of using the mouse, you can use the keyboard to edit rules.

Select one or more rule elements, and then press the Menu key  .

This displays the menu of choices for replacing the selected element(s). Use the cursor keys to select what you want.

The following table lists the other keys you can use to edit rules. Select one or more rule elements and then press the key.

Table 2. Using the keyboard to edit rules

Task	Key	Description
Add action	+	Inserts a new placeholder for an action statement below the condition or action that contains the selected element.
Delete action	-	Deletes the action statement that contains the selected element(s).
Display menu	Insert or Menu key	Displays the context menu for the selected element.
Edit literal	F2 or Enter	Displays a dialog in which you can edit the selected literal value.
Move to next rule	Page Down	Selects the first element in the next rule. If the focus is on the last rule, the focus stays where it is. If the focus is on a global rule, pressing this key selects the first element in the next global rule. If the focus is on the last global rule, pressing this key does not select the first element in the first local rule. The focus stays where it is. Note: if you selected the whole rule, so that the red, rectangular outline appears around it, pressing Page Down does nothing.
Move to previous rule	Page Up	Selects the first element in the previous rule. If the focus is on the first rule, the focus stays where it is. If the focus is on a local rule, pressing this key selects the first element in the previous local rule. If the focus is on the first local rule, pressing this key does not move to the last global rule. The focus stays where it is. Note: if you selected the whole rule, so that the red, rectangular outline appears around it, pressing Page Up does nothing.
Move to next element	→	Selects the next element in the statement. If the last element is already selected, pressing the left arrow key does nothing.
Move to previous element	←	Selects the previous element in the statement. If the first element is already selected, pressing the right-arrow does nothing.

Task	Key	Description
Move to next statement	↓	Selects the first element in the next condition or action statement. If the selected element is in the last global or local action statement, pressing this key does nothing.
Move to previous statement	↑	Selects the first element in the previous condition or action statement. If the selected element is in the first global or local condition, pressing this key does nothing.
Revert to top	Delete	Resets the selected element or elements as far back before any changes as possible.
Revert selection	Backspace	Resets the selected element or elements to its (their) previous value.
Select first element	Home	Selects the first element in the condition or action statement in which you had selected an element.
Select last element	End	Selects the last element in the condition or action statement in which you had selected an element.
Select multiple elements	Shift + →	Adds one or more subsequent elements to the selection. Event Modeler examines each subsequent element in order until it enlarges the selection to a set of elements that can be replaced as a unit. This might mean that only the next element is added to the selection, or that multiple subsequent elements are added.
	Shift + ←	Adds one or more previous elements to the selection. Event Modeler examines each previous element in order until it enlarges the selection to a set of elements that can be replaced as a unit. This might mean that only the previous element is added to the selection, or that multiple previous elements are added.

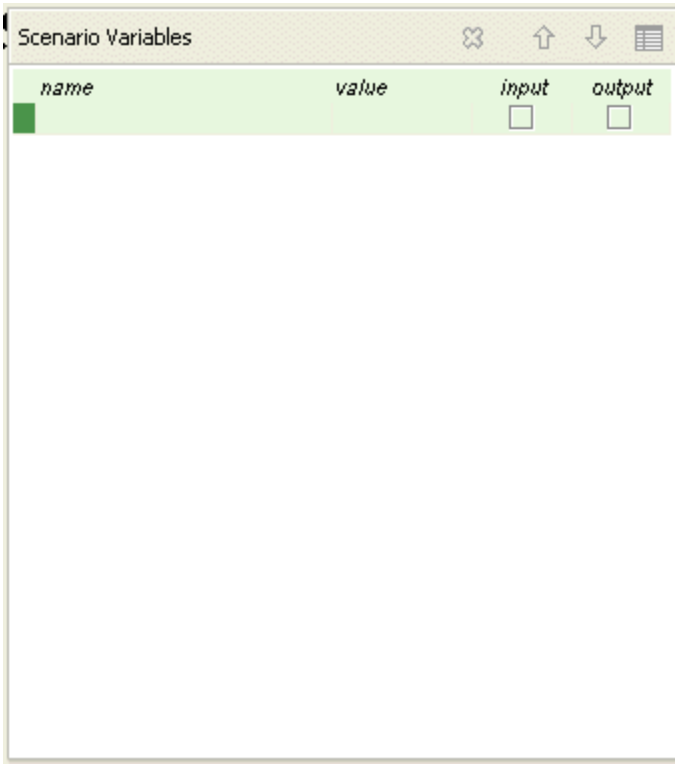
Working in the Rules panel

Using the Variables tab

The Variables tab lists and allows modification of all the variables available for use in a scenario.

In order to explore its features, create a new scenario by selecting File > New > Scenario from the Apama Studio menu.

Observe the Variables tab. Note the selection of buttons on its toolbar, and the fact that it contains a table, with two rows and four columns.



The first row contains column headings, while the second row appears empty. The variables table always displays a line for each variable defined, with a final empty line from which you can add new variables. In this case, no variables are yet defined, so the table only contains the final empty line.

The columns are `name`, `value`, `input` and `output`, and in addition each variable row has a dark green square to the left of it.

By default the background of these rows is green; green being used throughout the Event Modeler to denote scenario variables.

Using Event Modeler

Adding a variable

To add a variable:

1. Left click on the empty row in the cell under the column heading `name`. The cell will become highlighted with a border appearing around it. This means you can type in the cell.

   Alternatively you could double click on the cell, and this would display a flashing text entry cursor in the cell.

2. After selecting the name entry cell, type in a name for your new variable, like `var1`, and either click elsewhere or press Enter.

Note how a new empty line is added to the bottom of the table. The name of a scenario variable must be unique within the set of a scenario's variables. A scenario variable can have the same name as a parameter of a block that the scenario uses.

3. Create a second variable by clicking on the `name` cell in the final empty row and naming it `var2`.

Using the Variables tab

Renaming a variable

If you left click on the name of a variable to select its `name` cell, you can type in a new name, effectively renaming the variable.

Alternatively you can double click on the cell, and this will display a flashing text entry cursor in the cell, allowing you to edit the previous name.

Recall that variable names must be unique – if you type a name already in use it will revert to its old value on acceptance.

Using the Variables tab

Selecting a variable

If you want to carry out some variable operations, like moving a variable, or viewing its properties, you first need to select it.

You can do this by clicking on the green square at the beginning of each row. This selects the entire row. Notice how the icons on the Variables tab change to indicate they are now available.

You can select multiple variables in one go. Select the first one normally. Then, while holding down the Ctrl (Control) key, select any additional variables. Alternatively, hold down the Shift key to select all variables from the first one selected to the current one.

Using the Variables tab

Determining which states use a particular variable

Event Modeler displays a dotted green border around each state that uses the selected variable when you do either of the following:

- Highlight a row in the Variables tab by clicking on the green square at the beginning of the row.
- Click on a variable in a rule.

Using the Variables tab

Moving a variable

Once you have selected a variable you can move it up and down in the table by using the  and  symbols.

Changing a variable's position in the Variables tab has no effect on scenario execution other than appearing in that order whenever the scenario is opened from disk.

You can also select multiple variables and move all of them at the same time. Hold down the Ctrl key when you select each variable. Or use the Shift key select a range of variables.

You cannot move the last empty line, and cannot move variables below it.

Using the Variables tab

Deleting a variable

You can delete a variable by right-clicking its name and selecting Cut from the context menu.

Once you have selected a variable you can delete it by clicking on the  icon on the main toolbar or by pressing Del.

If you have selected multiple variables, they will all be deleted.

If any rules' condition or action parts refer to the variable you have removed, the references will be *reverted* back to their non-terminals. This will make those rules, and therefore the enclosing states and the scenario, unfinished.

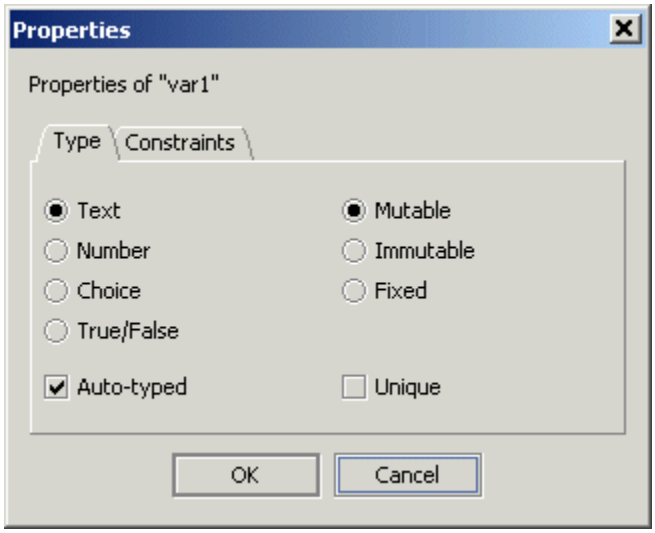
Using the Variables tab

Changing a variable's properties

Once you have selected a variable you can change its properties. To display a variable's properties, either click (again) on the green square at the left of its row, or else click on the  icon in the Variables tab toolbar.

This will display the **Properties** dialog.

This dialog has two tabbed panes, Type and Constraints.



Use the Type pane to change the variable's type and mutability properties.

Use the Constraints pane to specify what values are valid for that variable.

Remember that mutability properties and value constraints only apply to an end user's interaction with the scenario through a dashboard. They do not apply when a variable is wired to another variable or a block field, or to any assignments carried out in any action part of any rule.

Note that the constraints available change according to the variable's type, so the contents of the Constraints pane change dynamically as you select different types on the Type pane.

The options available for both panes have already been described in "About scenario variables" on page 25.

When a rule's condition or action parts refer to a variable, in the majority of cases those references are type specific. For example a `'condition variable'` non-terminal can only be replaced by a scenario variable that is of type `True/False`. Therefore, if you change the type of a variable after having used it in any rule conditions or actions, the references to it will be *reverted* back to their non-terminals if they become invalid. This will make those rules, and therefore the enclosing states and the scenario, unfinished.

Using the Variables tab

Setting a variable's value

Once you have created a variable you can also set its initial value. This is the value that the variable will have at the start of execution of any scenario instance before it is modified by the user or by an action in a rule.

You may have noticed that a default value is always displayed in the value cell. By default a variable is set to be `Auto-Typed`, and initially set to be of `Text` type with the empty string as its value — `""`.

You can change the initial value by clicking on the `value` cell for the particular variable, and then typing in the appropriate value, or else double clicking on the cell to get a text entry cursor. The former method over-writes any previous value; while the latter technique lets you edit the existing value.

If the variable is set to be `Auto-Typed`, you can type in any value. The variable's type will then be deduced, and may therefore be changed, by what you have typed in.

If you type any whole number (for example, `5`, `25`, `-145`) the variable will be set to `Number`, with the constraint `Whole number`. If you supply a number with a fractional part (`4.45`, `.68456`, `-23.`), the variable will be set to be a `Number` with no constraints. If you enter one of `true` or `false` (any mixture of case will work, for example, `TRUE`, `True`, `tRue`), the variable will be assumed to be of `True/False` (conditional) type. Everything else is taken to imply a `Text` variable.

If the variable is not `Auto-Typed`, you are only allowed to enter values that are valid according to the type of the variable and any constraints imposed on it. So, for example, if the variable is of `Number` type, you cannot enter "`Hello`" as a valid value. If you attempt to do so, the variable's value will be reset to the previously set value, or the default for that type if none had been set, that is `0` or `0.0`.

Using the Variables tab

Variable input and output

As described in "Variable constraints" on page 27, a variable can be marked as being an input variable, or an output variable, or both. These indicators are used by the dashboard to restrict which scenario variables it should make available to the end-user. For output variables it can also auto-generate specific functionality.

By default these indicators are off for each variable. Click on the check boxes in the `input` and `output` columns to set them. The space bar also toggles this on and off.

Using the Variables tab

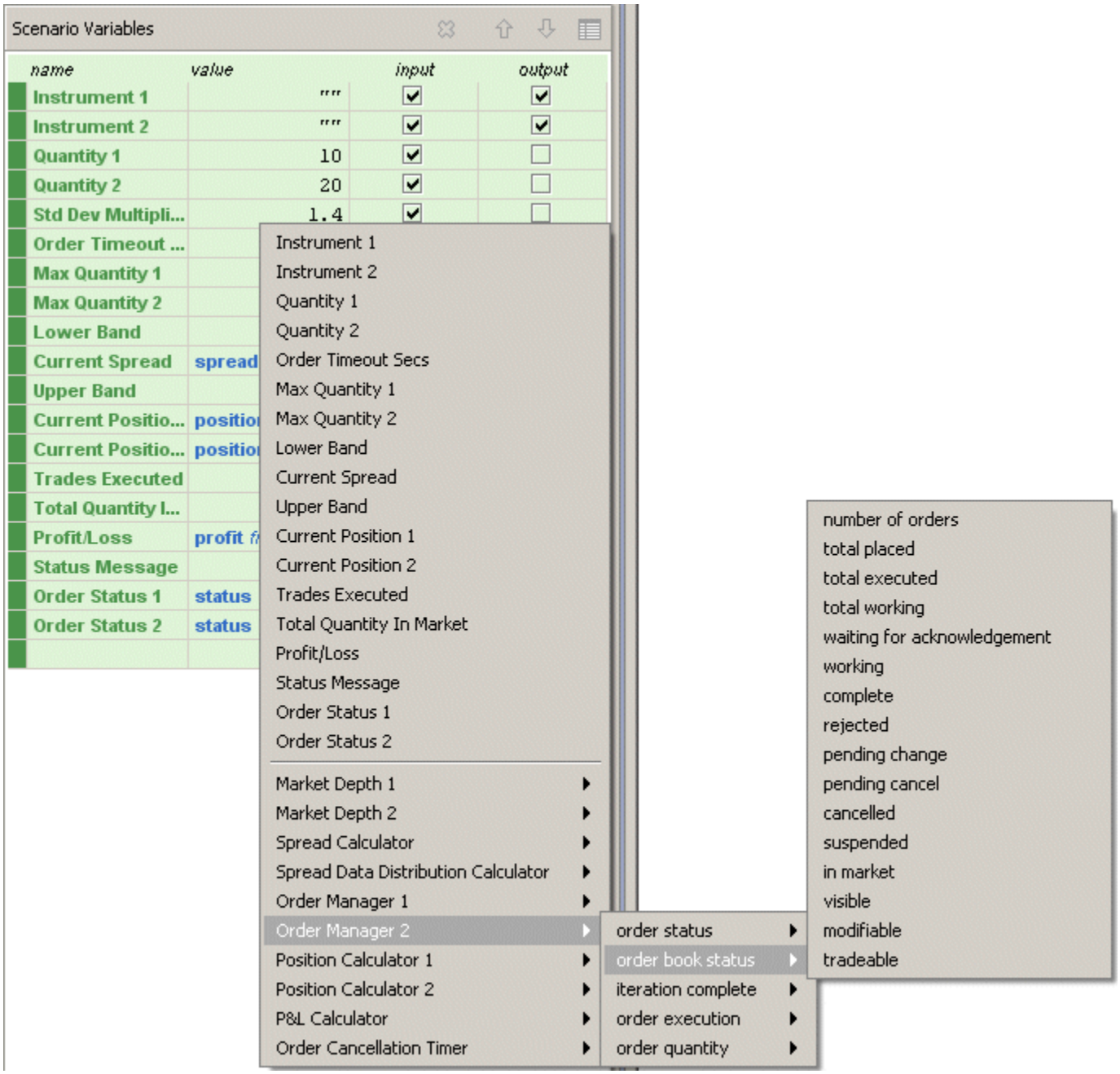
Linking a variable to a block output field

"Linking variables, block parameters, and block output fields" on page 29 described how one can set up a link between a scenario variable and another variable, or to the value of a block output field. Once this link is set up the variable will always have the same value as the source variable or the output field.

If the value of the source variable or output field changes, the destination variable's value will get updated automatically to be the same value.

You can set up such a link by right-clicking while pointing to the `value` cell for the variable to be linked. If the scenario contains any other variables or block instances with output feeds and fields, a pop-up menu will appear listing these.

Figure 14. Linking a variable to a block output field



When you select the output field to link with the variable, the field's name, preceded by the enclosing block instance's name, is displayed in the `value` cell.

The source variable or field chosen does not have to be of the same type as the destination variable.

If the destination variable is `Auto-Typed`, it can be wired to other variables or block output fields of any type, and will inherit their type once the wiring is carried out.

If it is not `Auto-Typed`, and it is not of the same type as the source, the source value will be converted to the destination variable's type before being copied to it. If this is not possible, a default value is set. See "Conversion rules for variable types" on page 65. For this reason, it is important to set up these links carefully.

Using the Variables tab

Conversion rules for variable types

This table summarizes the conversion rules:

Table 3. Conversion rules for variable types

	Number	Number (whole)	Text	Choice	Condition
Number	Copy the value	Copy the value and round it to the nearest integer value	Copy the value as a string.	Copy the value as a string.	`false`
Number (whole)	Copy the value	Copy the value	Copy the value as a string.	Copy the value as a string.	`false`
Text	Try to convert to a valid number up to the first non-numeric character, set to 0.0 if first character is not a number.	Try to convert to a valid number up to the first non-numeric character, set to 0 if first character is not a number.	Copy the value.	Copy the value.	If the value is `true` then set to true, else false. Case is ignored.
Choice	Try to convert to a valid number up to the first non-numeric character, set to 0.0 if first character is not a number.	Try to convert to a valid number up to the first non-numeric character, set to 0 if first character is not a number.	Copy the value.	Copy the value.	If the value is `true` then set to true, else false. Case is ignored.
Condition	1.0 for true 0 for false	1 for true 0 for false	Copy the value as a string.	Copy the value as a string.	Copy the value.

Examples

Text Source	Number Target
`"information"`	`0` or `0.0`

Text Source	Number Target
"-2.45"	-2.45
"456test"	456

Using the Variables tab

Using the Catalogs tab

The Catalogs tab displays catalogs of block templates that are available for use in a scenario. A catalog of block templates is a folder that contains one or more `.bdf` files, each defining a block template that the user can instantiate in a scenario. A catalog of block templates can also contain subfolders that themselves contain `.bdf` files. This hierarchical organization of a catalog appears when it is displayed in the Catalogs tab.

This text uses the term *block template* to refer to a block's definition on disk (within a `.bdf` file), whereas *block* is used to refer to an instance of a block template that has been added to the scenario.

The format and structure of a `.bdf` file is discussed in "File Definition Formats" in *Using Apama Studio*.

Typically, you might want to use multiple block template catalogs to distinguish between block templates supplied by Apama, block templates that you have developed yourself, and block templates that you have obtained from third parties.

In addition, within each block template catalog, as the number of block templates available to a scenario author could be very large it is useful to organize them into categories that reflect their functionality. Furthermore, as the block templates available are enhanced and new versions released, one is likely to need access to multiple versions of the same block templates.

A block template catalog's folder structure is therefore as follows:

- A root folder that represents the block template catalog, and within it,
- One or more sub-folders that represent functional categories of block templates, and within each,
- A folder called *`block_template_name`*`.bdf`, which contains
- The different available versions of a block template in distinct `.bdf` files.

The default block template catalog is simply called `blocks`. In the Catalogs tab, it appears as Standard Blocks.

Using Event Modeler

Adding a block template catalog

When Event Modeler is open it automatically makes the default catalog `blocks` available. If you have another block template catalog available on your system and want to make those block templates available to your scenario, use Apama Studio to add the block catalog to your project:

1. In the Apama Developer perspective, right-click the project name and select Properties.

2. In the **Properties** dialog, expand Apama and click Catalogs.
3. Click the Blocks tab and then Add.
4. In the **Source Folder Selection** dialog, click on catalogs to highlight it and click Create New Folder.
5. In the Folder name field enter the name of the catalog you are adding.
6. To add the complete contents of the catalog you specified, click Finish, and then click OK twice. You are done.
7. To choose particular files to add, click Next. Specify inclusion and/or exclusion patterns and click Finish. Then click OK twice.

Also use the Blocks tab in the **Properties** dialog when you want to remove a block template catalog.

Using the Catalogs tab

Selecting and inspecting a block template

The Catalogs tab is divided horizontally into two areas.

The top area displays the available catalogs. Expand each catalog to view its contents. When you select one of the following, a description of it appears in the bottom area:

- A particular version of a block template
- A block parameter
- A block operation
- A block input feed or input field
- A block output feed or output field

Using the Catalogs tab

Adding a block instance to the scenario

To add a block template to your scenario, first select it from the Catalogs tab. Open the folder it is in, select the block you want, and if there is more than one version, select the version you want. The recommendation is to use the most recent version, which is implemented in a way that delivers better performance than the older version. Also, the most recent version is parallel-aware. Older versions will be removed in a future release.

Then click on the  icon in the tab's toolbar to add this block to the scenario. You will see it appearing in the Blocks tab.This instance of the block template in the scenario will be automatically named. The name assigned will be the block template name followed by `1`, to indicate that this is the first instance of this block.

As implied, it is possible to add multiple instances of the same block to the scenario. These will be named sequentially to differentiate between them. The unique naming of each instance is important,

as all block instance feeds, fields, parameters, and operations are referred to from within rules by the enclosing block instance's name.

Using the Catalogs tab

Using the Functions tab

The Functions tab presents an organized view of the functions available for use in Event Modeler. The functions are organized in a folder hierarchy.

A function catalog allows you to organize a large number of functions into a manageable set of categories that indicate their functionality. A function catalog has the following structure:

- A root folder that represents the function catalog, and within it,
- One or more sub-folders that represent functional categories of functions, and within each of the sub-folders,
- `.fdf` files that define a group of related functions.

Such a catalog is installed by the Event Modeler installer. The default function catalog is simply called `functions`. To display this catalog, click the Functions tab.

Using Event Modeler

Adding a function catalog

When Event Modeler is open it automatically makes the default catalog `functions` available. If you have another functions catalog available on your system and want to make those functions available to your scenario, use Apama Studio to add the function catalog to your project:

1. In the Apama Developer perspective, right-click the project name and select Properties.
2. In the **Properties** dialog, expand Apama and click Catalogs.
3. Click the Functions tab and then Add.
4. In the **Source Folder Selection** dialog, click on catalogs to highlight it and click Create New Folder.
5. In the Folder name field enter the name of the catalog you are adding.
6. To add the complete contents of the catalog you specified, click Finish, and then click OK twice. You are done.
7. To choose particular files to add, click Next. Specify inclusion and/or exclusion patterns and click Finish. Then click OK twice.

Also use the Functions tab in the **Properties** dialog when you want to remove a block template catalog.

