FUJITSU Software
Interstage Big Data
Complex Event Processing Server V1.1.0

Developer's Reference

Linux(64)

J2UL-1668-02ENZ0(00)
February 2014

Preface

Purpose of this document

This document provides a reference for the processing language used by Interstage Big Data Complex Event Processing Server (hereafter referred to as "BDCEP"), input adapter communication methods, etc. It also describes the language format, available functions, and communication protocols.

Intended readers

This document is intended for users who are considering developing applications that use BDCEP.

Structure of this document

This document is structured as follows:

Chapter 1 Complex Event Processing Language Reference

Reference for the event processing language used by Complex Event Processing of BDCEP. This section describes the language syntax.

Chapter 2 Filter Rule Language Reference

Language reference for the rules used by High-speed Filter of BDCEP. This section describes the language format, the available functions, etc.

Chapter 3 Input Adapter Reference

Reference explaining the BDCEP input adapter functionality for each communication protocol. This section also provides examples to be used for reference when developing event sender applications.

Chapter 4 Custom Listener Reference

Reference for developing user-developed Java classes. It provides an overview of user-developed Java classes and contains information required for their development.

Conventions

The notation conventions used in each chapter of this document are as follows:

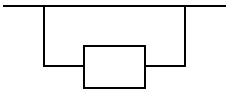
Chapter 1, "Complex Event Processing Language Reference" conventions

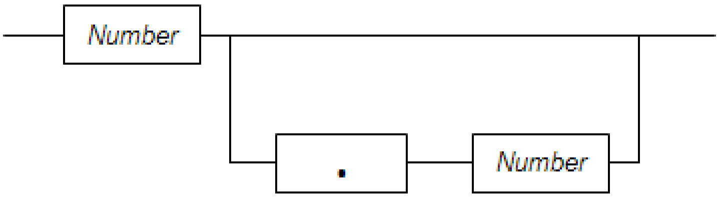
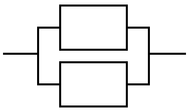
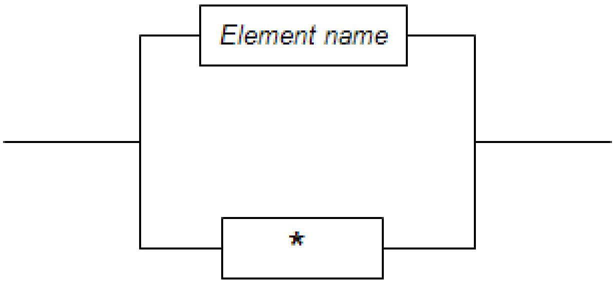
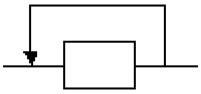
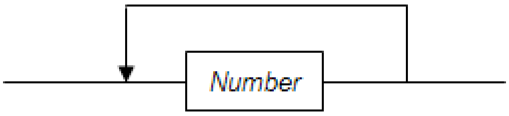
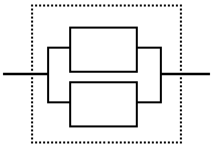