You must ensure that the `function name` attribute is unique within the directory in which you save the `.fdf` file. If you save a function definition file in a function directory that has been added to Event Modeler, and your new `.fdf` file does not have a unique `function name` attribute, you receive an error message about this when you open Event Modeler. You must resolve this error condition before you

try to use either of the duplicate functions. If you do not, you cannot predict which function Event Modeler will actually use when you call one of the duplicate functions.

Using the Functions tab

Selecting and inspecting a function

The Functions tab is divided horizontally into two areas. The top area lists the categories of functions in the catalog, and within each, the available functions. You can expand each function to view its parameters and return value. When you select a function name a description of that function appears in the bottom area.

Using the Functions tab

Using the Blocks tab

The Blocks tab lists all block instances that have been added to the scenario. From it you can select and delete a block, view its parameters, and link them to scenario variables or other block instances' output fields.

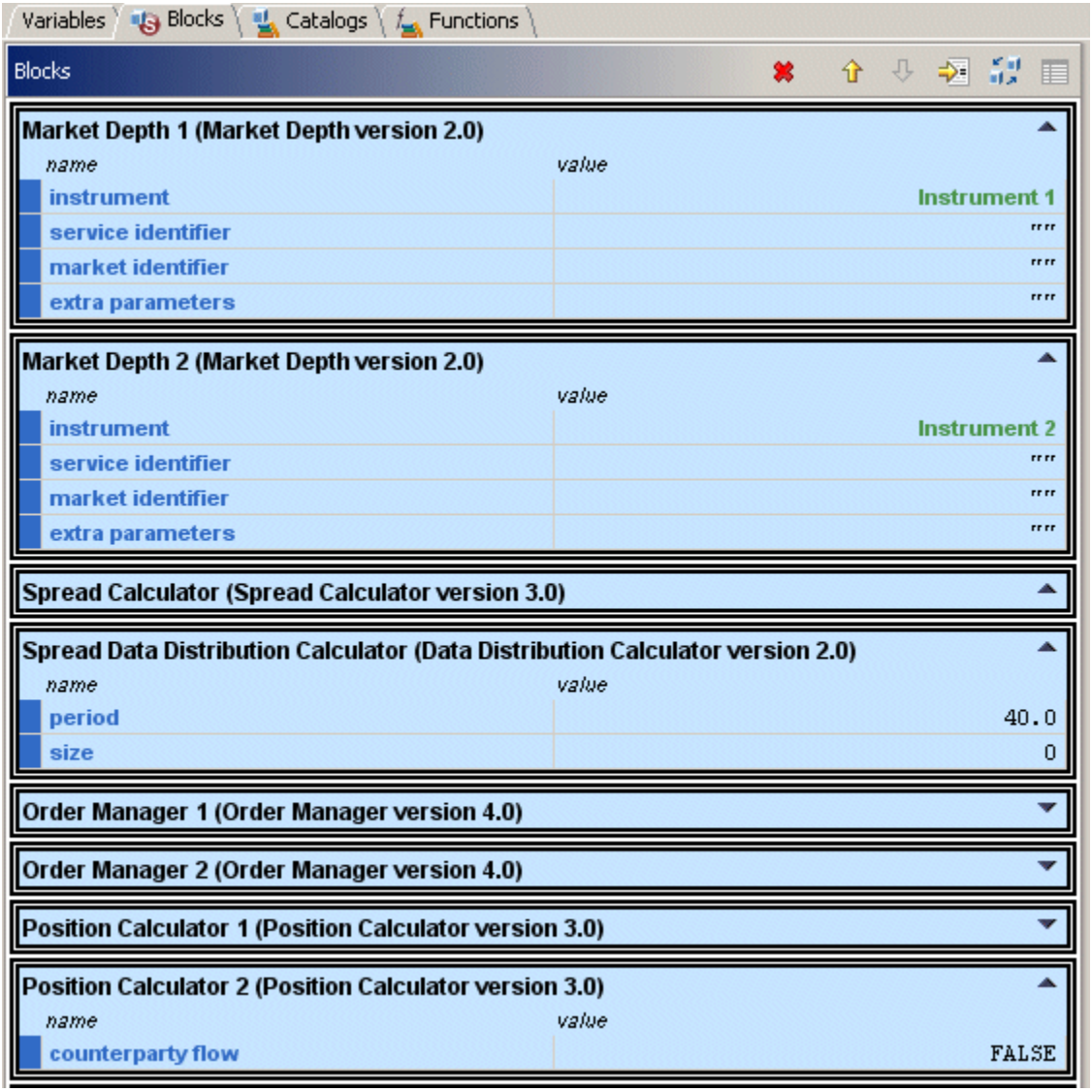
The Blocks tab is initially empty, but it then gets populated with block instances as you add these to the scenario from the Catalogs tab.

As you add block instances, each appears in the Blocks tab as a distinct element. By default, each is given a blue background, although this can be changed in the Event Modeler's preferences.

For each block instance, the representing element lists the block instance's name, and name of the block definition it was added from (this is in parenthesis), followed by a table with two columns, `name` and `value`.

Each row in the table contains a parameter, and similar to the table in the Variables tab, each is preceded by a solid blue square.

Figure 15. Sample Blocks tab



Note that a block does not have to have any parameters, and some of the standard blocks supplied by Apama are like this.

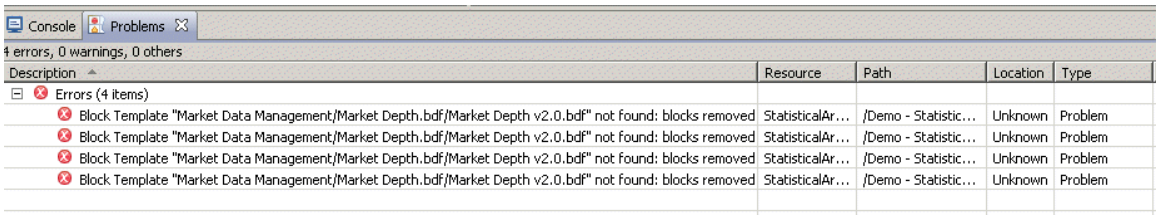
Interaction with this *parameters table* is similar to that in the Variables tab, with the distinction that it is not possible to add new parameters, rename them, re-order them, or change their properties. This functionality is not possible because the number, name and nature of block parameters is defined in the block's definition.

Once a block is added to the scenario, its parameters, output feeds and operations are available for interaction within rule conditions and actions. When a scenario is loaded the Event Modeler will reload that block's definition from its `.bdf` file and check that none of the referenced parameters, output feeds or operations have changed. If they have then any references will be *reverted* back to their non-terminals.

If you load a scenario and a block that you previously added to that scenario is missing Event Modeler reverts values of any variables that depended on that block's feeds to their default values. You receive a message that the block is missing when you open the scenario. Also, an entry for each missing block appears in the Problems view as shown in the figure below. Double-clicking on

a missing block entry in the Problems view displays the Block Wiring for the scenario without the missing blocks.

Figure 16. Problems view with missing blocks entries



Using Event Modeler

Interacting with a block instance

To select a block instance you need to left click somewhere within its display element other than inside its parameters table. For example, clicking on its name or on the table's column heading will select the block instance.

Once a block is selected,

- You can delete it by pressing Del, or by clicking on the icon in the toolbar.

  If any rules' condition or action parts refer to any feed, field, parameter or operation of the block instance you have removed, the references will be *reverted* back to their non-terminals. This will make those rules, and therefore the enclosing states and the scenario, unfinished.

- You can move the instance's relative position in the tab by clicking on the and icons in the tab's toolbar.
- You can browse the instance's block template definition in the Catalogs tab by clicking on the icon in the tab's toolbar.
- You can switch all references in rules and mappings from this block to another block by clicking on the icon. This operation is described in more detail later.
- Event Modeler displays a dotted blue border around each state that uses the selected block.

Another way to see which states use a particular block is to click that block in a rule. Event Modeler displays a dotted blue outline around the states that use the selected block.

Using the Blocks tab

Selecting a parameter

To select a block parameter, click on the solid blue square to the left of the parameter's name.

The entire row will be highlighted with a dark red background.

Using the Blocks tab

Viewing a parameter's properties

Once a parameter is selected, you can view its properties. You can do this by either clicking again on the solid blue square, or else by clicking on the  icon in the Blocks tab's toolbar.

This will display the **Properties** dialog. Properties for block parameters are almost identical to properties for scenario variables, with the distinction that the former cannot be modified in the Event Modeler. For this reason all settings in the **Properties** dialog will be grayed out. You can view them but you cannot change them.

Using the Blocks tab

Setting a parameter's initial value

As with scenario variables, block parameters need to have an initial value. This will be displayed in the `value` column. You can modify this initial value for each block instance's parameters by clicking on the `value` cell and typing in a new initial value. Alternatively you can double click on the `value` cell to edit the existing initial value.

Note that as with scenario variables, you are only allowed to supply an initial value that is compatible with the parameter's type and constraints (if any). If you specify an invalid value, the initial value will be reset to the default for that type.

Using the Blocks tab

Linking a parameter with a variable or output field

"Linking variables, block parameters, and block output fields" on page 29 described how one can set up a link between a block instance's parameter and the value of a scenario variable or block output field. Once this link is set up the block parameter will always have the same value as the source variable or block output field. If the value of the source variable or output field changes, the destination parameter's value gets updated automatically to be the same value.

You can set up such a link by right-clicking while pointing to the `value` cell for the parameter to be linked. If the scenario contains any variables or block instances, a pop-up menu will appear listing those variables, the block instances, their output feeds, and within those, their output fields.

When you select a variable or output field to link with the parameter, the variable's or field's name is displayed in the `value` cell.

The variable or field chosen does not have to be of the same type as the parameter. If it is not of the same type, its value will be changed to the parameter's type before being copied to the parameter. If this is not possible, a default value is set. See "Conversion rules for variable types" on page 65.

Since this could set the parameter to unexpected values, it is important to set up these links carefully.

Using the Blocks tab

Switching blocks

Consider the situation where you wish to replace a block in your scenario with another one. A common occurrence of this is if you wish to upgrade your block, for example by replacing version 1 of a block with a newer version 2.

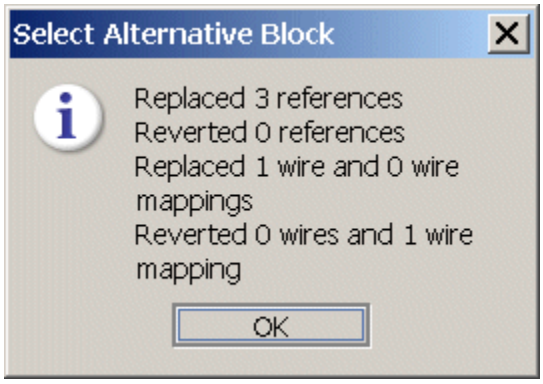
The problem with this is that if you delete the version 1, all references to its parameters, feeds and operations will be reverted or reset. You would then have to add the new block of the more recent version and re-establish all the references.

To facilitate this operation you can *switch blocks* as follows:

1. In the Catalogs tab, add the newer block to the scenario.
2. In the Blocks tab, select the block you want to replace.
3. In the Blocks tab's toolbar, click on  to be prompted for which block you want to use to replace the selected block.
4. Select the name of the replacement block from the choice list, and click OK.

Event Modeler tries to replace all references to the old block with the corresponding interface elements of the new one. Event Modeler also replaces the wiring of the old block with wiring for the new block.

At the end of the switching operation, a dialog appears that summarizes how many elements were replaced and which had to be reverted. For example:



Event Modeler can replace only those parameters, feeds and fields, and operations of the same name. If any elements do not have a corresponding element in the replacement block they will be reverted or removed, as follows:

- References are reverted to their non-terminals.
- In a wire mapping for which the source block output field has changed, the destination block input field is reverted to the default value for its type. For example, if the destination block input field is an integer, the field is reverted to `0`. The mapping itself is not removed even though it no longer has a source field.
- For a wire mapping for which the destination block input field has changed, the wire mapping is removed.

Using Event Modeler

Using the Block Wiring tab

At the bottom of the Event Flow panel, you can click the Block Wiring tab to replace the Event Flow and Rules panels with the Block Wiring tab. The purpose of the Block Wiring tab is to allow you to interactively define how your scenario's block instances are to be *wired together*.

Up to this point only block parameter wiring has been discussed. Recall that a block has parameters, input feeds, output feeds and operations. Parameters are intended for initializing the block, although they can then individually be updated during the block's lifetime to modify its operation. Input feeds, on the other hand, are normally used when a block's primary role is to process or transform some regularly changing data.

For example, the `Change Notifier` block's purpose is to generate a notification when the value of a numeric input data stream changes by a given amount over a configurable moving time window. Its parameters define the time window and the amount that the monitored values must change by to trigger the notification, while the actual values being monitored would of course be an input feed.

A block might accept input data while not having an input feed. This is normally because the block's author expects their block to be used alongside, and get all its input data from, dedicated EPL such as that included with external adapters. Good examples of this are the `Market Data Management` and the `Order Management` blocks such as `Market Depth`.

In general, a block is written to have exposed input feeds if its inputs can be provided by other blocks.

If you open a scenario and a block that was previously added to that scenario is missing you receive a pop-up error message, Event Modeler removes the block from the block wiring display, and there is an entry indicating the missing block in the Problems view.

Using Event Modeler

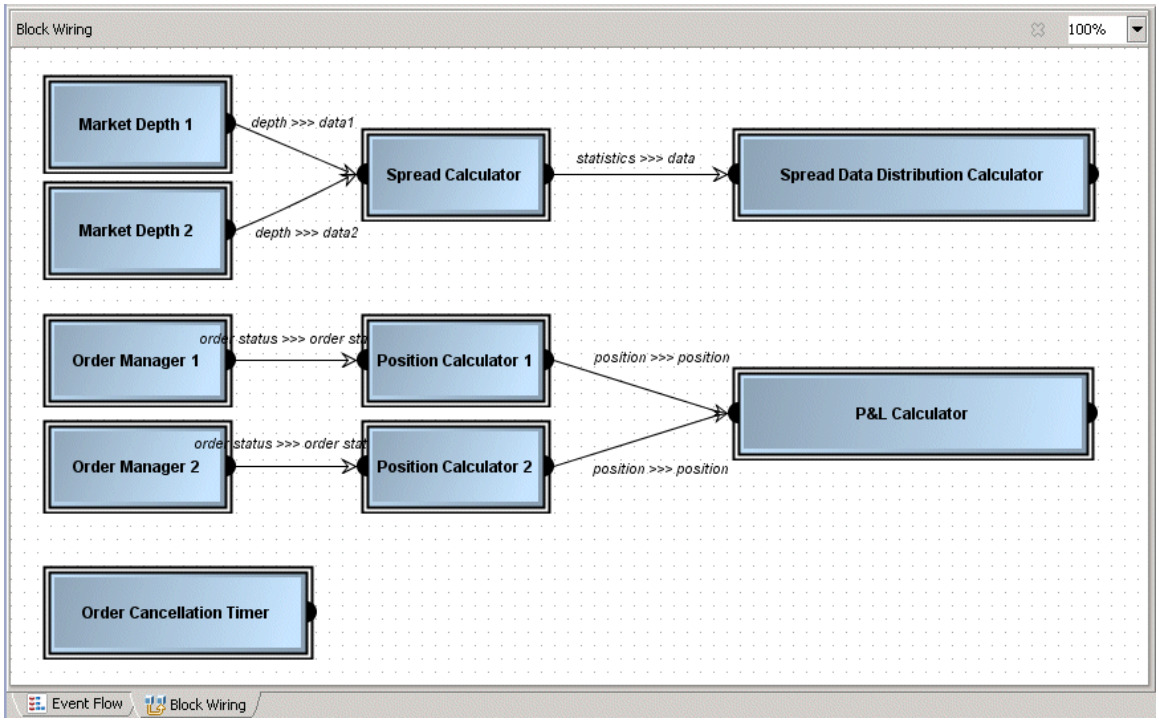
Wiring block input feeds

Two block instances are said to be wired together if one block's input feed is attached to the other's output feed. Output fields from the source block's output feed then need to be mapped (that is, connected) to the destination block's input feed's input fields.

The Block Wiring tab displays a solid blue labeled rectangle for each of the block instances that have been added to the scenario. Unless re-organized, these will initially be displayed in a partially overlapping stack at the top-left of the tab.

If a block instance has one or more input feeds, its rectangle will have a *wiring point* on the left hand side. This is a small solid black semi-circle. Similarly, if it has one or more output feeds, its rectangle will have a wiring point on the right hand side. Blocks with both input and output feeds exhibit wiring points on both sides. The figure below shows the Block Wiring tab.

Figure 17. Sample Block Wiring panel



Using the Block Wiring tab

Selecting, resizing, and moving block instances

Interaction with the block instances in the Block Wiring tab is similar to that in the Event Flow tab.

Click on a block instance rectangle to select it. The rectangle's border will become bold red and eight drag handles will appear around the rectangle.

To move a rectangle simply press and hold the left mouse button while pointing to it, and drag to the desired location. Release the mouse button to confirm the new location.

You can use the drag rectangles to resize the rectangle in any of the eight coordinates. As above, point to a drag handle, press the left mouse button and hold down while dragging the handle to the desired location. If you hold down the Shift key while dragging, you will restore and then preserve the rectangle's aspect ratio.

Using the Block Wiring tab

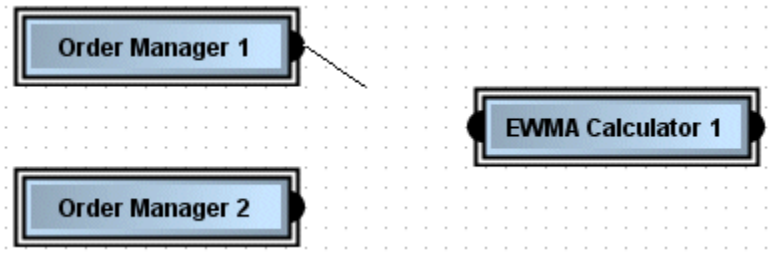
Wiring two blocks together

In order to wire two blocks together, it is best to place them side by side so that the *source* block instance is displayed on the left and the *destination* instance is to the right of it.

Then point to the output wiring point on the source block. Note how the mouse cursor changes. Press the left mouse button, and while holding it down, drag to the input wiring point on the

destination block. If a connection is possible the line being dragged from one wiring point to another will turn bold to indicate that you can now release the mouse button and create the wire.

Figure 18. Block wiring example



If you release the mouse button elsewhere, and when the line being dragged is not bold, then nothing will happen. You can try again.

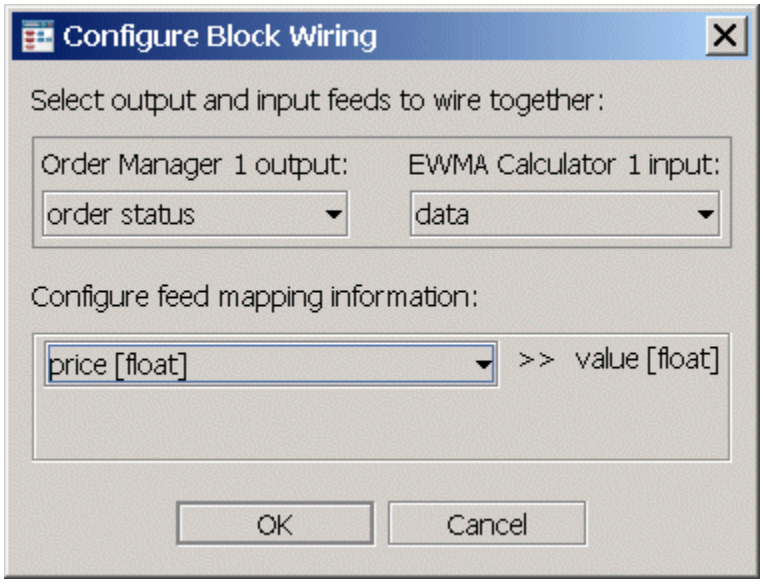
If you release the mouse button correctly at a point where the line can be created, then the Configure Block Wiring dialog will appear.

Using the Block Wiring tab

Connecting feeds and specifying feed mapping

The **Configure Block Wiring** dialog has two main areas.

Figure 19. Configure Block Wiring dialog



The first area is labeled "`Select output and input feeds to wire together:`". The bordered area underneath it will list all the output feeds of the source block instance on the left, and the input feeds of the destination block instance on the right.

Use the pull-down selectors for each block instance to define which feed should be mapped to which. Note that each wire corresponds to a *single* mapping of one output feed to one input feed.

Therefore once you have selected the output feed and the input feed, consider the second area of the dialog. This is labeled "`Configure feed mapping information:`".

Within the bordered area underneath this label you will see a listing of all the input fields contained within the input feed selected previously. To the left of each field you need to specify the source output field that is to be connected to it. Use the pull-down selector to view the output fields available and to create the mappings.

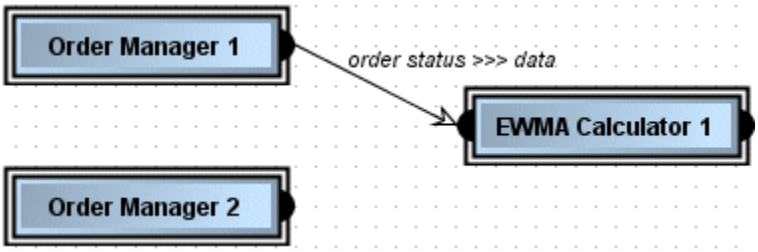
You can map a single output field to several input fields, or create distinct mappings for each.

At runtime, the field to field mapping will ensure that the input field of the destination block instance will always be kept the same as the value of the output field of the source block instance. When the output field changes, which might be very frequently, the input field will be updated immediately.

Alternatively, you can also just type in a value instead of selecting an output field. In that case the input field will become a constant, always containing the value you set. If you select the `*` option from the selector no mapping will be made, and the input field will be set to the default value for its type.

Click on OK to finish the wiring operation. A line will be displayed between the two block instances, labeled to indicate which feeds are involved in the wiring.

Figure 20. Block Wiring panel sample wire labels



Using the Block Wiring tab

Wiring a scenario variable to a block

You might want the value of a scenario variable to be the input for a block. To do this, use the `Variable Mapper` block. Wire the output of the `Variable Mapper` block to the input of the block that requires the scenario variable.

The `Variable Mapper` block takes the name of a scenario variable as the value of its only input parameter. When the value of the mapped variable changes, the `Variable Mapper` block sends the new value to its output feed. The output feed includes two values. The first value is the new value as a number. The second value is the new value as text. You can choose which representation you need to wire into another block.

Using the Block Wiring tab

Mapping type conversions

It is important to be aware that if the type of the source output field is not the same as the destination input field, type conversion will automatically take place.

The behavior here is the same as that already described when linking variables, parameters and output fields. That is, if the conversion cannot be carried out (such as when attempting to convert a non-numeric string to a number) then the destination field will be set to the default value for its type. See "Conversion rules for variable types" on page 65.

Using the Block Wiring tab

Editing block wiring

If you wish to edit the mapping of an existing wire just double click on the line representing the wiring.

Using the Block Wiring tab

Deleting a wiring

If you wish to delete an existing wire select the line representing the wiring by clicking on it. It will become a bold red to indicate it is selected.

You can then press Del to delete it, or else click on the  icon in the main toolbar.

Using the Block Wiring tab

Deleting a block instance

You can delete rectangles representing block instances. However, this is the same as deleting block instances from the Blocks tab.

To do this, select the block instance's rectangle, and then press the Del button. If that block had any wiring, either as a source or a destination, it will be removed.

If any rules' condition or action parts refer to any feed, field, parameter or operation of the block instance you have removed, the references will be *reverted* back to their non-terminals. This will make those rules, and therefore the enclosing states and the scenario, unfinished.

Using the Block Wiring tab

Using older versions of blocks

Apama 4.2 modified the interface for implementing blocks. All standard blocks have been updated to use this new interface. If you use a version of a block that implements the old interface, Event Modeler indicates this in the Block Wiring tab by using a different color around the perimeter of the block. Deprecated blocks (blocks that use the old interface and any blocks that are deprecated in the

future) have an orange border while current blocks have a black border. However, the selected block, of any type, has a red border.

You can use both deprecated and current blocks in the same scenario. However, if a scenario uses at least one deprecated block, the scenario instances cannot be run in parallel. In the Blocks tab and in the Block Wiring tab, blocks that are parallel-aware have a double-line border. Blocks that are not parallel-aware have a single-line border.

The recommendation is to update any custom blocks to the new interface. Support for the old interface will be removed in a future release. Information for converting custom blocks to the new interface is in the Apama 5.0 migration guide.

Using the Block Wiring tab

Troubleshooting invalid scenarios

Event Modeler does the following to help you troubleshoot scenario validation issues:

- An error in a scenario file causes Apama Studio to display an error icon in the Project Explorer panel on the scenario name, the scenarios folder, and the project folder.
- Apama Studio's Problems tab displays an entry for each error in a scenario.
- Double clicking a scenario error in the Problems tab opens the scenario that contains the error, if it is not already open, and selects the component associated with the error you clicked.
- If a global rule is incomplete (unfinished), the title of the rule appears in bright red, a red-outlined box appears around the rule definition, and the name of each state that the rule applies to also appears in bright red.
- If a local rule is incomplete the title of the rule appears in bright red, a red-outlined box appears around the rule definition, and the name of the state the rule applies to also appears in bright red.
- If a block is missing Event Modeler displays an error icon on the Block Wiring tab name 

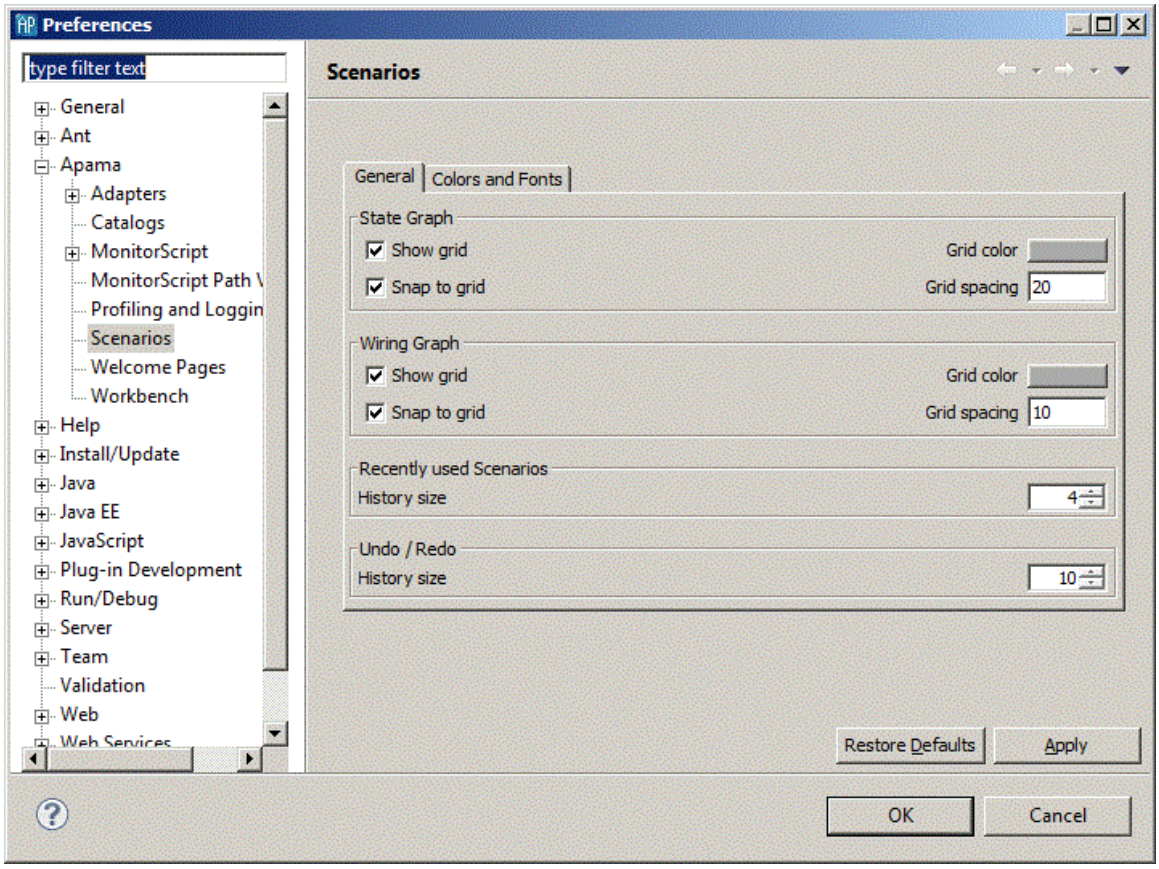
, removes the block from the wiring display, and displays an error in the Problems tab. This error identifies the missing block. Double clicking this error displays the Block Wiring panel that contained the missing block. The wiring display no longer shows the block that is missing and there is no error indicator in the wiring display for the missing block.

If there is a missing block whose feeds are used to set the values of scenario variables Event Modeler reverts the value of the scenario variable to its default value. No error indication appears.

Using Event Modeler

Setting preferences

You can display the Event Modeler **Preferences** dialog by selecting Windows > Preferences , expanding Apama, and selecting Scenarios:



The **Preferences** dialog has two tabs: General and Colors and Fonts.

The General tab contains these properties:

- State Graph
  - Show grid – Enable to show the grid in the Event Flow tab.
  - Snap to grid – Enable to turn on snap-to-grid in the Event Flow tab.
  - Grid color – Click on the color box to bring up a dialog from where you can choose a new color for the grid lines in the Event Flow tab.
  - Grid spacing – Enter a value to set the scale and spacing of the grid in the Event Flow tab.
- Wiring Graph
  - Show grid – Enable to show the grid in the Block Wiring tab.
  - Snap to grid – Enable to turn on snap-to-grid in the Block Wiring tab.
  - Grid color – Click on the color box to bring up a dialog from where you can choose a new color for the grid lines in the Block Wiring tab.
  - Grid spacing – Enter a value to set the scale and spacing of the grid in the Block Wiring tab.
- Recently used Scenarios

  History size – This setting specifies how many previously edited scenarios the Event Modeler should remember.
- Undo / Redo

History size – This setting specifies how many edits the Event Modeler should remember for the purpose of being able to undo them. The Undo button can be clicked multiple times to undo several actions at once, up to the limit set in this property.

Colors and Fonts allows you to change the colors and fonts of most of the graphical elements of the Event Modeler display.

Using Event Modeler

Exporting scenarios as EPL

To export one or more scenarios as EPL:

1. From the Apama Studio menu, select File > Export.
2. Expand Apama, select Export as MonitorScript, and click Next.
3. Select the project that contains the scenario(s) you want to export.
4. Select the scenario(s) to export and whether to export them in debug mode.
5. Identify the output directory for the generated EPL.
6. Click Finish.

Using Event Modeler

Exporting scenarios as block templates

To export a scenario as a block template:

1. From the Apama Studio menu, select File > Export.
2. Expand Apama, select Export as Block, and click Next.
3. Select the project that contains the scenario(s) you want to export.
4. Select the scenario(s) to export and whether to export.
5. Identify the output directory for the generated block template. By default, Apama Studio puts the generated block template in the Generated scenario blocks catalog in the catalogs directory of the project.
6. Click Finish.

Using Event Modeler

Event Modeler command line options

After you define a scenario, you can use a command line to generate EPL for that scenario, or to generate a block from that scenario. This might be useful for custom scripting. The Event Modeler executable is in the `bin` directory of your Apama installation directory. In addition to generating EPL or a block, you can use the command line format to obtain information about Event Modeler. Information about all Event Modeler command line options is in the table at the end of this topic.

Scenario to EPL

The command line format for generating EPL from a scenario is as follows:

```
event_modeler.exe -Xgenerate sdf_file_path EPL_file_path
```

sdf_file_path	Path of the scenario definition file for the scenario that you want to save.
EPL_file_path	Name of the new monitor.

For example:

```
event_modeler.exe -Xgenerate c:\dev\scenario1.sdf scenario1.mon
```

This example generates the `scenario1.mon` file from the `scenario1.sdf` scenario definition file.

Scenario to block

The command line format for generating a block from a scenario is as follows:

```
event_modeler.exe -XgenerateBlock scenario block catalog
```

scenario	Path of the scenario definition file for the scenario that you want to save as a block.
block	Name of the new block.
catalog	Path of the blocks catalog in which to save the new block.

For example:

```
event_modeler.exe -XgenerateBlock scenario1.sdf scenario1Block.bdf C:/Apama/blocks
```

This example generates the `scenario1Block.bdf` file from the `scenario1.sdf` file and stores the new block in `C:/Apama/blocks`.

All options

The format for executing `event_modeler.exe` is as follows:

```
event_modeler.exe [options] [scenarioFile1.sdf scenarioFile2.sdf ...]
```

Table 4. event_modeler options

`-h | --help`	Displays this information.
`-v | --version`	Displays Event Modeler version information
`-c | --conf` *file*	Path to Event Modeler configuration file. The default is `event_modeler_config.xml`.
`-l | --logfile` *file*	Identifies the name of the Event Modeler log file.
`-V | --loglevel` *level*	Specifies the log level.

`-f | --file` *file*	Loads the specified scenario definition file into Event Modeler. Repeat to load multiple scenario definition files.
`-XgenerateDebug [true|false]`	Generate debug output or not (default is true).
`-Xgenerate scenario EPL_file`	Generate EPL from the specified scenario definition file.
`-XgenerateBlock scenario block catalog`	Generate a block from the specified scenario definition file and save the new block in the specified catalog.
`-XforceBlockCatalogPaths path[,path ...]>`	Force Event Modeler to use the specified comma separated block catalog paths.
`-XaddBlockCatalogPaths path[,path ...]>`	Add the comma separated block catalog paths to Event Modeler.
`-XforceFunctionCatalogPaths path[,path ...]`	Force Event Modeler to use the specified comma separated function catalog paths.
`-XaddFunctionCatalogPaths path[,path ...]>`	Add the comma separated function catalog paths to Event Modeler.

Using Event Modeler

Chapter 3: Working with Blocks Created from Scenarios

- Terminology for using scenario blocks 85
- Benefits of scenario blocks 86
- Steps for using scenario blocks 86
- Background for using scenario blocks 86
- Saving scenarios as block templates 87
- Incrementing scenario block version numbers 87
- Adding a scenario block to a main scenario 88
- Examining a scenario block's source scenario 88
- Descriptions of scenario block parameters 88
- Descriptions of scenario block operations 89
- Descriptions of scenario block feeds 90
- Setting parameters before creating sub-scenarios 93
- Creating sub-scenarios 94
- Deleting sub-scenarios 95
- Modifying sub-scenario input variable values 96
- Iterating through sub-scenarios 96
- Obtaining variable values from sub-scenarios 98
- Linking sub-scenarios with other blocks 98
- Inheriting sub-scenarios 98
- Observing changes in sub-scenarios 100
- Performing simple calculations across sub-scenarios 102

In the Event Modeler, you can export a scenario to create a block. You can then use this block in other scenarios. This chapter provides information and instructions for using blocks that you create from scenarios.

For a sample scenario that uses a block that was created from a scenario, open the `ScenarioAsBlockExample.sdf` file in the Event Modeler. This file is in the `samples\scenarios` directory of your Apama installation directory.

You cannot create a block from a parallel-aware scenario. Nor can you create a block from a non-parallel-aware scenario and then mark that block as parallel-aware.

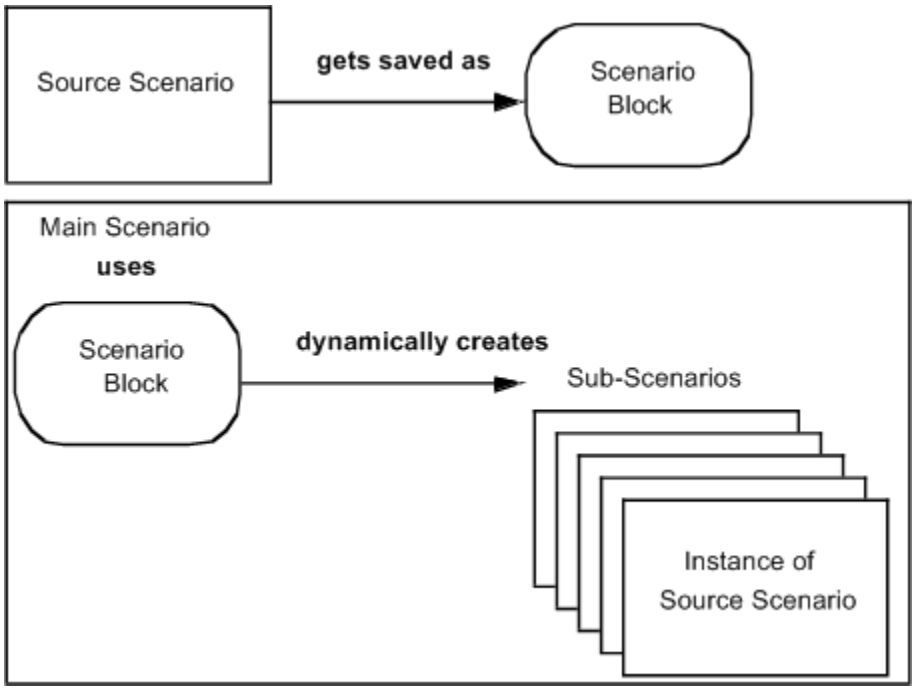
Terminology for using scenario blocks

To use blocks created from scenarios, you must understand the following terms:

- Source scenario — A scenario block that you export to create a block.
- Scenario block— A block that you create from a scenario by selecting Scenario > Generate Block in the Apama Studio menu and then saving and building the project. Alternatively, you can select Export as Block from File > Export dialog.
- Main scenario — A scenario that uses a scenario block.
- Sub-scenario — A source scenario instance that a scenario block dynamically creates. When you use a scenario block in a main scenario, the scenario block manages sub-scenarios according to the rules you define in the main scenario. The operations a scenario block can perform on a sub-scenario include `create`, `retrieve`, `commit`, `delete`, `delete all`, `iterate`, and `next`.
- Context instance — Also referred to as the context sub-scenario. This is the current sub-scenario. A scenario block can create any number of sub-scenarios. However, at any point in time, a main scenario can modify only the context instance. Certain operations make a particular sub-scenario the context instance. You can also set the value of the scenario block `instance id` parameter to the instance ID for a particular sub-scenario and then call the scenario block `retrieve` operation to make that sub-scenario the context instance.

The following figure shows the relationships among these items.

Figure 21. Relationship between source scenario and scenario block



Working with Blocks Created from Scenarios

Benefits of scenario blocks

The benefit of using a scenario block is that you can write a scenario once and then use it any number of times without having to manually create instances of that scenario. Instead, in your main scenario, you define rules that create and manage the instances of the source scenario. When a main scenario uses a scenario block, the scenario block dynamically creates and manages instances of the source scenario according to the rules you define in the main scenario. The main scenario functions as a management tool for the sub-scenarios. This allows self-contained units of work that start and finish within the main scenario.

A main scenario can use several different scenario blocks. This lets you define multiple source scenarios, and then pull them together into a single main scenario.

Like all blocks, using scenario blocks makes propagating updates to the source scenario easier. For example, suppose you have 10 instances of a scenario. If you need to change that scenario, you must also update the 10 instances. Now suppose you have a main scenario that uses a scenario block to create 10 sub-scenarios. If you need to modify the source scenario, you only need to also update the main scenario that uses the scenario block.

Working with Blocks Created from Scenarios

Steps for using scenario blocks

The general steps for using scenario blocks are as follows:

1. Define and save the source scenario.
2. Generate a block from the source scenario to create your scenario block. This makes your new scenario block available for selection in the Catalogs tab.
3. Define a main scenario.
4. Add your scenario block to your main scenario.
5. In your main scenario, define rules that refer to your scenario block.
6. Deploy the source scenario. You can do this in Apama Studio, or by injecting the `.sdf` file into the correlator with the `engine_inject` utility. If the source scenario requires any event types or other EPL to be injected before you can run it, be sure to inject those items before you try to run the main scenario.
7. Deploy the main scenario.

Working with Blocks Created from Scenarios

Background for using scenario blocks

To use scenario blocks in a main scenario, it is helpful to understand the implementation model. Consider a deck of cards with each card stacked on top of the other cards. Each card represents a sub-scenario, which is an instance of the source scenario.

When a sub-scenario generates an update event, that sub-scenario pops to the top of the stack of sub-scenarios, like you might move a card to the top of a deck. When a sub-scenario is at the top of the

stack of sub-scenarios, you can access the values associated with that sub-scenario. Any time you can access the values associated with a sub-scenario, that sub-scenario is the context sub-scenario. For example, when a sub-scenario completes its processing, the scenario block sends an update event to its `output` feed. This update event makes the completed sub-scenario the context instance. Consequently, you can do something like this:

```
When  instance status from MyScenarioBlock(output) = "ENDED"
Then  quantity = quantity + subquantity from MyScenarioBlock(output)
```

If `quantity` is a variable in the main scenario, this action increases the value of the `quantity` variable upon the completion of each sub-scenario. You do not need to first retrieve a sub-scenario to obtain the value of its `subquantity` variable.

As you can see, one way to operate on a particular sub-scenario is to wait for that sub-scenario to be the context sub-scenario. Another way to operate on a particular sub-scenario is to make that sub-scenario be the context sub-scenario. You do this by specifying the context ID of the sub-scenario you want to operate on and then calling the `retrieve` operation.

A main scenario can use two or more instances of the same scenario block. Each scenario block manages only the sub-scenarios it creates. However, you can change this according to the value you specify for the scenario block `inheritExternalInstances` parameter. See "Inheriting sub-scenarios" on page 98.

Working with Blocks Created from Scenarios

Saving scenarios as block templates

To use a source scenario as a block, you must save it as a block, which creates a new block template.

To create a block template from a scenario:

1. In Event Modeler, open the scenario from which you want to create a block.
2. Ensure that the scenario is complete and correct.
3. In the Event Modeler menu bar, select Scenario and ensure that there is a check next to Generate Block.

Whenever you save and/or build the project, Event Modeler generates a block template from this scenario. You can see the block template in the `Generated scenario blocks` catalog in your project's Catalogs tab. The name of the block template is the name of the scenario with the `.bdf` extension. If you have already saved a version of this scenario as a block, Event Modeler sets the version field to the revision level of the latest scenario block exported from this scenario.

Working with Blocks Created from Scenarios

Incrementing scenario block version numbers

To increment the version number, export one or more scenarios as blocks:

1. From the Apama Studio menu, select File > Export
2. In the **Export** dialog, expand Apama, click Export as Block, and click Next.
3. In the Project: field, select the project that contains the scenario(s) you want to export.
4. In the Export column, select one or more scenarios to export as blocks and click Next.

5. Select the folder in which you want to save your new block. By default, Event Modeler saves scenario blocks in the `catalogs\Generated scenario blocks` directory of your project directory.

   The name of the new block is always the name of the scenario with the `.bdf` extension. If you have already saved a version of this scenario as a block, Event Modeler sets the version field to the revision level of the latest scenario block exported from this scenario. To save a newer version, increment the version number.

   To create a new folder in which to store your new scenario block, click New..., specify the name of the new folder, and click OK.

   To add a new catalog in which to store your new scenario block, switch to Apama Developer perspective, right-click the project name, select Properties, and click the Blocks tab. Then return to the **Export As Block** dialog.

6. Click Export. Your new scenario block is immediately available for selection from the Catalogs tab.

You can nest a scenario block in another scenario block. In other words, you can export a main scenario as a block, and use the new scenario block in some other main scenario.

Working with Blocks Created from Scenarios

Adding a scenario block to a main scenario

You add a scenario block to a main scenario as you would add any other block to a scenario.

1. In the Catalogs tab, select the scenario block you want to use in your main scenario.
2. In the Catalogs tool bar, click the Add Selected Block Template button. The scenario block you selected now appears in the Blocks tab.

You can now use the scenario block as you would any other block.

Working with Blocks Created from Scenarios

Examining a scenario block's source scenario

After you add a scenario block to a main scenario, you might like to look at the scenario block's source scenario. To do this:

1. Select the scenario block in the Blocks tab.
2. Right-click to display the context menu.
3. Select Open Source Scenario....

This displays a separate copy of Event Modeler with the source scenario open.

Working with Blocks Created from Scenarios

Descriptions of scenario block parameters

A scenario block has the following parameters:

- `instance id` — This is a string that identifies a sub-scenario. An instance ID must be unique within a main scenario. In the main scenario, you set the value of the `instance id` parameter to indicate the sub-scenario that is the target of the next scenario block operation.
- `deleteChildrenOnTerminate` — Boolean that indicates whether all sub-scenarios terminate when the main scenario terminates. The default behavior is that sub-scenarios remain active if the main scenario terminates. That is, the default is false.

  If the main scenario inherits sub-scenarios from other main scenarios, the inherited sub-scenarios would also terminate when the value of the `deleteChildrenOnTerminate` parameter is true.
- `inheritExternalInstances` — Indicates whether the main scenario inherits sub-scenarios created by other main scenarios. When the main scenario inherits sub-scenarios, it means that the main scenario can operate on inherited sub-scenarios as though it had created those sub-scenarios. For details, see "Inheriting sub-scenarios" on page 98.
- *`input-variables`* — There is one parameter for each source scenario variable that is marked as input. For example, if the source scenario has a `quantity` input variable, then a scenario block created from that source scenario has a `quantity` parameter. The recommendation is that you mark a source scenario variable as input or output and not as both.

When you add a scenario block to a main scenario, the initial value of the `instance id` parameter is an empty string, `""`. When you call the `create` operation on a scenario block and the value of the `instance id` parameter is an empty string, the scenario block generates the ID that it assigns to the new sub-scenario. This ensures that the instance ID is unique within the main scenario. You can obtain the assigned instance ID from the scenario block `output` feed.

Generated instance IDs would look something like the following for a scenario block named `MyScenarioBlock`:

```
MyScenarioBlock1;1
MyScenarioBlock1;2
MyScenarioBlock1;3
```

and so on

When you want to specify the ID that the scenario block assigns to a new sub-scenario, set the value of the `instance id` parameter and then call the `create` operation. If you specify an instance ID that already exists, and call the `create` operation, the `create` operation fails.

Working with Blocks Created from Scenarios

Descriptions of scenario block operations

You can call the following operations on a scenario block:

- `create` — Creates a sub-scenario.
- `delete` — Deletes the sub-scenario identified by the value of the `instance id` parameter.
- `delete all` — Deletes all sub-scenarios that this scenario block manages. The sub-scenarios that a scenario block manages are the sub-scenarios that the scenario block created and has not yet deleted. A main scenario can use two or more instances of the same scenario block. Each scenario block manages only the sub-scenarios it creates. In a main scenario, the `A1` scenario block has no information about sub-scenarios created by the `A2` scenario block.

- `retrieve` — Retrieves the sub-scenario identified by the value of the `instance id` parameter. The retrieved sub-scenario becomes the context instance. To modify any values associated with a sub-scenario, the sub-scenario must be the context instance. The `retrieve` operation does not modify the current values of the scenario block's parameters.
- `commit` — Changes and saves the values of the context sub-scenario's input variables that correspond to scenario block parameters whose values have changed since the previous `create`, `iterate`, `next`, `retrieve`, or `commit` operation, whichever came last.
- `iterate` — Starts an iteration through the sub-scenarios that this scenario block manages. After you call the `iterate` operation, the first sub-scenario that the block created is the context sub-scenario. You do not need to call the `next` operation to retrieve the first sub-scenario. To restart an iteration, call the `iterate` operation again.
- `next` — Moves to the next sub-scenario in the iteration and makes that sub-scenario, if there is one, the context instance. The `next` operation visits the sub-scenarios in the order in which the scenario block created them.

  Call this operation after a call to the `iterate` operation. When you call `next`, if there is a valid next instance, the scenario block sends an event to the `output` feed. You can obtain the instance ID for the new context instance from this event.

  There are no timing issues because the scenario block immediately performs the `next` operation and sends an event to the `output` feed. That is, you do not need to wait for the `next` operation to complete before you issue an action that operates on the sub-scenario that is the context instance as a result of the `next` operation.

Working with Blocks Created from Scenarios

Descriptions of scenario block feeds

Scenario blocks have no input feeds. Scenario blocks have three output feeds:

- `output` — Provides updated information about a sub-scenario. The scenario block sends output to this feed whenever the value of a sub-scenario variable changes. The main scenario that created the sub-scenario, and any other main scenarios that inherit the sub-scenario each get an `output` feed to indicate the changes.
- `iteration ended`— Indicates whether an iteration is complete.
- `group info` — Provides cumulative information about all sub-scenarios managed by this scenario block.

The following table describes the fields in each output feed.

Feed	Fields	Description
`output`	`instance id`	String that identifies the sub-scenario that changed.
	`instance owner`	Identifies the user account under which the main scenario that is using this scenario block was created.
	`instance created`	Boolean value that is true after the sub-scenario is created.

Feed	Fields	Description
	`instance ended`	Boolean value that is true after the sub-scenario stops processing. This can happen because it fails, is deleted, or ends its normal processing.
	`instance status`	Enumerated string field that indicates the status of the sub-scenario. The value is one of the following: • `RUNNING` — The sub-scenario has been created and has not ended, failed, or been deleted. • `ENDED` — The sub-scenario has ended normally; it reached its end state. • `FAILED` — The scenario block failed to create the sub-scenario, perhaps because of a duplicate instance ID. Or, the sub-scenario failed because something went wrong while it was running. For example, the sub-scenario tried to divide by zero. • `DELETED` — The main scenario called the delete operation, which removes the sub-scenario from the correlator. Or, some other external entity deleted the sub-scenario from the correlator. • `UNKNOWN` — The status of the sub-scenario is unknown. For example, the status is unknown after you invoke the create operation and before the scenario block actually creates the sub-scenario.
	`variables`	In the `output` feed, there is a field for each source scenario variable. Each of these fields contains the current value of the variable for the identified sub-scenario.
`iteration ended`	`complete`	Boolean value that is true when iteration through the sub-scenarios that this scenario block manages is complete. When you call the `next` operation, and there is not another sub-scenario in the iteration, then the `iteration ended` feed outputs a value of true for the `complete` field.
`group info`	`total created`	Integer that indicates how many sub-scenarios this scenario block has created since it began processing.
	`total deleted`	Integer that indicates how many sub-scenarios this scenario block has deleted since it began processing.
	`total loaded`	Integer that indicates how many sub-scenarios created by this scenario block are loaded in the correlator. This includes sub-scenarios that are running, plus sub-scenarios that failed while they were running, plus sub-scenarios that have ended. This number does not include sub-scenarios that the scenario block tried to create and

Feed	Fields	Description
		failed to create. In other words, the total loaded is equal to the total created minus the total deleted.
	`number running`	Integer indicating how many sub-scenarios created by this scenario block are running.
	`number ended`	Integer indicating how many sub-scenarios created by this scenario block are still loaded but have ended.
	`number failed`	Integer that indicates how many sub-scenarios created by this scenario block are still loaded but have failed.
	`summary`	Convenience string that summarizes the information provided by the other `group info` fields. For example: `"Total Created: 100, Total Deleted: 40, Total Loaded: 60, Number Running: 10, Number Ended: 48, Number Failed: 2".`

Inheritance affects the totals in the `group info` feed as follows:

- `total created` indicates the number of sub-scenarios that were created and that the main scenario could operate on. This number only goes up. This number includes sub-scenarios created by this main scenario as well as inherited sub-scenarios created by other main scenarios.
- `total deleted` indicates the number of sub-scenarios that were deleted while the main scenario could operate on them. This number only goes up. This number includes sub-scenarios created by this main scenario as well as inherited sub-scenarios.
- `total loaded`, `total running`, `number ended`, and `number failed` indicate the number of sub-scenarios that are currently loaded in the correlator and that the main scenario can operate on. This number goes up and down.

For example, suppose `inheritExternalInstances` is set to `Owner` for `MainScenarioA`. Now suppose `MainScenarioB`, which has the same owner as `MainScenarioA`, creates a new sub-scenario. The `total created` field for `MainScenarioA` gets incremented by 1. Now suppose that `MainScenarioC`, which has a different owner, creates the same type of sub-scenario. The `total created` field for `MainScenarioA` would not get incremented.

Following is an example of an `output` feed. Suppose the source scenario defines the following variables:

- `SYMBOL` (Input)
- `SIDE` (Input)
- `PRICE` (Output)
- `QUANTITY SOLD` (Output)

The `output` feed would have the following fields:

```
instance id
instance owner
instance created
instance ended
instance status
SYMBOL
SIDE
```

```
PRICE
QUANTITY SOLD
```

Working with Blocks Created from Scenarios

Setting parameters before creating sub-scenarios

When you add a scenario block to a main scenario, the scenario block's parameters have default values according to their types. For example, the default value of a string parameter is an empty string (`""`).

After you add a scenario block to a main scenario, you can set initial values for the scenario block's parameters in the Blocks tab. However, it is important to understand that the values you set are initial values and not default values. During execution of a main scenario, if you want to change the value of a parameter, you must explicitly do so. After you modify the value of a parameter, if you require the parameter to have its initial value, you must explicitly set it to its initial value.

When you call the `create` operation, the newly created instance's input variables take their values from the current values of the corresponding scenario block parameters. The current values of the parameters might or might not be the initial values; if you modified a parameter value, the parameter has the last value that was assigned to it. If you then call the `create` operation, the scenario block assigns that last value to the sub-scenario's corresponding input variable.

To create a sub-scenario that has the initial parameter values for its input variables, do one of the following:

- If the main scenario has not made any changes to the scenario block's parameter values, call the `create` operation.
- If the main scenario has made changes to parameter values, explicitly specify the value of each parameter, and then call the `create` operation. This is the safest way to ensure that you create the sub-scenario with the values you want. A common mistake is to forget that you changed the value of a parameter in the course of some work. If you then create a new sub-scenario, it has the updated value of the parameter and not the initial value.

For example, consider the following set-up: `MyScenarioBlock` has three parameters that correspond to three input variables: `Input1`, `Input2`, and `Input3`. The initial value of each parameter is `blue`. The value of the `instance id` parameter is the empty string, which means that the scenario block generates the instance IDs for you. In a rule, you can set parameter values and create sub-scenarios as follows:

`When  true`
`Then  Input1 = green`
`Then  create [MyScenarioBlock]`
	Creates the `MyScenarioBlock1;1` instance. The values of the parameters and the values of the input variables in this instance are `green`, `blue`, and `blue`.

`Then  Input2 = purple` `Then  create [MyScenarioBlock]`	Creates the `MyScenarioBlock1;2` instance. The values of the parameters and the values of the input variables in this instance are `green`, `purple`, and `blue`.
`Then  Input3 = white` `Then  create [MyScenarioBlock]`	Creates the `MyScenarioBlock1;3` instance. The values of the parameters and the values of the input variables in this instance are `green`, `purple`, and `white`.

`Then instance id = MyScenarioBlock1;2` `Then retrieve [MyScenarioBlock]`	Makes the second created sub-scenario the context instance. The variables in this instance have the values `green`, `purple`, and `blue`. Note that this is not the same as the current parameter values, which are `green`, `purple`, and `white`. The `retrieve` operation does not modify the current values of the scenario block's parameters.
`Then Input2 = gold` `Then commit [MyScenarioBlock]`	After the `commit` operation, the values of this sub-scenario's input variables are `green`, `gold`, and `blue`. The values of the corresponding scenario block parameters are `green`, `gold`, and `white`. The `commit` operation modifies only the context instance. It does not modify any other sub-scenarios. The `commit` operation makes only those changes made since the `retrieve` operation. For example, it does not change the value of `Input3` to `white`.
`Then create [MyScenarioBlock]`	Creates the `MyScenarioBlock1;4` instance. The values of the input variables in this instance are `green`, `gold`, and `white`, which are the current values of the corresponding parameters.

Working with Blocks Created from Scenarios

Creating sub-scenarios

The scenario block `create` operation creates a new sub-scenario with the current values of the scenario block's input-variables parameters. A sub-scenario is an instance of the source scenario. Call this operation for each sub-scenario you want to create.

You can have any number of sub-scenarios running in parallel. You do not need to wait for one sub-scenario to complete processing before you create another sub-scenario. When you invoke the `create` operation, the scenario block immediately sends an update event to its `output` feed. The fields in this event have the following values:

- `instance id` — This field provides the instance ID of the sub-scenario being created. This is either the instance ID you specified as the value of the `instance id` parameter before you called the `create` operation, or it is the instance ID generated by the scenario block if the value of the `instance id` parameter was an empty string. For the format of a generated instance ID, see "Descriptions of scenario block parameters" on page 88.
- `instance created` — This field is false because the scenario block has not yet created the new sub-scenario.
- `instance ended` — This field is also false.
- `instance status` — This field has a value of `UNKNOWN` because, again, the scenario block has not yet created the new sub-scenario.

In addition, the `output` feed contains a field for each variable that the source scenario defines.

As soon as the scenario block actually creates the new sub-scenario, it sends another event to the `output` feed. This time, if creation was successful, the `instance created` field is true, and the `instance status` field is `RUNNING`. For example, you might want to do something like this:

```
State: Step 1
When  true
Then  create [MyScenarioBlock]
Then  move to state [Step 2]
State: Step 2
When  instance created from MyScenarioBlock (output)
Then  status = "Instance created successfully"
```

When the scenario block sends the first event after you invoke the `create` operation, that event indicates that the sub-scenario you are creating is the context sub-scenario. For example, to issue two orders in sequence you can specify the following:

```
State 1
When  true
Then  Symbol from MyScenarioBlock = "APMA"
Then  create [MyScenarioBlock]
Then  continue
When  instance status from MyScenarioBlock(output) = "ENDED"
Then  Quantity = Quantity + Quantity from MyScenarioBlock(output)
Then  Symbol from MyScenarioBlock = "MSFT"
Then  create [MyScenarioBlock]
Then  move to state [State 2]
State 2
When  instance status from MyScenarioBlock(output) = "ENDED"
               (Note that this now reflects the second sub-scenario created.)
Then  Quantity = Quantity + Quantity from MyScenarioBlock(output)
```

Alternatively, you can do it this way:

```
When  true
Then  Symbol from MyScenarioBlock = "APMA"
Then  create [MyScenarioBlock]
Then  Symbol from MyScenarioBlock = "MSFT"
Then  create [MyScenarioBlock]
Then  Symbol from MyScenarioBlock = "ORCL"
Then  create [MyScenarioBlock]
```

To operate on a sub-scenario that you just created, you must wait for the value of the `instance status` field to be `RUNNING`.

Working with Blocks Created from Scenarios

Deleting sub-scenarios

To delete a sub-scenario when it reaches its end state:

1. Check the `output` feed for a true value for the `instance ended` field.
2. Call the `delete` operation.

The output event that the scenario block sends to its output feed to indicate that the instance has finished processing also makes the completed instance the context instance. Consequently, you do not need to set the `instance id` parameter before you call the `delete` operation.

Working with Blocks Created from Scenarios

Unconditionally deleting a sub-scenario

To unconditionally delete a sub-scenario:

1. Set the `instance id` parameter to the instance ID of the sub-scenario you want to delete.
2. Call the `retrieve` operation.
3. Call the `delete` operation.

Deleting sub-scenarios

Deleting all sub-scenarios

To delete all sub-scenarios that this scenario block created but has not yet deleted:

1. Call the `delete all` operation.
2. Watch the `group info` feed's `total loaded` field for a value of `0`.

Deleting sub-scenarios

Modifying sub-scenario input variable values

To modify the value of one of a sub-scenario's input variables:

1. Set the `instance id` parameter to the instance ID of the sub-scenario whose input variable you want to change.
2. Call the `retrieve` operation so that the sub-scenario you want to modify is the context instance.
3. Set the value of the scenario block's parameter that corresponds to the input variable you want to change. You can do this for each input variable you want to change.
4. Call the `commit` operation to save your changes. This does the following:
   - Updates only the sub-scenario identified by the `instance id` parameter.
   - Updates each input variable that corresponds to a scenario block parameter that you modified since the `retrieve` operation.
   - Sends output to the `output` feed to indicate the current variable values.

Working with Blocks Created from Scenarios

Iterating through sub-scenarios

To iterate through the sub-scenarios that a particular scenario block manages, you can do something like the following:

1. In `State 1`, call the `iterate` operation to start an iteration. After you call `iterate`, the first sub-scenario that the block created becomes the context instance.
2. Move to `State 2`.
3. In `State 2`, determine whether you are done iterating through the sub-scenarios.
   a. If the value of the `complete` field in the `iteration ended` output feed is true, then you are done iterating. Move to `State 3`.
   b. If there are no sub-scenarios, the value of the `complete` field is true immediately after calling the `iterate` operation.
   c. If the value of the `complete` field in the `iteration ended` output feed is false, then you are not done iterating. Do the following:
      - Do something. For example, aggregate some quantity.
      - Call the `next` operation to make the next sub-scenario the context instance. The `iterate` operation visits the sub-scenarios in the order in which they were created.
      - Move to `State 2`.

Following are rules that perform these steps:

```
State 1
  When  true
  Then  iterate [MyScenarioBlock]
  Then  move to State 2
State 2
  When  complete from MyScenarioBlock(iteration ended)
  Then  move to State 3
  When  true
  Then  Quantity = Quantity + Quantity from MyScenarioBlock(output)
  Then  next [MyScenarioBlock]
  Then  Move to State 2
```

In your main scenario, you might want to start the iteration and perform the iteration in a single state. One way to do this is to use a Boolean variable that indicates whether an iteration is in progress. In the following example, `iterating` is a Boolean variable:

```
Iterate State
  When  not iterating
  Then  iterating = true
  Then  iterate [MyScenarioBlock]
  Then  continue
  When  complete from MyScenarioBlock(iteration ended)
  Then  iterating = false
  Then  Move to AnotherState
  When  true
  Then  Quantity = Quantity + Quantity from MyScenarioBlock(output)
  Then  next [MyScenarioBlock]
  Then  Move to Iterate State
```

There is no significant performance advantage of using one of the above iteration techniques rather than the other. Choose the simplest approach for your Scenario. To restart an iteration, call the `iterate` operation.

Note: You might find it convenient to use the Filtered Summary block instead of an iteration. The Filtered Summary block can calculate totals and averages across sub-scenarios. For any other calculations, you would need to iterate through sub-scenarios. See the "Filtered Summary v2.0" on page 161 for details.

Working with Blocks Created from Scenarios

Obtaining variable values from sub-scenarios

Because a sub-scenario is an instance of its source scenario, each sub-scenario contains the variables defined in its source scenario. To obtain the current value of a sub-scenario's variable, check the scenario block's `output` feed. The `output` feed contains a field for each source scenario variable. The scenario block updates its `output` feed whenever there is a change to the value of a sub-scenario variable.

Working with Blocks Created from Scenarios

Linking sub-scenarios with other blocks

You can share sub-scenario instance IDs with other blocks. For example, the `Wait` block supports multiple concurrent timers. You could assign an ID to each timer and then use that same ID to create a sub-scenario. You could do this multiple times. When a timer fires, you can use the ID it reports to retrieve the associated sub-scenario and perform some operation on it, such as deleting it. For example:

```
When  time up from Wait (timer)
Then  instance id from MyScenarioBlock = timer id from Wait (timer)
Then  retrieve [MyScenarioBlock]
Then  continue
When  instance status from MyScenarioBlock (output) is equal to "RUNNING"
Then  move to state[next]
```

Working with Blocks Created from Scenarios

Inheriting sub-scenarios

A scenario block has the `inheritExternalInstances` parameter, which indicates whether the main scenario inherits sub-scenarios created by other main scenarios. Inherited sub-scenarios are always

- Loaded in the correlator
- Created by the same type of scenario block as the scenario block for which you are setting the parameter.

When the main scenario inherits sub-scenarios, it means that the main scenario can operate on inherited sub-scenarios as though it had created those sub-scenarios. For example, if the main scenario iterates over its sub-scenarios, the iteration includes inherited sub-scenarios.

Working with Blocks Created from Scenarios

Description of inheritExternalInstances values

The `inheritExternalInstances` parameter has one of the following values:

- `None` — The main scenario can operate on only the sub-scenarios it creates. This is the default.

- `Owner` — The main scenario can operate on sub-scenarios that have the same owner as the main scenario.

  Every main scenario is created under a particular user account. This account is the owner of the main scenario and consequently it is also the owner of each sub-scenario that the main scenario creates. Each scenario block has an `instanceowner` output field that indicates the owner.

- `All` — The main scenario can operate on all sub-scenarios created by scenario blocks that are the same type as the scenario block for which you are setting the `inheritExternalInstances` parameter. It does not matter which main scenario created the sub-scenario or which account owns the sub-scenario.

Inheriting sub-scenarios

Notes for setting the inheritExternalInstances parameter

You can change the value of the `inheritExternalInstances` parameter during Scenario execution. When you do, the new value takes effect immediately. Likewise, as other main scenarios create sub-scenarios, a main scenario might inherit those sub-scenarios if it has a value of `Owner` or `All` for its `inheritExternalInstances` parameter.

When a main scenario changes the value of the `inheritExternalInstances` parameter, the scenario block searches within the correlator for sub-scenarios that the main scenario now inherits. For each sub-scenario that the scenario block finds, it sends data to its `output` feed. For example, if the scenario block finds five sub-scenarios that the main scenario now inherits, the scenario block sends five sets of data to its `output` feed. The scenario block also sends data to its `group info` feed that includes the inherited sub-scenarios in the counts. Subsequently, if any main scenarios create or terminate sub-scenarios that another main scenario inherits, or if any inherited sub-scenarios fail, the scenario block in the inheriting main scenario sends data to its `output` feed just as if the inheriting main scenario had created the sub-scenario.

A particular main scenario does not need to create any sub-scenarios before it can inherit sub-scenarios created by other main scenarios. For example, you might define a scenario block whose only purpose is to monitor inherited sub-scenarios and perform some sort of aggregation or analysis. Or, you can define a scenario block with true as the value of the `deleteChildrenOnTerminate` parameter. When you want to terminate all instances of that type of sub-scenario you need to only terminate one main scenario.

Keep in mind that inherited sub-scenarios are shared by more than one main scenario. That means that more than one main scenario can operate on the same sub-scenario. Be sure to consider this when you design your application.

Inheriting sub-scenarios

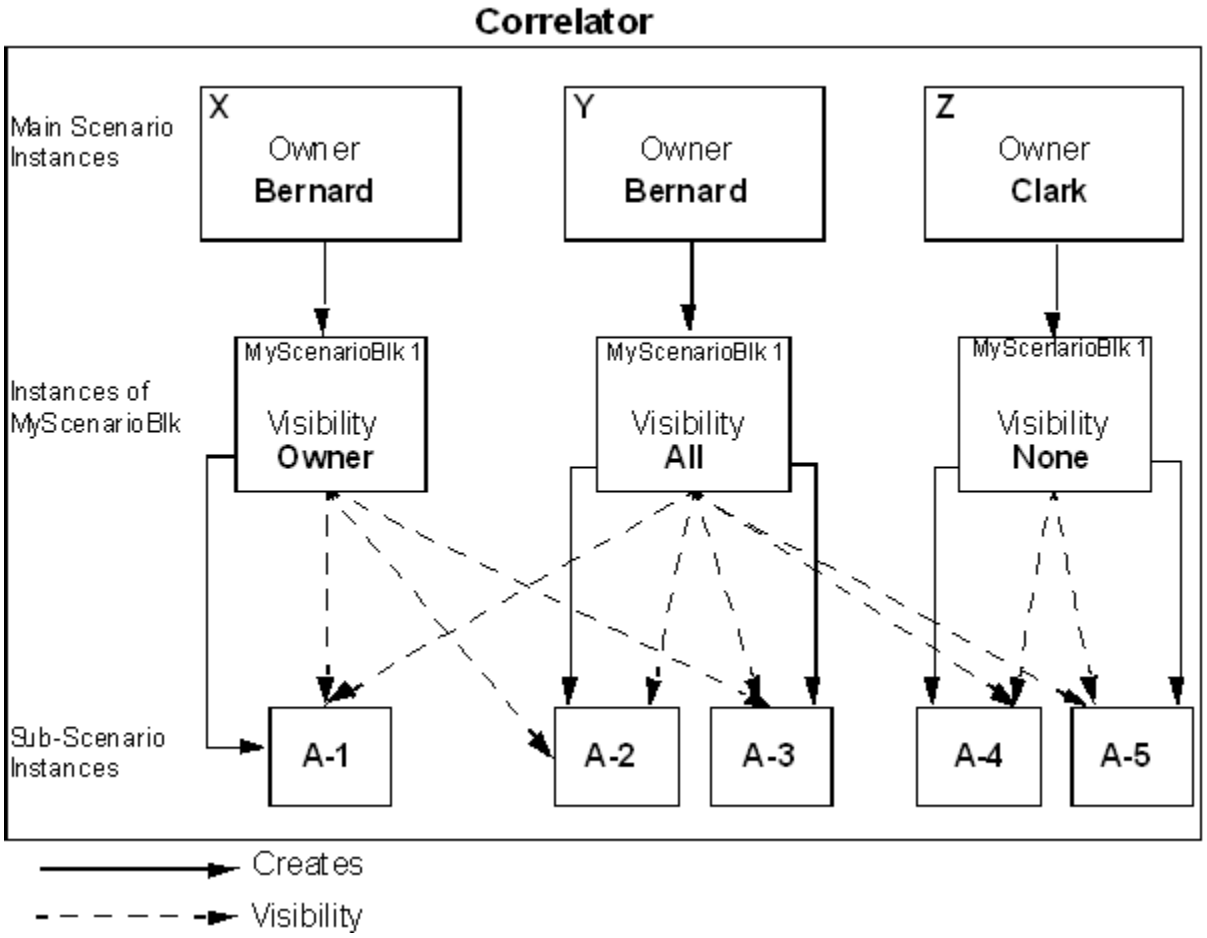
Example of inheriting sub-scenarios

The following figure illustrates how the `inheritExternalInstances` parameter works. Each main scenario is owned by the user account under which it was created. When a main scenario inherits a sub-scenario, the inherited sub-scenario is visible to the main scenario.

Remember that inherited sub-scenarios are always of the same type as the scenario block for which you are setting the `inheritExternalInstances` parameter. In the following figure, the scenario blocks are each shown as `MyScenarioBlk 1`. They could of course have been shown as `MyScenarioBlk 2`, `MyScenarioBlk 3`, and `MyScenarioBlk 5`, or any other similar combination. The important point is that they are all instances of `MyScenarioBlk`. In the figure,

- Main scenario `x` can operate on sub-scenarios `A-1`, `A-2`, and `A-3`.
- Main scenario `y` can operate on sub-scenarios `A-1`, `A-2`, `A-3`, `A-4`, and `A-5`.
- Main scenario `z` can operate on sub-scenarios `A-4` and `A-5`.

Figure 22. Scenario inheritance



Inheriting sub-scenarios

Observing changes in sub-scenarios

The Change Observer block watches a set of sub-scenarios for changes in the value of one of the sub-scenario variables. You specify which variable you want to watch. When the value changes, the Change Observer block sends data to its `change` output feed. The output feed indicates the old value and the new value. You use one Change Observer block for each variable that you want to observe. See "Change Observer v2.0" on page 159 for details.

For example, suppose your main scenario uses the Trader scenario block and the Price Checker scenario block. The Trader scenario block output fields include:

- `instance id [string]`
- `instance owner [string]`
- `instance created [Boolean]`
- `instance ended [Boolean]`
- `instance status [UNKNOWN, RUNNING, ENDED or FAILED]`
- `trading [Boolean]`

The Price Checker scenario block output fields include the following:

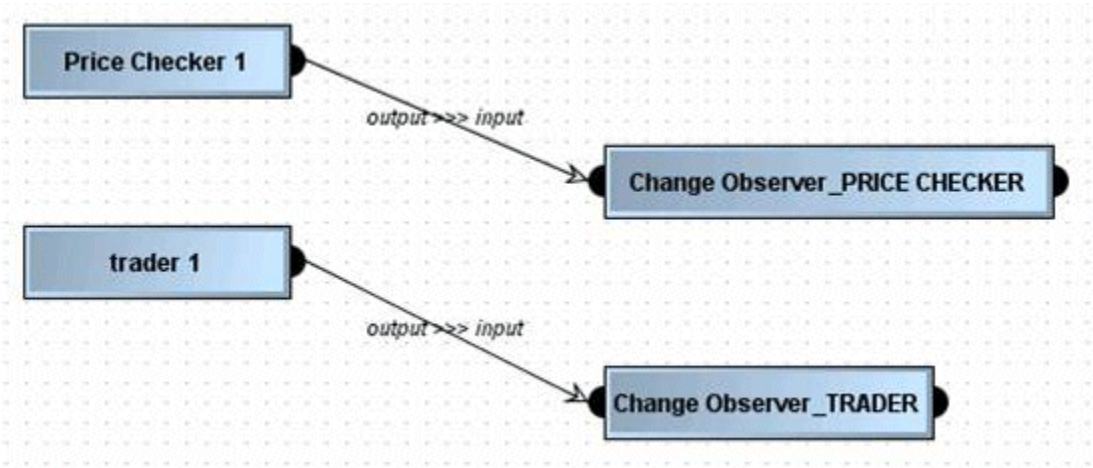
- `instance id [string]`
- `instance owner [string]`
- `instance created [Boolean]`
- `instance ended [Boolean]`
- `instance status [UNKNOWN, RUNNING, ENDED or FAILED]`
- `price [number]`

In your main scenario, you create several Trader sub-scenarios — each one trades in a different market. When a Trader sub-scenario finishes trading, it sends data to its output feed and this data includes `trading=false`.

You also create several Price Checker sub-scenarios — one for each type of stock symbol you are trading. When the price of a stock being checked changes, the Price Checker sub-scenario sends data to its output feed and this data includes the new price.

In your main scenario, you want to monitor changes in the Trader `trading` field and in the Price Checker `price` field. To do this, use an instance of the Change Observer block for each field. The block wiring would look like this:

Figure 23. Block wiring example

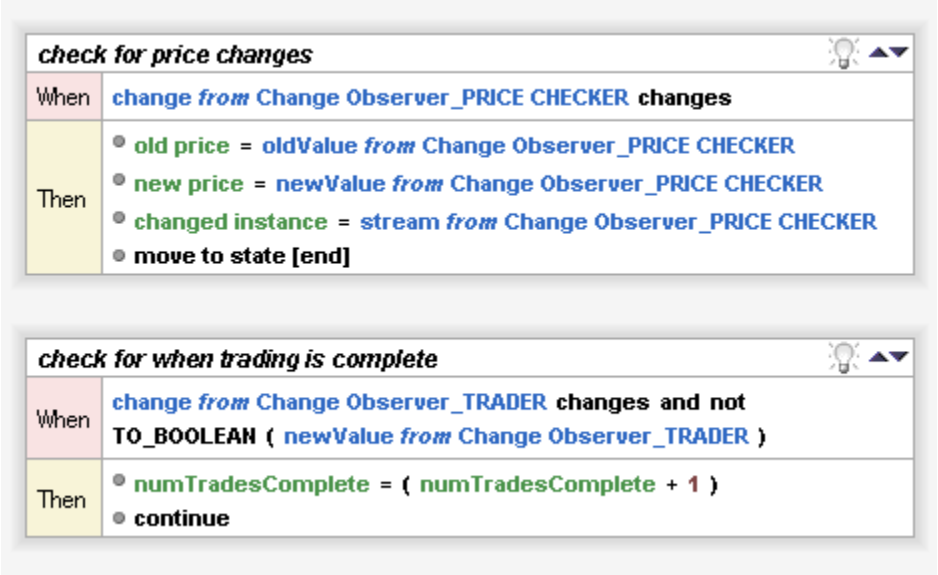


The Change Observer_PRICE CHECKER block sends an output feed whenever a Price Checker sub-scenario sends a price change to its output feed. The Change Observer_TRADER block sends an output feed whenever a Trader sub-scenario stops trading or starts trading, as indicated by the `trading` field in its output feed. You would wire their fields as follows:

Wire Price Checker 1 Output Feed	To Change Observer_PRICE CHECKER Input Feed
Output feed: `output`	Input feed: `input`
`instance id` [`string`] output field	`stream` [`string`] input field
`price` [`float`] output field	`watchValue` [`string`] input field
Wire Trader 1 Output Feed	**To Change Observer_TRADER Input Feed**
Output feed: `output`	Input feed: `input`
`instance id` [`string`] output field	`stream` [`string`] input field
`trading` [`boolean`] output field	`watchValue` [`string`] input field

The rules to implement this would look something like the following: (Note that the name of the Change Observer block output feed is `change`.

Figure 24. Sample rules



Working with Blocks Created from Scenarios

Performing simple calculations across sub-scenarios

The Filtered Summary block performs simple calculations across a set of sub-scenarios. This is an alternative to iterating over a set of sub-scenarios. The Filtered Summary block can operate on only

floating point values. You can use this block to calculate sums and averages. See "Filtered Summary v2.0" on page 161 for details.

In more general terms, the Filtered Summary block performs calculations on a keyed set of floating point values. Typically, you use the sub-scenario instance ID as the key. The key's associated value is the value of a sub-scenario floating point variable that you want to use in an aggregate calculation.

To use the Filtered Summary block, wire output fields from the scenario block to input fields of the Filtered Summary block. Typically, you want to map the scenario block `instance id` output field to the Filtered Summary `key` input field. Then map a floating point sub-scenario variable from the scenario block `output` feed to the Filtered Summary `value` input field.

You can specify filters to perform calculations on a sub-group of sub-scenarios. For example, suppose you wanted to calculate the total number of shares purchased by sub-scenarios owned by `John`. To accomplish this, you do the following two things:

- Map the scenario block `instance owner` output field to the Filtered Summary block `filter` input field.
- Set the Filtered Summary `filter` parameter to `"John"`.

When the Filtered Summary block receives input from your scenario block, it checks whether the value of the `filter` input field is equal to the value of the `filter` parameter. If the values are equal, (in the example, they are both `"John"`) the Filtered Summary block sends output to its output feed. If the values are not equal, the Filtered Summary block sends no output.

Now suppose that you want to exclude shares purchased by John from your calculation. That is, you want to know the total number of shares purchased by everyone except John. To make this happen, you perform one step in addition to the steps already described. Set the Filtered Summary block's `filter is "not equal to"` parameter to true. Now the Filtered Summary block sends output only when the `filter` input field is not equal to `"John"`.

You can also remove keys and their associated values from the Filtered Summary block's internal data store. This lets you exclude data from certain sub-scenarios from the calculations. You do this with the `deleteKey` operation and the `keyToDelete` parameter. One way to do this is to define a global rule that watches for sub-scenarios to terminate. When a sub-scenario terminates, you can specify its instance ID as the key and remove the data for that key from the Filtered Summary block's store of data.

Working with Blocks Created from Scenarios

Chapter 4: Using Functions in Event Modeler

- Reference information for provided functions 104
- About defining your own functions 116

In Event Modeler, when you define a rule, you can use a function to specify the value, or part of the value, of a condition or action. Event Modeler provides a number of functions that you can use. In addition, you can define your own functions.

To use a function in a rule, select Standard Functions from the context menu when defining a rule. Event Modeler displays only those functions that are valid for the portion of the rule you are defining.

Reference information for provided functions

Event Modeler provides a number of functions. Each function is defined in its own function definition file (`.fdf` file) in the `catalogs/functions` directory of the your Apama installation directory. A function definition file is an XML file that contains metadata about the function plus the EPL that implements the function.

The following tables describe the functions provided in Event Modeler. Your Apama Service Provider might have included additional functions that are not documented here.

- "Date and time functions" on page 104
- "IO functions" on page 108
- "System value functions" on page 109
- "Miscellaneous functions" on page 110

Using Functions in Event Modeler

Date and time functions

The following table describes the date and time functions.

Typical use

A typical use of most of these functions is something like the following:

```
ADD_YEAR(GET_CURRENT_TIME_AS_NUMBER(), 5)
```

Table 5. Date and time functions

Function name	Return value	Parameters	Description
ADD_DAYS	float	float *dateTime* float *nrDays*	Given a date plus a number of days, returns the result date in seconds since the epoch.
ADD_HOURS	float	float *dateTime* float *nrHours*	Given a date plus a number of hours, returns the result date in seconds since the epoch.
ADD_MINUTES	float	float *dateTime* float *nrMins*	Given a date plus a number of minutes, returns the result date in seconds since the epoch.
ADD_MONTHS	float	float *dateTime* float *nrMonths*	Given a date plus a number of months, returns the result date in seconds since the epoch.
ADD_WEEKS	float	float *dateTime* float *nrWeeks*	Given a date plus a number of weeks, returns the result date in seconds since the epoch.
ADD_YEARS	float	float *dateTime* float *nrYears*	Given a date plus a number of years, returns the result date in seconds since the epoch.
FORMAT_TIME	string	float *TimeInSeconds* string *TimeFormat*	Returns the specified time and date in a formatted string. For example: `FORMAT_TIME(GET_CURRENT_TIME(), "dd-MM-yyyy HH:mm:ss")` For format options, see "Using the Time Format plug-in" in *Developing Apama Applicatons in EPL*.
GET_CURRENT_DATE	string	none	Returns the current date in a formatted string. For example, `"11 June 2007"`.
GET_CURRENT_DATE_TIME	string	none	Returns the current date and time in a formatted string. For example, `"11 June 2007 11:10:23"`.
GET_CURRENT_TIME	string	none	Returns the current time in a formatted string. For example, `"11:10:25"`.
GET_CURRENT_TIME_AS_NUMBER	float	none	Returns the current time as a number of seconds since the epoch, January 1, 1970.

Function name	Return value	Parameters	Description
`GET_CURRENT_TIME _FORMATTED`	`string`	`string` *`TimeFormat`*	Returns the current time and date in a formatted string. For format options, see "Using the Time Format plug-in" in *Developing Apama Applicatons in EPL*.
`GET_DAY_IN_WEEK`	`float`	`float` *`dateTime`*	Returns the day of the week for the given date.
`GET_DAY_IN_YEAR`	`float`	`float` *`dateTime`*	Returns the day in the year for the given date.
`GET_MONTH_IN_YEAR`	`float`	`float` *`dateTime`*	Returns the month in the year for the given date.
`GET_WEEK_IN_MONTH`	`float`	`float` *`dateTime`*	Returns the week in the month for the given date.
`GET_WEEK_IN_YEAR`	`float`	`float` *`dateTime`*	Returns the week in the year for the given date.
`IS_LEAP_YEAR`	`boolean`	`float` *`year`*	Returns true if the given year is a leap year.
`PARSE_TIME`	`float`	`string` *`TimeDate`* `string` *`TimeFormat`*	Returns the specified time and date in a numeric format. For example: `PARSE_TIME (GET_CURRENT_TIME(), "H:m:s")` For format options, see "Using the Time Format plug-in" in *Developing Apama Applicatons in EPL*.

Reference information for provided functions

Extended math functions on float types

The following table describes the extended math functions on `float` types.

Table 6. Extended math functions on **`float`**types

Function name	Return value	Parameters	Description
`ACOS`	`float`	`float` *`value`*	Returns the inverse cosine of the value in radians. If the value's absolute value is greater than `1` then `ACOS()` returns `NaN`.

Function name	Return value	Parameters	Description
ACOSH	float	float *value*	Returns the inverse hyperbolic cosine of the value. If the value's absolute value is less than 1 then ACOSH() returns NaN.
ASIN	float	float *value*	Returns the inverse sine of the value in radians. If the value is NaN then ASIN() returns the value. If the value's absolute value is greater than 1 then ASIN() returns NaN.
ASINH	float	float *value*	Returns the inverse hyperbolic sine of the value.
ATAN	float	float *value*	Returns the inverse tangent of the value.
ATAN2	float	float *x* float *y*	Returns the two-parameter inverse tangent of the two values.
ATANH	float	float *value*	Returns the inverse hyperbolic tangent of the value.
CBRT	float	float *value*	Returns the cube root of the value.
COS	float	float *value*	Returns the cosine of the value. The value should be in units of radians.
COSH	float	float *value*	Returns the hyperbolic cosine of the value.
ERF	float	float *value*	Returns the error function value for the given value.
EXPONENT	float	float *value*	Returns the exponent where the given value is equal to $mantissa*2^{exponent}$, assuming 0.5 <= |mantissa| < 1.0.
FMOD	float	float *nominator* float *denominator*	Returns *nominator* mod *denominator* in exact arithmetic.
FRACTIONALPART	float	float *value*	Returns the fractional component of the value.

Function name	Return value	Parameters	Description
GAMMAL	float	float *value*	Returns the logarithm of the gamma function.
ILOGB	integer	float *value*	Returns the binary exponent of the specified non-zero value.
INTEGRALPART	integer	float *value*	Returns the integeral part of a floating pont value.
MANTISSA	float	float *value*	Returns the mantissa where the given value is equal to $mantissa*2^{exponent}$, assuming `0.5 <= |mantissa| < 1.0`.
NEXTAFTER	float	float *x* float *y*	Returns the next machine floating point number after *x* in the direction toward *y*.
SCALBN	float	float *x* integer *n*	Returns *x*2n*.
SIN	float	float *value*	Returns the sine of the specified value, which should be in units of radians.
SINH	float	float *value*	Returns the hyperbolic sine of the value.
TAN	float	float *value*	Returns the tan of the value, which should be in units of radians.
TANH	float	float *value*	Returns the hyperbolic tangent of the value.

Reference information for provided functions

IO functions

The following table describes the IO functions.

Table 7. IO functions

Function name	Return value	Parameters	Description
LOG	string	string *message*	Logs the specified string to the correlator log.

Function name	Return value	Parameters	Description
		`string logLevel`	
`PRINT`	`string`	`string message`	Displays the specified string in the correlator console.

Reference information for provided functions

System value functions

The following table describes the system value functions.

Table 8. System value functions

Function name	Return value	Parameters	Description
`GET_DASHBOARD_ INSTANCEID`	`string`	None	Returns the instance ID of the current Scenario instance for use in dashboards. The `apama.instanceId` field contains this value.
`GET_INSTANCEID`	`string`	None	Returns the complete instance ID of the current Scenario instance. For example: `"default.myScenario.1"`.
`GET_INSTANCE_ OWNER`	`string`	None	Returns the value of the `owner` attribute of the current Scenario instance. This might be, but is not necessarily, the account Id that created the Scenario. You can use the Scenario service API to create Scenario instances and set the `owner` attribute to a value you choose. When you use a dashboard to create Scenario instances, the `owner` attribute has the value of the account you logged into.
`GET_NUMERIC_ INSTANCEID`	`float`	None	Returns only the number at the end of the complete instance Id of the current Scenario instance. For example, if the complete instance Id is `default.myScenario.1`, this function returns `1`.
`GET_SCENARIO_ID`	`string`	None	Returns the unique scenario ID of the current scenario definition. The correlator uses this key to create new instances of the scenario.
`GET_SCENARIO_ NAME`	`string`	None	Returns the display name of the current Scenario.

Reference information for provided functions

Miscellaneous functions

The following table describes the miscellaneous functions.

Table 9. Miscellaneous functions

Function name	Return value	Parameters	Description
ABS	number	number *value*	Returns the absolute value of the number supplied.
ADD_EXTRAPARAM	text	text *payload*, text*fieldname*, text*value*	This function is deprecated. Use the DICT_SET function instead. Takes an existing extraParam value and adds the specified field and value to it.
CEIL	number (whole number)	number *value*	Returns the ceiling integer value of the number passed. This is the smallest possible integer that is larger than the value supplied.
CONCAT	text	text *prefix*, text *suffix*	Concatenates two strings and returns the result as a string.
CONCAT	text	text *prefix*, choice *suffix*	Concatenates an enumeration value to a string, and returns the result as a string.
CONCAT	text	text *prefix*, number *suffix*	Concatenates a number to a string, and returns a string.
CONDITIONAL	text	condition *condition* text *true_result* text *false_result*	Functions like an IF statement. The first parameter is the expression to be evaluated, similar to a condition in an IF statement. The second and third parameters are the values to return according to the result of the condition. The second parameter represents a true result. The third parameter represents a false result. See

Function name	Return value	Parameters	Description
			"Example of CONDITIONAL function" on page 113.
DICT_GET	text	text *dictAsString* text *key*	Reads the dictionary specified by *dictAsString* and returns the value of the specified *key*. Specify the dictionary in dictionary<string,string>.toString() format. Returns an empty string if the key is not present or the string representation of the dictionary is "".
DICT_GETORDEFAULT	text	text *dictAsString* text *key* text *default*	Reads the dictionary specified by *dictAsString* and returns the value of the specified *key*. Specify the dictionary in dictionary<string,string>.toString() format. Returns the specified *default* text if the key is not present or the string representation of the dictionary is "".
DICT_HASKEY	boolean	text *dictAsString* text *key*	Reads the dictionary specified by *dictAsString* and returns true if the specified *key* exists in that dictionary. Specify the dictionary in dictionary<string,string>.toString() format..
DICT_SET	text	text *dictAsString* text *key* text *value*	Reads the dictionary specified by *dictAsString* and adds or replaces the specified *key*/*value* pair. Specify the dictionary in dictionary<string,string>.toString() format. An empty string for *dictAsString* is treated as an empty dictionary. Returns a string representation of the dictionary.

Function name	Return value	Parameters	Description
FLOOR	number (whole number)	number *value*	Returns the floor integer value of the number passed. This is the largest possible integer that is smaller than the value supplied.
GET_EXTRAPARAM	text	text *payload*, text *fieldname*	This function is deprecated. Use the DICT_GET function instead. Returns the value from extraParam data of the specified field, else an empty string.
HAS_EXTRAPARAM	boolean	text *payload*, text *fieldname*	This function is deprecated. Use the DICT_HASKEY function instead. Returns true if the extraParam data has the specified field value.
ISFINITE	boolean	float *value*	Returns true if *value* is finite, that is, it is not infinite or NaN.
ISINFINITE	boolean	float *value*	Returns true if *value* is infinite, that is, it is positive or negative infinity.
ISNAN	boolean	float *value*	Returns true if *value* is NaN, that is, it is not a number.
MAX	number	number *value1*, number *value2*	Returns the largest of two numbers.
MIN	number	number *value1*, number *value2*	Returns the smallest of two numbers.
POW	number	number *value*, number *exponent*	Returns the value of the first parameter to the power of the second parameter.
REPLACE	text	text *value*, text *old*, text *new*	Replaces all string occurrences of *old* in *value* with *new*.
RND	number	number *lower bound*, number *upper bound*	Returns a random number between the specified boundaries.

Function name	Return value	Parameters	Description
ROOT	number	number *value*, number *exponent*	Returns the value of the first parameter root of the second parameter.
ROUND	number	number *value*, number *decimal_places*	Rounds a float to a given number of decimal places. You can specify a negative number for *decimal_places* to round in the opposite direction. See "Example of ROUND function" on page 113.
TO_BOOLEAN	condition	text *value*	Converts a string to a Boolean value, and returns the Boolean value. This function is case insensitive.
TO_NUMBER	number	choice *value*	Converts an enumeration to a number, and returns the number.
TO_NUMBER	number	text *value*	Converts a string to a number, and returns the number.
TO_TEXT	text	condition *value*	Converts a Boolean value to a string, and returns the string.
TO_TEXT	text	number *value*	Converts a number to a string, and returns the string.

Example of CONDITIONAL function

```
side = CONDITIONAL (price is greater than 50, "BUY", "SELL")
```

If the price is greater than 50, this function returns `"BUY"`. The `side` Scenario variable is set to `BUY` or `SELL` according to whether the price variable is greater than 50.

Example of ROUND function

You can specify a negative number to round in the opposite direction. For example:

Value	Decimal places	Result
12345.6543	4	12345.6543
12345.6543	3	12345.654
12345.6543	2	12345.65
12345.6543	1	12345.7

Value	Decimal places	Result
12345.6543	0	12346.0
12345.6543	-1	12350.0
12345.6543	-2	12300.0
12345.6543	-3	12000.0
12345.6543	-4	10000.0
12345.6543	-5	0.0

Reference information for provided functions

Extended math functions on float types

The following table describes the extended math functions on `float` types.

Table 10. Extended math functions on **`float`**types

Function name	Return value	Parameters	Description
`ACOS`	`float`	`float` *`value`*	Returns the inverse cosine of the value in radians. If the value's absolute value is greater than `1` then `ACOS()` returns `NaN`.
`ACOSH`	`float`	`float` *`value`*	Returns the inverse hyperbolic cosine of the value. If the value's absolute value is less than `1` then `ACOSH()` returns `NaN`.
`ASIN`	`float`	`float` *`value`*	Returns the inverse sine of the value in radians. If the value is `NaN` then `ASIN()` returns the value. If the value's absolute value is greater than `1` then `ASIN()` returns `NaN`.
`ASINH`	`float`	`float` *`value`*	Returns the inverse hyperbolic sine of the value.
`ATAN`	`float`	`float` *`value`*	Returns the inverse tangent of the value.
`ATAN2`	`float`	`float` *`x`* `float` *`y`*	Returns the two-parameter inverse tangent of the two values.

Function name	Return value	Parameters	Description
ATANH	float	float *value*	Returns the inverse hyperbolic tangent of the value.
CBRT	float	float *value*	Returns the cube root of the value.
COS	float	float *value*	Returns the cosine of the value. The value should be in units of radians.
COSH	float	float *value*	Returns the hyperbolic cosine of the value.
ERF	float	float *value*	Returns the error function value for the given value.
EXPONENT	float	float *value*	Returns the exponent where the given value is equal to mantissa*2^exponent, assuming 0.5 <= |mantissa| < 1.0.
FMOD	float	float *nominator* float *denominator*	Returns *nominator* mod *denominator* in exact arithmetic.
FRACTIONALPART	float	float *value*	Returns the fractional component of the value.
GAMMAL	float	float *value*	Returns the logarithm of the gamma function.
ILOGB	integer	float *value*	Returns the binary exponent of the specified non-zero value.
INTEGRALPART	integer	float *value*	Returns the integeral part of a floating pont value.
MANTISSA	float	float *value*	Returns the mantissa where the given value is equal to mantissa*2^exponent, assuming 0.5 <= |mantissa| < 1.0.
NEXTAFTER	float	float *x* float *y*	Returns the next machine floating point number after *x* in the direction toward *y*.
SCALBN	float	float *x* integer *n*	Returns *x**2*n*.

Function name	Return value	Parameters	Description
SIN	float	float *value*	Returns the sine of the specified value, which should be in units of radians.
SINH	float	float *value*	Returns the hyperbolic sine of the value.
TAN	float	float *value*	Returns the tan of the value, which should be in units of radians.
TANH	float	float *value*	Returns the hyperbolic tangent of the value.

Reference information for provided functions

About defining your own functions

You define a function in Apama Studio. In the Apama Developer perspective, select File > New > Scenario Function . Apama Studio prompts you for some metadata and then creates a skeleton function definition file (`.fdf`), which is an XML file. The skeleton file indicates where you need to add data and what kind of data you need to add.

See *Using Apama Studio*, "Creating new scenario functions" for details about the scenario function definition file format.

The content of a function definition file must comply with the DTD in the `etc/fdf.dtd` file in the Apama installation directory.

The following topics provide additional information about using functions that you define in Event Modeler.

- "Sample ABS function definition file" on page 116
- "Sample function definition file with imports element" on page 117
- "About function names" on page 118

Related Topic

"Adding a function catalog" on page 68

Using Functions in Event Modeler

Sample ABS function definition file

Following is the function definition file for the absolute value (`ABS`) function. This function returns the absolute value of the given parameter. For example, if the input is `-123`, the `ABS` function returns `123`.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE function SYSTEM "http://www.apama.com/dtd/fdf.dtd">
<!--Apama Function Definition File-->
<function name="ABS" display-string="ABS" return-type="float">
  <version>
    <id>1.0</id>
    <date>22 Nov 2004</date>
    <author>Matthew Amos</author>
    <comments>External function</comments>
  </version>
  <description>
    Return the abs value of the number passed
  </description>
  <parameters>
    <fixed-parameter name="value" type="float" />
  </parameters>
  <code><![CDATA[
    action #name#(float f) returns float {
      return f.abs();
    }
  ]]></code>
</function>
```

Notes

Notes for this function:

- The value of the `function name` attribute, ABS, is unique within the directory that contains this `.fdf` file.
- Appears as ABS in the Event Modeler rules menu.
- Returns a float.
- Metadata indicates who wrote the function and when the function was written.
- Description briefly describes what the function does.
- There is one parameter called `value` and it is of type `float`.
- Name of the single action is the placeholder `#name#`. This is always what you specify as the name of the function in the `code` element.
- The EPL in the `CDATA` section is standard EPL. You can use locally defined variables in addition to the function's parameters. To use a Scenario variable, assign its value to a function parameter.

About defining your own functions

Sample function definition file with imports element

Following is a function definition file that specifies the `imports` element.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE function SYSTEM "http://www.apama.com/dtd/fdf.dtd">
<!--Apama Function Definition File-->
<function name="ExtractTimeField" display-string="ExtractTimeField"
    return-type="float">
  <version>
```

```
    <id>1.0</id>
    <date>17 May 2005</date>
    <author>Ben Spiller</author>
    <comments>External function</comments>
  </version>
  <description>
    Return the value of a single field from the specified
    time string (using the TimeFormatPlugin). Date fields
    include 'dd', 'MM' and 'yyyy'. Time fields include 'HH',
    'mm' and 'ss'.
  </description>
  <imports>
    <import library="TimeFormatPlugin" alias="timePlugin"/>
  </imports>
  <parameters>
    <fixed-parameter name="time" type="float" />
    <fixed-parameter name="field identifier" type="string" />
  </parameters>

  <code><![CDATA[
    action #name#(float time, string field_id) returns float
    {
      // If the field string is invalid, make it obvious!
      if field_identifier.length() == 0 then {
      return 0.0;
    }

      // Should return 0 if the field specifier is invalid
      return #timePlugin#.format
          (time,"%"+field_id).toFloat();
    }
  ]]></code>
</function>
```

Notes

Notes for this function:

- The value of the `function name` attribute, ExtractTimeField, is unique within the directory that contains this `.fdf` file.
- Appears as ExtractTimeField in the Event Modeler rules menu.
- Returns a float.
- The `imports` element specifies `timePlugin` as the alias for the plug-in, and `TimeFormatPlugin` as the shared library that contains the plug-in.
- The `code` element specifies `timePlugin` to refer to required plug-in.
- Takes two parameters — a `float` that specifies a time, and a `String` that specifies a field ID.
- The EPL ensures that the `ID` field is valid and then invokes the `format` function by specifying the alias for the `TimeFormatPlugin` library:

  ```
  return #timePlugin#.format
  ```

About defining your own functions

About function names

Functions have several different names:

- The file name — this is the name of the file that contains the function definition, for example, `String_String_Concat.fdf`.
- The logical name — this is the name specified by the `function name` attribute in the `.fdf` file. Event Modeler uses the logical name to distinguish each function from every other function in a particular directory. Within each directory, this value must be unique. For example, `SSConcat`.
- The display name —this is the name that appears in the Event Modeler Functions tab. For example, `Concat`. This name also appears in the Rules panel context menu.

The contents of a function definition file contain something like this near the beginning:

```
<function name="SSConcat" display-string="Concat"
   return-type="string">
```

In this example, the logical name is `SSConcat`. The display name is `Concat`.

For example, it is possible to have the following three functions in the same directory:

Filename:	`String_String_Concat.fdf`	`String_Integer_ Concat.fdf`	`String_Integer_String_ Concat.fdf`
Parameters:	`String, String`	`String, Integer`	`String, Integer, String`
Display name:	`Concat`	`Concat`	`Concat`
Logical name:	`ConcatSS`	`ConcatSI`	`ConcatSSS`

Note that these functions have the same display name but different logical names. An exact duplicate of any of these functions can be in a directory other than the directory that already contains its duplicate.

When you select functions from the rules editor context menu, Event Modeler displays the arguments that each function takes. Consequently, if two functions have the same display name, you can distinguish them by their arguments. For example:

```
TO_NUMBER('choice' value)
TO_NUMBER ('text' value)
```

About defining your own functions

Chapter 5: Using Standard Blocks

- A block's lifecycle 121
- General analytic blocks 122
- The Timer blocks 133
- The Utility blocks 137
- Database functionality—storage and retrieval 151
- Blocks for working with scenario blocks 159

Blocks are ready packaged modules that you can import and use in your scenarios. They can accept inputs, execute some logic of their own, and generate output. In Event Modeler, in the Catalogs tab, you can view and select the blocks provided with Apama.

A block is defined in a Block Definition File, or `.bdf`. This XML file describes what the block does and its implementation in Apama EPL. A block can consist of:

- *Parameters* – a block can have a number of parameters, which when set configure its behavior. Parameters differ from input fields, in that the latter are like work packages for the block to process and are expected to change all the time, while the former are typically only set to initialize the block and whenever its core behavior needs to be modified. Parameters are typed in the same way as scenario variables. Parameters are all provided at initialization time and can then be updated individually.
- *Operations* – in addition to any standard behavior that is hard-wired into it, a block can also have a number of explicit operations that can be invoked by the scenario. For example, typical operations are to start processing some data and to stop. If an operation requires any configuration information this is usually passed in through a block parameter.
- *Input feeds* – an input feed can be hooked up to a live stream of event data, like a price quote stream. Within it, an input feed will define one or more input fields, which can be mapped to data in the stream. When event data arrives, the fields' values get updated. These fields are typed in the same way as scenario variables.
- *Output feeds* – an output feed is a stream of output data that can be generated by the block. Each output feed corresponds to an event that can be generated by the block, and embeds one or more output fields. The fields are updated as a result of operations carried out by the block. These fields are typed in the same way as scenario variables.

When you add a block to a scenario, you are specifying that each instance of that scenario should create an instance of that block running within the scenario. Whether the block instance then starts executing some activity immediately or waits for some operation on it to be called depends entirely on how the block itself is written.

There is no restriction on the number of block instances that you can add to a scenario. It is possible to add multiple instances of the same block to a scenario. To ensure there is no conflict, each instance has its own operations, parameters and fields clearly tagged by its unique name.

You can save a scenario as a block, and then use that scenario block in other scenarios. In this way, you can create composite scenarios. However, you cannot create a block from a scenario that can run

in parallel. Also, you cannot create a block from a non-parallel scenario and then mark that block as parallel-aware. See "Working with Blocks Created from Scenarios" on page 84.

If there is no standard block that meets your needs you can use Apama Studio's block editor to create a custom block. You can use the block editor to define the block's parameters, operations, input feeds and output feeds, or you can use the block editor to create the block from an event definition. See "Creating Blocks" in *Using Apama Studio*.

Notes

Only the latest version of each standard block is documented here. Except where noted otherwise, one earlier version of each standard block is included in Apama. However, use of the latest version of a standard block is recommended for the following reasons:

- It implements the block as an event type, which is faster than the previous interface.
- It is parallel-aware. You can use it in a parallel scenario.
- Support for the earlier version will be removed in a future release.

Most standard blocks are automatically available to your scenario from the Catalogs tab. However, some standard blocks are available only if you add a particular bundle to your project. Where this is the case, the description of the standard block notes this.

A block's lifecycle

This section describes a block's lifecycle

1. You use Apama Studio to define a block. Apama Studio saves it as a *Block Definition File* (`.bdf`). This is an XML document, and it contains the interface of the block in XML elements as well as the EPL that defines the block's functionality.

   The EPL template for a block is the `<code>` section within the block's `.bdf` file. This contains the actual implementation of the block, embedding the custom behavior that identifies the block.

2. A scenario is defined within Event Modeler. This scenario is made to import one or more instances of the block. The scenario is saved to disk in a *Scenario Definition File* (`.sdf`) which is also an XML document. This document contains a reference to the location of any imported blocks' `.bdf` files. It does not embed the blocks themselves.

   During this stage, the contents of the `<code>` section in the `.bdf` are read in and all EPL names that are tagged with `#` characters are replaced with unique names that distinguish this particular block instance from any other that the scenario imports. The modified block EPL is then added to the scenario's EPL. Because certain elements of the EPL in the `<code>` section are renamed, this section of the code is often termed an EPL *template*.

3. The scenario and the referenced blocks are converted to an EPL file (`.mon`), either explicitly with File > Export > Apama > Export as EPL or implicitly when running the project from Apama Studio.

4. The EPL containing the combined scenario and block code described in Step 3 is injected into, and parsed by the correlator. Note that if the EPL supplied in the `.bdf` file is invalid, the correlator will reject the scenario at this stage. However, if the EPL is valid but does not correctly implement the block's interface, it will still inject successfully. This situation cannot be detected until the scenario does not function as expected.

5. At this point the EPL for the scenario and its embedded block(s) is now in the correlator. This means that actual instances of the scenario can be created by end users. Assume that a dashboard has been created with Dashboard Builder to go with the scenario, and that end users can therefore interact with the scenario through the Dashboard Viewer. When a user logs into the scenario's application and creates an instance (sometimes referred to as a strategy), the correlator will create a specific working instance of the scenario and of its embedded block(s). Each instance is unique and distinct. Therefore, if the scenario embedded two blocks (or even two copies of the same block), and three instances of it are created from a dashboard, there will then be three instances of the scenario and six block instances.

Therefore, when you add a block to a scenario in Event Modeler, you are effectively specifying that real instances of that scenario should each create an instance of that block running within them. Whether the block instance then starts executing some activity immediately or else waits for some operation on it to be called depends entirely on how the block itself was written.

It is possible to add multiple instances of the same block to a scenario in Event Modeler. Since their operations, parameters and fields are clearly specified by their enclosing block instance's name when invoked from the scenario there is no conflict at runtime. There is no restriction on the number of block instances that can be added to a scenario.

Using Standard Blocks

General analytic blocks

This section discusses Event Modeler analytic blocks.

- "Change Notifier v2.0" on page 122
- "Correlation Calculator v2.0" on page 124
- "Data Distribution Calculator v2.0" on page 125
- "Median and Mode Calculator v1.0" on page 127
- "Moving Average v1.0" on page 128
- "Spread Calculator v3.0" on page 129
- "Statistics Calculator v1.0" on page 130
- "Velocity Calculator v2.0" on page 132

Using Standard Blocks

Change Notifier v2.0

The Change Notifier block sends out a notification when its input data stream changes by a given amount over a configurable, moving time window. When a sufficiently large positive or negative change has occurred, the output feed will indicate this by setting the `changed` field to true. The output feed can be configured to automatically reset to its unchanged state a certain time after triggering by setting the `reset period` parameter.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation. Must be greater than zero.
`amount`	The change amount value, zero to ignore. A notification will be sent if the difference between the oldest value inside the time window and the most recent sample is greater than this amount. Absolute values are used in the calculations.
`percentage`	The change percentage value, zero to ignore. Absolute values are used as for the `amount` parameter. 100.0 means to look for a doubling of the input values.
`reset period`	Following the detection of a big enough change, the output feed will be reset to its un-triggered state after this interval. It is specified in seconds, and is ignored if less than or equal to zero.

At least one of `amount` and `percentage` should be different from 0.0, otherwise no notifications will occur.

Operations

Operation	Description
`start`	Starts checking for changes in the input data feed.
`stop`	Stops checking for changes.
`clear`	Discards all stored values.
`reset`	Resets the `changed` notification flag.

Input feeds

Feed	Field	Description
`data`	`value`	Feed of input values

Output feeds

Feed	Fields	Description
`notify`	`percentage change`	The amount of change measured as a percentage.
	`amount change`	The amount of change.

Feed	Fields	Description
	`changed`	Set true to indicate a sufficiently large change has occurred. Is reset to false by calling operation `reset`, or after the specified `reset period`.

General analytic blocks

Correlation Calculator v2.0

The Correlation Calculator block calculates the correlation coefficient between two streams of data. The calculation may be performed over an unlimited set of data from each stream, or a set limited by number of samples or age of samples. The calculator only generates an output if there is at least one suitable sample from each stream.

Correlation coefficient

A correlation coefficient approaching `+1.0` shows a strong correlation between the streams, a coefficient close to `0.0` shows little or no correlation between the streams and a coefficient approaching `-1.0` shows an inverse correlation between the streams; for example, if one is increasing, the other is decreasing.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation.
`size`	The maximum number of sample pairs that are used in the calculation. A pair consists of a sample from one stream, and the most recent sample from the other stream. The oldest sample is replaced by the newest sample when the total number of samples has reached this limit.

One or both of the above parameters must be `0`, in which case that limit is not imposed. It is not possible to restrict the number of samples by both age and number of samples, but it is possible to not impose any limit on the number of samples (thus an infinite set of samples is kept). Note that imposing a limit after input events have been received will clear all existing samples.

Operations

Operation	Description
`start`	Starts the calculation of coefficients. Must be called before the calculator will generate any statistics.
`stop`	Stops the calculation of further coefficients. Any subsequent input feeds are ignored.

Operation	Description
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data1`	`value`	The first input set.
`data2`	`value`	The second input set.

Note that at least one feed from both sets needs to have been received (and if set, within `period` seconds) before an output will be generated.

Output feeds

Feed	Fields	Description
`statistics`	`correlation`	The correlation coefficient (between `-1.0` and `+1.0`).
	`samples`	The number of sample pairs used for this calculation.

General analytic blocks

Data Distribution Calculator v2.0

The Data Distribution Calculator block calculates some common statistics from a set of samples. Like the correlation block, the set of samples may be unlimited in size, or constrained by a maximum number of samples or a maximum age of samples. Note that execution of the Median and Mode Calculator block, Moving Average block or Statistics Calculator block is faster than execution of the Data Distribution Calculator block. This is because those blocks perform a subset of the processing of the Data Distribution Calculator block.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation.
`size`	The maximum number of samples that are used in the calculation. The oldest sample is replaced by the newest sample when the total number of samples has reached this limit.

One or both of the above parameters must be `0`, in which case that limit is not imposed. It is not possible to restrict the number of samples by both age and number of samples, but it is possible to not impose any limit on the number of samples (thus an infinite set of samples is kept).

Operations

Operation	Description
`start`	Starts the calculation of statistics. Must be called before the calculator will generate any statistics.
`stop`	Stops the calculation of further statistics. Any subsequent input feeds are ignored.
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data`	`value`	The feed of values. The time of a value is taken to be the correlator's current time.

Output feeds

Feed	Fields	Description
`statistics`	`value`	The most recent value received in the input feed.
	`mean`	The arithmetic mean of the distribution.
	`mode`	The most commonly occurring value, if there is one.
	`no unique mode`	`true` if there is no single mode.
	`median`	The mid point of the ordered set of data values.
	`standard deviation`	Standard deviation of the data set.
	`variance`	Variance of the distribution.
	`skew`	Degree of skewed-ness of the distribution.
	`kurtosis`	Kurtosis measure of the distribution.
	`samples`	The number of samples used for this calculation.

General analytic blocks

Median and Mode Calculator v1.0

The Median and Mode Calculator block calculates the median and the mode from the input data stream over a configurable time window and sample set size. This block performs a subset of the processing performed by the Data Distribution Calculator block. Consequently, execution of this block is slightly faster than execution of the Data Distribution Calculator block. Like the Correlation Calculation block, the set of samples may be unlimited in size, or constrained by a maximum number of samples or a maximum age of samples.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation.
`size`	The maximum number of samples that are used in the calculation. The oldest sample is replaced by the newest sample when the total number of samples has reached this limit.

One or both of the above parameters must be `0`, in which case that limit is not imposed. It is not possible to restrict the number of samples by both age and number of samples, but it is possible to not impose any limit on the number of samples (thus an infinite set of samples is kept).

Operations

Operation	Description
`start`	Starts the calculation of statistics. Must be called before the calculator will generate any statistics.
`stop`	Stops the calculation of further statistics. Any subsequent input feeds are ignored.
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data`	`value`	The feed of values. The time of a value is taken to be the correlator's current time.

Output feeds

Feed	Fields	Description
`statistics`	`value`	The most recent value received on the input feed
	`mode`	The most commonly occurring value, if there is one.
	`no unique mode`	`true` if there is no single mode.
	`median`	The mid point of the ordered set of data values.
	`samples`	The number of samples used for this calculation.

General analytic blocks

Moving Average v1.0

The Moving Average block calculates the moving average from the input data stream over a configurable time window and sample set size. Like the Correlation Calculation block, the set of samples may be unlimited in size, or constrained by a maximum number of samples or a maximum age of samples. The Moving Average block performs a subset of the processing performed by the Data Distribution Calculator block. Consequently, execution of the Moving Average block is considerably faster than execution of the Data Distribution Calculator block.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation.
`size`	The maximum number of samples that are used in the calculation. The oldest sample is replaced by the newest sample when the total number of samples has reached this limit.

One or both of the above parameters must be `0`, in which case that limit is not imposed. It is not possible to restrict the number of samples by both age and number of samples, but it is possible to not impose any limit on the number of samples (thus an infinite set of samples is kept).

Operations

Operation	Description
`start`	Starts the calculation of statistics. Must be called before the calculator will generate any statistics.
`stop`	Stops the calculation of further statistics. Any subsequent input feeds are ignored.
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data`	`value`	The feed of values. The time of a value is taken to be the correlator's current time.

Output feeds

Feed	Fields	Description
`statistics`	`value`	The most recent value received on the input feed.
	`mean`	The arithmetic mean of the distribution.
	`samples`	The number of samples used for this calculation.

General analytic blocks

Spread Calculator v3.0

The Spread Calculator block calculates the difference between the latest data points of two streams. The output feed also provides the time of the event. This can either be supplied in the input feed or, if no mapping is provided for the input feed, the correlator's current time is used. Note that the first result will not be generated until both input feeds have received an event.

Parameters

There are no parameters for this block.

Operations

Operation	Description
`start`	Starts the calculation of differences. Must be called before any output events are sent.
`stop`	Stops the calculation of further coefficients. Any subsequent input feeds are ignored.
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data1`	`value`	The first feed of values.
	`time`	The timestamp of the data point. Leave unmapped (i.e. left as `0`) to set the time as the correlator's current time.
`data2`	`value`	The second feed of values.
	`time`	The timestamp of the data point. Leave unmapped (i.e. left as `0`) to set the time as the correlator's current time.

Output feeds

Feed	Fields	Description
`statistics`	`last1`	The most recent value sent to the `data1` feed.
	`time1`	The time of the most recent value sent to the `data1` feed.
	`last2`	The most recent value sent to the `data2` feed.
	`time2`	The time of the most recent value sent to the `data2` feed.
	`spread`	Difference between `last1` and `last2`. Will be negative if `last2` is greater than `last1`.

General analytic blocks

Statistics Calculator v1.0

The Statistics Calculator block calculates running statistics from a set of samples. Like the correlation block, the set of samples may be unlimited in size, or constrained by a maximum number of samples or a maximum age of samples. The Statistics Calculator block performs a subset of the processing

performed by the Data Distribution Calculator block. Consequently, execution of the Statistics Calculator block is considerably faster than execution of the Data Distribution Calculator block.

Parameters

Parameter	Description
`period`	The maximum age of any sample that is used in the calculations, in seconds. Any samples older than this will be discarded before performing the calculation.
`size`	The maximum number of samples that are used in the calculation. The oldest sample is replaced by the newest sample when the total number of samples has reached this limit.

One or both of the above parameters must be `0`, in which case that limit is not imposed. It is not possible to restrict the number of samples by both age and number of samples, but it is possible to not impose any limit on the number of samples (thus an infinite set of samples is kept).

Operations

Operation	Description
`start`	Starts the calculation of statistics. Must be called before the calculator will generate any statistics.
`stop`	Stops the calculation of further statistics. Any subsequent input feeds are ignored.
`clear`	Discards all current data.

Input feeds

Feed	Fields	Description
`data`	`value`	The feed of values. The time of a value is taken to be the correlator's current time.

Output feeds

Feed	Fields	Description
`statistics`	`value`	The most recent value received on the input feed.
	`mean`	The arithmetic mean of the data set.
	`standard deviation`	Standard deviation of the data set.
	`variance`	Variance of the data set.

Feed	Fields	Description
	skew	Degree of skewed-ness of the data set.
	kurtosis	Kurtosis measure of the data set.
	samples	The number of samples used for this calculation.

General analytic blocks

Velocity Calculator v2.0

Velocity calculates the rate of change (that is, change divided by the time between the changes) of the last two values of a stream. The time of incoming events is taken to be the correlator's current time. Note that the first result will not be generated until two events have been received on the input feed.

Parameters

This block has no parameters.

Operations

Operation	Description
start	Starts the calculation of velocity. Must be called before any output events are sent.
stop	Stops the calculation of velocity. Any subsequent input feeds are ignored.
clear	Discards all current data.

Input feeds

Feed	Fields	Description
data	value	The feed of values.

Output feeds

Feed	Fields	Description
velocity	value	The difference of the last two values divided by the time between the last two values. Values are assumed to arrive at no less than 0.01 seconds apart. Thus, no two events are considered to have the same timestamp, which would mean the velocity could not be computed.

General analytic blocks

The Timer blocks

Apama provides two timer blocks.

- "Schedule v3.0" on page 133
- "Wait v3.0" on page 136

Using Standard Blocks

Schedule v3.0

The Schedule block sends an output feed at a given time in the future. The time is specified by any combination of weekday, month, year, hour, minute and seconds. Any of the parameters may take a negative value, which means any value is allowed. Multiple timers may be started in a single block, each one having a different timer id. This timer id is supplied in the output feed when the timer fires, so may be used to determine what to do upon the timer firing.

Parameters

Parameter	Description
`timer id`	A string that distinguishes this timer from other timers in this block. An empty string is valid.
`month`	The month of the year (`1-12`) or negative for any month of the year.
`day`	The day of the month (`1-31`) or negative for any day of the month.
`hour`	The hour of the day (`0-23`) or negative for any hour of the day.
`minute`	The minutes past the hour (`0-59`) or negative for any minute.
`second`	The seconds past the minute (`0-59`) or negative for any second.

Operations

Operation	Description
`start`	Starts the specified timer id.
`cancel`	Cancels the specified timer id.
`retrieve`	Retrieve the details of the specified timer id by setting the output feed accordingly.

Input feeds

This block has no input feeds.

Output feeds

Feed	Fields	Description
`timer`	`timer id`	A string that distinguishes this timer from other timers in this block. An empty string is valid.
	`month`	The month (`1-12`).
	`day`	The day of month (`1-31`).
	`hour`	The hour (`0-23`).
	`minute`	The minute (`0-59`).
	`seconds`	The seconds (`0-59`).
	`time up`	`true` if time is up, `false` otherwise (i.e. on retrieval).
`book`	`num timers`	The number of currently active timers known to this block.

Examples

The following tables list the values for parameters that will trigger at the times described.

Example 1:

Parameter	Value	When triggered
`month`	`-1`	Once a month, on the first of every month, at 03:00:00.
`day`	`1`	
`hour`	`3`	
`minute`	`0`	
`seconds`	`0`	

Example 2:

Parameter	Value	When triggered
`month`	`-1`	Every hour, at 15 minutes past the hour.
`day`	`-1`	

Parameter	Value	When triggered
hour	-1	
minute	15	
seconds	0	

Note that the time and date information is simply a copy of the parameters used when starting the timer. Any field whose corresponding parameter was given a negative value will have that same value.

Example 3:

Parameter	Value	When triggered
month	-1	Every second.
day	-1	
hour	-1	
minute	-1	
seconds	-1	

Example 4:

Parameter	Value	When triggered
month	-1	Every day at noon
day	-1	
hour	12	
minute	0	
seconds	0	

Example 5:

Parameter	Value	When triggered
month	5	Once a year, at exactly 16:31:28 on 31st May
day	31	
hour	16	
minute	31	

Parameter	Value	When triggered
`seconds`	`28`	

The Timer blocks

Wait v3.0

The Wait block sends an output feed at a given time in the future. The time is specified by a number of seconds to wait from the time the start operation is called. A timer may be set to repeat. Multiple timers may be started in a single block, each one having a different timer id. This timer id is supplied in the output feed when the timer fires, so may be used to determine what to do when that happens.

Parameters

Parameter	Description
`timer id`	A string to identify this timer from others in used in this block (an empty string is valid).
`time`	The number of seconds to wait.
`repeat`	`true` if the timer should repeat, `false` if a single-shot.

Operations

Operation	Description
`start`	Starts the specified timer id.
`cancel`	Cancels the specified timer id.
`retrieve`	Retrieve the details of the specified timer id by setting the output feed accordingly.
`reset`	Resets the output feed. Useful for repeating timers to set the output feed's time up field to false.

Input feeds

This block has no input feeds.

Output feeds

Feed	Fields	Description
`timer`	`timer id`	The id of the timer, as supplied by the `timer id` parameter.

Feed	Fields	Description
	`time`	The time to wait in seconds.
	`repeat`	`true` if the timer repeats.
	`time up`	`true` if time is up, `false` otherwise (i.e. on retrieval).
`book`	`num timers`	The total number of timers known to this block.

The Timer blocks

The Utility blocks

Apama provides a number of utility blocks.

- "Dictionary v2.0" on page 137
- "File Reader v2.0" on page 138
- "File Writer v2.0" on page 140
- "History Logger v2.0" on page 141
- "Input Merger v2.0" on page 143
- "List v2.0" on page 144
- "Scenario Terminator v2.0" on page 145
- "Status v2.0" on page 146
- "Variable Mapper v2.0" on page 150

Using Standard Blocks

Dictionary v2.0

As scenarios do not support a `dictionary` type, the Dictionary block addresses this potential requirement by providing an associative map of (string) keys and values. It provides facilities for adding, accessing, removing, as well as iterating across, elements within this map.

Parameters

Parameter	Description
`key`	Holds the key for a `add` / `get` operation.
`value`	Holds the value for a `add` / `get` operation.

Operations

Operation	Description
`add`	Adds the name-value pair stored in `key` and `value` to the dictionary. If the key already exists, the value will be overwritten with the new value.
`get`	Retrieves the value for the key stored in the `key` parameter and causes a result to be sent out on the output stream.
`clear`	Empties the dictionary.
`remove`	Removes the entry with the key stored in the `key` parameter from the dictionary - fails silently if key does not exist (removed key and value will be sent out on the result output feed).
`next`	For iterating through the dictionary - forces the next result to be output.
`reset`	Resets the iterator to the first entry in the dictionary.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`result`	`key`	The key for the entry.
	`value`	The value of the entry.
	`found`	`true` if the key was found in the dictionary, `false` otherwise.
	`size`	Number of entries in the dictionary.

The Utility blocks

File Reader v2.0

The File Reader lets a scenario read a line at a time from a specified file using the File adapter with the `JMultiFileTransport` transport layer and the `JNullCodec` codec plug-in.

For details about using the File adapter, see *Developing Adapters*, "Apama file adapter".

The same File Reader block can read from multiple files.

Parameters

Parameter	Description
`Transport Name`	The name of the instance of the `JMultiFileTransport` to use. This must match a transport instance name specified in the IAF configuration file.
`File Name`	The name of the file to read.
`Lines In Header`	The number of lines to skip at the beginning of the file.
`File Channel`	The name of the channel to output file events to. The various file events are defined in the `FileEvents.mon` file, and the definitions are in the `com.apama.file` package. You can find `FileEvents.mon` in the `adapters/monitors` directory of your Apama installation directory.

Operations

Operation	Description
`Open File`	Opens a file according to the current values of the `Transport Name`, `File Name`, `Lines In Header` and `File Channel` parameters
`Close File`	Closes a file according to the current values of the `Transport Name`, `File Name` and `File Channel` parameters.
`Read Line`	Reads a line from the file. Uses the current values of the `Transport Name`, `File Name` and `File Channel` parameters.
`Get File Status`	Explicit call to update the `Status` output feed. Uses the current values of the `Transport Name`, `File Name` and `File Channel` parameters.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`line`	`file name`	The name of the file associated with the current line.
	`file transport`	The name of the transport associated with the current line.
	`line`	String that contains the current read line.
`error`	`file name`	The name of the file that returned the error.
	`file transport`	The name of the transport that returned the error.

Feed	Field	Description
	message	The error message returned.
status	file name	The name of the file associated with the status update.
	file transport	The name of the transport associated with the status update.
	more available	A flag that indicates whether there are currently more lines to read from the file.
	file currently open	A flag that indicates whether or not the file is currently open.

The Utility blocks

File Writer v2.0

The File Writer block lets a scenario write a line at a time to a specified file using the File adapter with the JMultiFileTransport transport plug-in and the JNullCodec codec plug-in. A single File Writer block can write to multiple files.

Parameters

Parameter	Description
Transport Name	The name of the instance of the JMultiFileTransport to use. This must match an instance name specified in the IAF configuration file.
File Name	The name of the file to write.
Append	A flag indicating whether to append to the end of a file, or whether to replace the contents of the existing file.
Line	The line to be written to the file identified by the File Name parameter.
File Channel	The name of the channel to output file events to. The various file events are defined in the FileEvents.mon file and they are defined in the com.apama.file package. You can find the FileEvents.mon file in the adapters/monitors directory of your Apama installation directory.

Operations

Operation	Description
`Open File`	Opens a file according to the current values of the `Transport Name`, `File Name`, `Append` and `File Channel` parameters.
`Close File`	Closes a file according to the current values of the `Transport Name`, `File Name` and `File Channel` parameters.
`Write Line`	Writes a line to the file identified by the current values of the `Transport Name`, `File Name`, `Line` and `File Channel` parameters.
`Get File Status`	Explicit call to update the `Status` output feed. Uses the current values of the `Transport Name`, `File Name` and `File Channel` parameters.

Input feeds

This block has no input feeds

Output feeds

Feed	Field	Description
`error`	`file name`	The name of the file that returned the error.
	`file transport`	The name of the transport that returned the error.
	`message`	The error message returned.
`status`	`file name`	The name of the file associated with the status update.
	`file transport`	The name of the transport associated with the status update.
	`file currently open`	A flag that indicates whether the file is currently open.

The Utility blocks

History Logger v2.0

The History Logger block maintains an ordered and (optionally) time-stamped history of text messages. This is normally used in conjunction with multi-line entries in dashboards, such as history lists, where a fixed size list is used to contain a rolling window of constantly changing information.

Parameters

Parameter	Description
`entry`	An entry to be added to the history.
`timestamps`	Index for an `add`, `clear` or `retrieve` operation.
`most recent first`	Set to `true` to order the history so that the most recent element is first, `false` for least recent first.
`max size`	Maximum number of entries to retain - set to `0` to retain all entries.
`delimiter`	String to separate history entries when output by the block. If not specified, the default is "`\n`" (linefeed).
`time format`	String format to display time-stamps, if required. A default format is used if this is not set.

Operations

Operation	Description
`add`	Adds the content of entry to the history - an output update will automatically be produced.
`clear`	Clears the history.
`retrieve`	Causes the latest history to be output from the block as a single, delimiter-separated string.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`history`	`size`	Number of entries in the history.
	`text`	Text representation of the history, where each entry is optionally time-stamped and separated by the delimiter string.

The Utility blocks

Input Merger v2.0

The Input Merger block collects a number of related field values and outputs them simultaneously.

Description

The input event is a field name/value pair. If the name in a pair matches one of the names in the `order` parameter, the corresponding value is stored for output. When all of the names in `order` have been matched at least once, the set of stored values is output. Note that multiple matches (and stores) can occur for any name. In this case, the latest store overwrites the value of the previous store, ensuring that each field has the latest value.

If the `incremental update` parameter is set, then further outputs are generated on any input that matches a field in the `order` parameter. If the `incremental update` parameter is not set, then further outputs are only sent once all fields have been received again (that is, the old input values are discarded). The `id` field increments with each output event, in either mode.

Parameters

Parameter	Description
`order`	A comma-separated list of up to 8 field names to match against names on the input stream. The order in which the names are listed is the order in which they appear on the output. Note that fields may not contain commas, but they may be repeated or be an empty string.
`incremental input`	If `true`, a change to a single field listed in the `order` parameter results in an output being generated once all input fields have been received at least once — that is, the first output is still generated only when all fields have been received.

Operations

Operation	Description
`start`	Activate merger.
`stop`	Deactivate merger.

Input feeds

Feed	Field	Description
`in`	`name`	Field name
	`value`	Field value

Output feeds

The `out` feed specifies the selected individual values from the input feed, in the order they are listed by the `order` parameter.

Feed	Field	Description
`out`	`id`	Increments each time an output event occurs, even if none of the other fields has changed from the previous output event.
	`1`	Field 1
	`2`	Field 2
	`3`	Field 3
	`4`	Field 4
	`5`	Field 5
	`6`	Field 6
	`7`	Field 7
	`8`	Field 8

The Utility blocks

List v2.0

As scenarios do not support a sequence type, the List block addresses this potential requirement by providing a dynamically-sized sequence of string items. It provides facilities for adding, inserting, accessing, removing, as well as iterating across, elements within this sequence.

Parameters

Parameter	Description
`item`	Holds an item for an `add` or `nextIndex` operation.
`index`	Index for an `add`, `get` or `remove` operation.

Operations

Operation	Description
`add`	Adds the value currently held in `item` to the end of the list.

Operation	Description
`insert`	Adds the value held in `item` to the list at the position held in `index`.
`get`	Retrieves the item stored at the position held in `index`.
`clear`	Empties the list.
`remove`	Removes the item at the position stored in the `index` parameter.
`next`	For iterating through the list - forces the next result to be output.
`reset`	Resets the iterator to the first entry in the list.
`nextIndex`	For iterating through the list - move the iteration position to the next instance of item stored in the item parameter and outputs the results.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`result`	`item`	Holds the item for a retrieval operation.
	`index`	Holds the index of a retrieved item.
	`found`	`true` if an item was found in the list, `false` otherwise.
	`size`	Number of entries in the list.

The Utility blocks

Scenario Terminator v2.0

The Scenario Terminator block is unusual in that it does not directly interact with the scenario through any feeds, parameters or operations. The Scenario Terminator block simply listens for special events that can be sent to the correlator, and terminates the scenario if requested to.

Description

The Scenario Terminator block depends on the `ScenarioDeleterSupport.mon` file, which is supplied in the `monitors` folder. This EPL file must be injected before a scenario containing the Scenario Terminator block can be injected.

Unlike other blocks, there is no value in including the block more than once, though doing so is not an error.

This block has no parameters, no operations, no input feeds, and no output feeds.

The Scenario Terminator block listens for the following events:

```
com.apama.scenarios.DeleteAllScenarios()
com.apama.scenarios.DeleteScenariosByUser(string owner)
```

The first deletes all scenarios with a Scenario Terminator block. The second deletes all scenarios for the given dashboard username that have a Scenario Terminator block. For example, to delete all scenarios for the user `roguetrader`, do the following:

```
com.apama.scenarios.DeleteScenariosByUser("roguetrader")
```

The Utility blocks

Status v2.0

The Status block obtains the status of an object managed by a service monitor. For example, you can use the Status block to obtain the status of a market, a connection, or some other component. The objects for which you can obtain status and the meaning of various parameters depend on the service monitor providing the status.

Usage notes

You use the Status block with the `com.apama.statusreport.*` events, which are defined in `StatusSupport.mon` in the `monitors` directory of your Apama installation directory. There are four `com.apama.statusreport` event types:

- `SubscribeStatus` events — the Status block sends a `SubscribeStatus` event to a service monitor to initiate receipt of status events from that service. A `SubscribeStatus` event identifies the ID of the service you want to receive status from, the object you want status for, the sub-service ID, if there is one, to receive status from, and the connection to use if there is a choice.

  In a `SubscribeStatus` event, when the service ID is an empty string, the Status block is initiating a status subscription with each service monitor that is listening for `SubscribeStatus` events that have an empty string for the service ID. In this case, you should expect to receive status events from more than one service.

- `UnsubscribeStatus` — the Status block sends an `UnsubscribeStatus` event to a service monitor to terminate receiving status from that service. An `UnsubscribeStatus` event identifies the same information as a `SubscribeStatus` event.

- `Status` — a subscribed service sends a `Status` event to the Status block to provide the status information. A service sends a `Status` event as the result of a new subscription and whenever there is a change in status. In addition to identifying the service that the information is from and the object that the information is for, the `Status` event contains a string that contains a status description, a sequence that contains one or more key words, a Boolean indication of whether the object is in a state in which it can be used, and a dictionary that contains any other information that the service can provide.

- `StatusError` — a subscribed service sends a `StatusError` event to the Status block when it cannot provide status information. In addition to identifying the service that the event is from and the object that the event pertains to, the `StatusError` event contains a free-form string that describes the problem, and a Boolean indication of whether the status subscription was terminated.

The Status block uses these events to interface with any service monitor that supports the `com.apama.statusreport` interface. In other words, these events form the message exchange protocol (MEP) between the Status block in your Apama application and service monitors. For example, a service monitor might be the part of your adapter that makes the features of the adapter available to your Apama application.

Parameters

Parameter	Description
`serviceID`	String that identifies the service monitor that you want to subscribe to for status information. Leave blank (empty string) to subscribe to all service monitors that are currently listening for `com.apama.statusreport.SubscribeStatus` messages.
`object`	String that identifies the object that you want status for. The service monitor defines the values that you can specify here. For example, a service monitor might provide status for `Connection` or `Market`.
`subServiceID`	For service monitors that provide sub-services, this string identifies the sub-service that you want to subscribe to for status information. If the service monitor has no sub-services, leave this parameter blank.
`connection`	For service monitors that provide status for several instances of the specified object, this string identifies the instance for which you want to obtain status information. If the service monitor provides status for only one instance, leave this parameter blank. For example, an adapter might connect to multiple sources of data. You would use this parameter to specify the data connection you are interested in. The service monitor must define the allowable values for the `connection` parameter.
`extract key 1` `extract key 2` `extract key 3`	These three parameters make it convenient to obtain particular values from the `extracted parameter`*n* output fields in the Status block output feed. Each parameter is a string that specifies a key whose value you want to obtain in the status received from the service monitor. For example, when you set the `extract key 1` parameter to the value of a key defined in the service monitor, the `Status` output feed contains the specified key's value in its `extracted parameter 1` field. These fields make it easier to access particular elements in the extra parameters field of the output feed. You do not need to parse the payload string in the extra parameters field yourself.

Operations

Operation	Description
`start`	Initiates subscription to the service monitor identified by the `serviceID` parameter, for information about the component identified by the `object` parameter. If the specified service monitor has sub-services or provides information about more than one object instance, the subscription is for the sub-service and connection identified by the values that the `subServiceID` and `connection` Status block parameters have when the `start` operation is called. If the value of the `serviceID` parameter is an empty string, the `start` operation initiates a subscription to each service monitor that is listening for `SubscribeStatus` events that have an empty string in their `serviceID` field. Under the covers, the Status block routes a `SubscribeStatus` event to the correlator. This event takes its values from the current values of the Status block parameters. After a service monitor receives a `SubscribeStatus` event, it starts sending `Status` events to the subscribing scenario.
`stop`	Terminates the subscription to the service monitor identified by the `serviceID` parameter. If the value of the `serviceID` parameter is an empty string, the `stop` operation terminates the subscription to each service monitor that is listening for `UnsubscribeStatus` events that have an empty string in their `serviceID` field. Under the covers, the Status block routes an `UnsubscribeStatus` event to the correlator. This event takes its values from the current values of the Status block parameters. If a scenario terminates without invoking the `stop` operation for a subscription, the block routes the appropriate `UnsubscribeStatus` events upon termination of the scenario.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`Status`	`serviceID`	String that identifies the service monitor that is providing the status.
	`object`	String that identifies the object that the status is for.
	`subServiceID`	String that identifies the sub-service that is providing the status. This is blank if the service has no sub-services.
	`connection`	String that identifies the object instance that status is being provided for.

Feed	Field	Description
	`description`	String that contains human-readable text that describes the status.
	`summaries`	One word or a series of space-separated words that describe the status. For example, `Connected`, `Disconnected`, `LoginFailed`. The service monitor defines and documents the words that can appear in the `summaries` field. While the `description` field is for a human reader, the `summaries` field contains key words that a scenario can act on. For example, suppose `summaries` contains `Disconnected`. The scenario can define a rule that specifies what to do when this service is disconnected.
	`available`	Boolean value that indicates whether the object is in a state where it can be used. For example, if you specify `Market` as the object, a value of `true` in the `available` field might mean that the market is open and accepting orders.
	`extra parameters`	Payload-format string that contains any other information that the service monitor provides for the object.

	`extracted parameter 1`
	
	`extracted parameter 2`
	
	`extracted parameter 3`
		Each of these parameters is a string that contains the value of one of the key/value pairs that is in the `extra parameters` output field. The particular key value that the field contains is determined by the value that the corresponding `extract key` *n* block parameter had when the block's `start` operation was invoked.
		
		For example, suppose that the `extract key 1` parameter has a value of `time`. The block then invokes the `start` operation to subscribe to a particular service monitor. When the block receives status information from that monitor, the block inserts the value of `time`, for example, `"12:34:56"` into the `extracted parameter 1` field and then sends the information to its `Status` output feed.

	`received status`	Boolean value that indicates whether a `Status` event has been received from the specified service monitor. Initially, this field is `false`. When the block receives a `Status` event, it sets this field to `true`. When the block unsubscribes from the specified service monitor or when the block receives a `StatusError` event, the block sets the `received status` field to `false`. A value of `true` means that the information in the `Status` output feed is from the latest `Status` event and no error has since been signaled by the service monitor. In other

Feed	Field	Description
		words, you can trust the information in the `Status` output feed.
	`fault`	Boolean value that indicates whether there was an error obtaining status information for the specified object. When the service monitor sends a `StatusError` event, the block sets this field to true. You should consider any information from this service monitor to be stale.
	`total`	Integer that indicates the number of objects for which all of the following are true: - The block is receiving status information for the object. - The block has not received a `StatusError` event from the service monitor since the block received the previous `Status` event. - The object is in a state in which it can be used. That is, the value of the `available` output field is `true`. This field makes it convenient to track when a subscription is no longer providing status information. For example, if a Status block has 4 subscriptions but `total` = `3`, then the scenario can take some action such as restoring the subscription, or not using stale data.

The Utility blocks

Variable Mapper v2.0

The Variable Mapper block lets you use a scenario variable as a data source for any other block. The Variable Mapper block takes the name of a scenario variable as the value of its only input parameter. When the value of the mapped variable changes, the Variable Mapper block sends the new value to its output feed. The output feed includes two values. The first value is the new value as a number. The second value is the new value as text. You can choose which representation you need to wire into another block.

Parameters

Parameter	Description
`variable`	Name of the scenario variable whose value you want to output.

Operations

None.

Input feeds

This block has no input feeds.

Output feeds

Feed	Field	Description
`variable updates`	`number`	New value of the scenario variable as a number type.
	`text`	New value of the scenario variable as a text type.

The Utility blocks

Database functionality—storage and retrieval

The Database blocks let you store rows in a database and send queries to the database to retrieve a set of rows. They take parameters that let you specify a `database` name, a `table` name, a `user name` and `password`, and a `service identifier`. Note that any password given in the scenario or through the dashboard will be visible on screen.

The ADBC Storage block takes a list of `fields` and a list of `values` as parameters. The block places the values into their corresponding entry into the list of fields. Alternatively, the storage block takes a storage query or statement.

The ADBC Retrieval block takes a query string as a parameter. If you specify a query template, there is a parameter for specifying the query template parameters.

The format for a complete query string is service specific, typically SQL or an SQL-like language. When you specify a complete query, the block ignores the parameters that list `fields`, `values`, or a `where` clause.

The retrieval block return a number of outputs, one for each field/value pair for each row that matched the query. The scenario needs to call the `next` operation to retrieve the next field/value pair. The row number indicates when a field/value pair belongs to a different row. The row number counts from 1 upwards.

The Database blocks are:

- "ADBC Storage v1.0" on page 151
- "ADBC Retrieval v1.0" on page 154

Using Standard Blocks

ADBC Storage v1.0

The ADBC (Apama Database Connector) Storage block uses the ADBC adapter to store data in a database. To make this block available to your scenario, add the ADBC for JDBC or ADBC for ODBC bundle to your project. Adding one of these bundles to your project automatically adds the ADBC Common bundle, which contains the ADBC blocks.

Description

The ADBC adapter is a standard adapter provided with Apama. It provides general database storage and retrieval (query) and also event capture and playback. The ADBC adapter supports both standard SQL and specialized databases. In particular, the adapter supports ODBC and JDBC. This support provides access to most commercial and open source SQL databases. ADBC provides a superset of the functionality that was available in the ODBC and JDBC Apama standard adapters.

The Storage block can also be used to perform standard SQL operations such as Delete, Update, and Rollback. To carry out an SQL operation, the value of the `statement` parameter (described below) should be set to the operation you want to carry out.

Parameters

Parameter	Description
`service identifier`	The name of the service to use.
`database`	The data source name of the database to connect to.
`user name`	The username to use when connecting to the database.
`password`	The password to use when connecting to the database (will be readable on screen).
`table`	The name of the table to store data in.
`fields`	A comma-separated list of field names.
`values`	A comma-separated list of values that will be placed in the fields list.
`statement`	If this is not empty, the correlator uses this as the storage command instead of using the `fields` and `values` parameters. This parameter can be set to an SQL operation such as `UPDATE`, `DELETE`, or `ROLLBACK`.
`autocommit`	The auto commit mode to use. The default is an empty string. Specify one of the following: • `OFF` indicates no auto commit mode. • `ADBC` indicates the ADBC adapter auto commit mode based on a time period. • `DATA_SOURCE` indicates a data source specific auto commit mode. This might not be available for all data sources.
`acknowledge store`	Boolean that indicates whether the data source returns an acknowledgement to indicate success or failure for each store performed. True indicates that the data source always sends an acknowledgement. False indicates that the data source returns only store errors. The default is true. The success

Parameter	Description
	acknowledgement along with the current auto commit setting determine whether the data has been stored. A commit operation might also be needed.
`unique connection`	Boolean that indicates whether or not to create a new database connection. True indicates that you want the block to always create a new connection. False indicates that the block can use an existing connection. The default is false.
`final store`	If `true` indicates this will be the last store operation performed. Default value is `false`.If `true` the output feed field `committed.final store complete` will be set to true after the store operation completes (success or failure).

Operations

Operation	Description
`connect`	Establish a connection to the database.
`store`	Store in the database the data held in the block's parameters.
`commit`	Commit any data sent to the database.
`rollback`	Rollback uncommitted changes to the database.
`reset`	Resets the output feed.
`disconnect`	Close the database connection.

Input feeds

This block has no input feeds.

Output feeds

Feed	Fields	Description
`result`	`success`	`true` if the last update to the database was successful.
	`message`	A message from the last database update operation.
	`connected`	`true` if connected to the database.
`committed`	`status`	`true` if the last commit operation succeeds, else false.
	`final store complete`	`true` when the `store` operation with the `final store` parameter set to `true` has completed.

Feed	Fields	Description
`rollback`	`status`	`true` if the last rollback operation succeeded; otherwise `false`.

Database functionality—storage and retrieval

ADBC Retrieval v1.0

The ADBC (Apama Database Connector) Retrieval block uses the ADBC adapter to retrieve data from a database. The ADBC adapter is a standard adapter provided with Apama. To make this block available to your scenario, add the ADBC for JDBC or ADBC for ODBC bundle to your project. Adding one of these bundles to your project automatically adds the ADBC Common bundle, which contains the ADBC blocks.

Description

The ADBC adapter is a standard adapter provided with Apama. It provides general database storage and retrieval (query) and also event capture and playback. The ADBC adapter supports both standard SQL and specialized databases. In particular, the adapter supports ODBC and JDBC. This support provides access to most commercial and open source SQL databases. ADBC provides a superset of the functionality that was available in the ODBC and JDBC Apama standard adapters.

The ADBC Retrieval block supports prepared queries, stored procedures, and query templates. For more information see:

- "Prepared queries" on page 157
- "Stored procedures" on page 158
- "Query templates" on page 158

Parameters

Parameter	Description
`service identifier`	The name of the service to use, or blank for any service.
`database`	The data source name of the database to connect to.
`user name`	The username to use when connecting to the database.
`password`	The password to use when connecting to the database (will be readable on screen).
`table name`	The name of the table to retrieve data from.
`query string`	The data source specific query statement to be used. If you specify a query template name, be sure to set the `query parameters` parameter as needed for the template.

Parameter	Description
`query parameters`	If you specify a query template in the `query string` parameter, specify the parameters for the query template here. This is a comma separated list of `name:value` pairs, for example, `TABLE_NAME:Trade, SORT_ORDER:asc`.
`input types`	The input types of the parameters in the query template that is specified in the query. These are listed in a comma separated list of types, such as `Double, Double, Float`.
`output types`	The output types of the parameters in the query template that is specified in the query. These are listed in a comma separated list of types, such as `Double, Double, Float`.
`prepared query named id`	A `String` that uniquely identifies this prepared query.
`prepared query params`	The parameters to a prepared query in the form of a comma separated list of values.
`batch size`	Number of rows to be buffered in the block. The default is 50. The maximum is 10,000.
`disable buffering`	Boolean that indicates whether the results are streamed automatically as they are received. True indicates that they are. When set, the next `rewind` and `reset` operations have no effect since they are not needed. For use when wiring the ADBC Retrieval block's output to another block. The default is false.
`unique connection`	Boolean that indicates whether or not to create a new database connection. True indicates that you want the block to always create a new connection. False indicates that the block can use an existing connection. The default is false.

Operations

Operation	Description
`connect`	Establish a connection to the database.
`query`	Perform the query operation.
`reset`	Reset the output feed.
`next`	Look up the next field/value pair.
`rewind`	Rewind to the first result in the current buffered batch, without performing the operation again.
`stop`	Stop the query, even if not complete.

Operation	Description
`disconnect`	Close the database connection.
`create prepared query`	Create a prepared query for use later, passing in the correct input types.
`run prepared query`	Run a previously created prepared query, passing in the relevant input parameters.
`delete prepared query`	Delete an existing prepared query.
`retrieve query templates`	Retrieve a full list of named queries available, including the query template name, parameters and description.

Input feeds

This block has no input feeds.

Output feeds

Feed	Fields	Description
`schema`	`names`	The field names of the results.
	`types`	The Apama types of the fields.
	`indexable`	The names of the fields that are indexes.
`results`	`number`	The row number of the field/value pair. A number of `-1` indicates the end of data.
	`field`	The name of the field the value was taken from.
	`value`	The value of the field.
`error`	`message`	A message that describes the error if the store operation was unsuccessful.
`status`	`no more`	`true` if the current query has been completed and no more field/value pairs are available after the current pair.

Feed	Fields	Description
	more available	true if there is more data available to be read within the current batch and false otherwise.
	connected	true if connected to the database.
prepared query	created	True if the query is successfully created; false otherwise.
	deleted	True if the query is successfully deleted; false otherwise.
query templates	retrieved	false until the last query template is retrieved, at which point becomes true.
	query name	The identifying query name.
	query parameters	The list of parameters that the query requires.
	query description	A brief description of the purpose of the query.

Note that it is possible for `no more` to be false and `more available` to be false; this means that the service is waiting for more results to become available, but they have not been supplied by the database yet. The scenario should wait until `more available` becomes true before calling `next`. As with the order manager iteration, the scenario will need to re-enter the state it is in while iterating, in order to re-evaluate all of the rules in that state.

Prepared queries

Creating prepared queries

1. The query string parameter should be set with the prepared query string, such as `SELECT * FROM tablename WHERE intfield < ?`
2. The input types of the input parameters in the prepared query being created. This is a comma-separated list of types, for example `Double, Double, Float`, etc.
3. The output types of the parameters in the prepared query being created should be set to a comma-separated list of types, for example `Double, Double, Float` if calling on a stored procedure.
4. In the block's `prepared query named id` parameter specify a unique identifier in the form of a user readable name (`string`) for this prepared query. Multiple prepared queries can exist in the block at any one time, so the identifier allows you to specify which query you want to use.
5. Call the `create prepared query` operation.

In the `prepared query` output feed, the `created` field will contain `true` if the query was successfully created.

Using prepared queries

1. In the block's `prepared query named id` specify the identifier of the prepared query you want to execute.
2. In the `prepared query params` parameter, list the values which should match, in types and number, those of the input types.
3. Call the `run prepared query` operation.
4. From this point on, the `no more` and `more available` fields and the `next` and `stop` operations behave in the same manner as they do for normal queries.

Deleting prepared queries

1. To delete a prepared query, set the `prepared query named id` parameter to the identifier of the prepared query you want to delete.
2. Call the `delete prepared query` operation.

   In the `prepared query` output feed, the `deleted` field will contain `true` if the query was successfully deleted.

Stored procedures

Stored procedures must be created and deleted externally to the retrieval block, as in the case when creating a table in the database.

1. Once the stored procedure exists in the database you can create a prepared query, as in "Prepared queries" on page 157, above. The syntax for using a stored procedure in a query string is in the form `{call demo_stored-procedure(?,?)}`.
2. Specify the `input types` and `output types` parameters. Use `NULL` in the list of types for padding purposes. For example, given a `Double` (input only), `Double` (both input and output), and `Float` (output only), for the `input types` parameter specify `Double, Double, NULL` and for the `output types` parameter specify `NULL, Double, Float`.
3. Set an identifier in the `prepared query named id` parameter with this prepared query for future use.
4. Call the `create prepared query` operation.

   In the `prepared query` output feed, the `created` field will contain `true` if the query was successfully created.
5. Using the prepared query associated with the stored procedure is the same as described in "Prepared queries" on page 157, above.

Query templates

Retrieving query templates

You can retrieve the list of query templates that are associated with the project, by calling the `retrieve query templates` operation. In the `query templates` output feed, the `query name`, `query paramters`, and `query description` fields show each query template's name, parameters, and description, respectively. The `retrieved` field is `true` when all query templates have been retrieved.

Running query templates

1. Set the block's `query string` parameter to the name of the query template you want to run, such as `findEarliest`.
2. In the block's `query parameters` parameter specify the query parameters required by the query template, for example, `TABLE_NAME:tableName,TIME_COLUMN_NAME:timefield`.
3. Call the `query` operation to execute the query template, in the same way as for normal queries.

Database functionality—storage and retrieval

Blocks for working with scenario blocks

Apama provides two blocks for working with scenario blocks.

- "Change Observer v2.0" on page 159
- "Filtered Summary v2.0" on page 161

Using Standard Blocks

Change Observer v2.0

The Change Observer block watches sub-scenarios for changes in the value of one of the sub-scenario variables. You specify which variable you want to watch. When the value changes, the Change Observer block sends data to its `change` output feed. The output feed indicates the old value and the new value.

Description

To use the Change Observer block, wire output fields from the scenario block to input fields of the Change Observer block. Typically, you want to map the scenario block `instance id` output field to the Change Observer `stream` input field. Then map one of the sub-scenario variables from the scenario block `output` feed to the Change Observer `watchValue` input field. When the Change Observer block detects a change in a variable value, it sends notification of this change to its output feed.

Typically, you use the sub-scenario instance ID as the key. The key's associated value is the variable whose value you want to watch.

You can specify a filter so that you obtain results from a particular set of sub-scenarios.

You can also remove keys and their associated values from the Change Observer block's internal data store. This lets you exclude certain data from calculations. One way to do this is to define a global rule that watches for sub-scenarios to terminate. When a sub-scenario terminates, you can specify its instance ID as the key and remove the data for that key from the Change Observer block's store of data.

For a detailed example of using the Change Observer block, see "Observing changes in sub-scenarios" on page 100.

Parameters

Parameter	Description
`filter`	String that indicates that you want to observe those key/value pairs for which the `input filter` field matches this field. An empty string as the value of either the `filter` parameter or the `input filter` field indicates that there is no filtering. If the value of the `filter is "not equal to"` parameter is true, and you specify a value for the `filter` parameter, the Change Observer block observes key/value pairs for which the `input filter` field does NOT match the value of the `filter` parameter.
`keyToDelete`	String that indicates a key for which you want to delete data from the Change Observer block's internal store of data. Invoke the `deleteKey` operation to delete the data associated with this key.
`filter is "not equal to"`	Boolean that indicates whether you want to match or not match the value of the `filter` parameter. When the `filter is "not equal to"` parameter is true, the Change Observer block observes key/value pairs for which the `input filter` field does NOT match the value of the `filter` parameter.

Operations

Operation	Description
`reset`	The Change Observer block stores data about the number of unique keys it has observed and their most recent associated values. This operation flushes that data; it is no longer accessible to the Change Observer block.
`deleteKey`	Deletes the key defined by the `keyToDelete` parameter. This operation deletes data from the Change Observer block's internal store of data. If the value of the `keyToDelete` parameter is an empty string, this operation does nothing.

Input feed

The Change Observer `input` input feed provides the key, the value, and possibly a filter.

Feed	Fields	Description
`input`	`stream`	String that contains the key for which you want the Change Observer block to observe changes. Typically, the key is the instance ID of a sub-scenario. The Change Observer block ignores blank keys, that is, a key that is an empty string.
	`watchValue`	String that contains the field you want to watch. Typically, this is the value of a sub-scenario variable.

Feed	Fields	Description
	`filter`	String that contains a filter for determining the key/value pairs you are interested in.

Output feed

The Change Observer `change` output feed indicates the key, its old value, and its new value.

Feed	Fields	Description
`change`	`stream`	String that contains the key that this change is for. Typically, this is the instance ID of a sub-scenario.
	`oldValue`	String that contains the value of the variable being observed just before the value changed.
	`newValue`	String that contains the new value of the variable being observed.

Blocks for working with scenario blocks

Filtered Summary v2.0

The Filtered Summary block performs simple calculations across a set of sub-scenarios. This is an alternative to iterating over a set of sub-scenarios. The Filtered Summary block can operate on only floating point values.

Description

In more general terms, the Filtered Summary block performs calculations on a keyed set of floating point values. Typically, you use the sub-scenario instance ID as the key. The key's associated value is the value of a sub-scenario floating point variable that you want to use in an aggregate calculation.

You can specify filters to perform calculations on a sub-group of sub-scenarios. You can also remove keys and their associated values from the Filtered Summary block's internal data store. This lets you exclude data from certain sub-scenarios from the calculations. One way to do this is to define a global rule that watches for sub-scenarios to terminate. When a sub-scenario terminates, you can specify its instance ID as the key and remove the data for that key from the Filtered Summary block's store of data.

To use the Filtered Summary block, wire output fields from the scenario block to input fields of the Filtered Summary block. Typically, you want to map the scenario block `instance id` output field to the Filtered Summary `key` input field. Then map a floating point sub-scenario variable from the scenario block `output` feed to the Filtered Summary `value` input field.

Parameters

Parameter	Description
`filter`	String that indicates that you want to perform calculations on only those key/value pairs for which the `input filter` field matches this field. An empty string as the value of either the `filter` parameter or the `input filter` field indicates that there is no filtering. If the value of the `filter is "not equal to"` parameter is true, and you specify a value for the `filter` parameter, the Filtered Summary block operates on key/value pairs for which the `input filter` field does NOT match the value of the `filter` parameter.
`keyToDelete`	String that indicates a key for which you want to delete data from the Filtered Summary block's internal store of data. Invoke the `deleteKey` operation to delete the data associated with this key.
`filter is "not equal to"`	Boolean that indicates whether you want to match or not match the value of the `filter` parameter. When the `filter is "not equal to"` parameter is true, the Filtered Summary block operates on key/value pairs for which the `input filter` field does NOT match the value of the `filter` parameter.

Operations

Operation	Description
`reset`	The Filtered Summary block stores data about the number of unique keys it has observed and their most recent associated values. This operation flushes that data; it is no longer accessible to the Filtered Summary block.
`deleteKey`	Deletes the key defined by the `keyToDelete` parameter. This operation deletes data from the Filtered Summary block's internal store of data. If the value of the `keyToDelete` parameter is an empty string, this operation does nothing.

Input feed

The `input` input feed provides the key, the value, and possibly a filter.

Feed	Fields	Description
`input`	`key`	String that contains the key under which you want the Filtered Summary block to store data in its internal data store. Typically, the key is the instance ID of a sub-scenario. The Filtered Summary block ignores blank keys, that is, a key that is an empty string
	`value`	A `float` value that you want to operate on. Typically, this is the value of a sub-scenario variable.

Feed	Fields	Description
	`filter`	String that contains a filter for determining the key/value pairs you are interested in.

Output feed

The `data` output feed indicates the number of keys for which data is stored, the sum of the stored values, and the average of the stored values.

Feed	Fields	Description
`data`	`numberOfKeys`	Integer that specifies the number of unique keys for which the Filtered Summary block currently stores data.
	`totalValue`	Floating point value that is the sum of the values that the Filtered Summary block currently stores.
	`averageValue`	Floating point value that is the average of the values that the Filtered Summary block currently stores.

Blocks for working with scenario blocks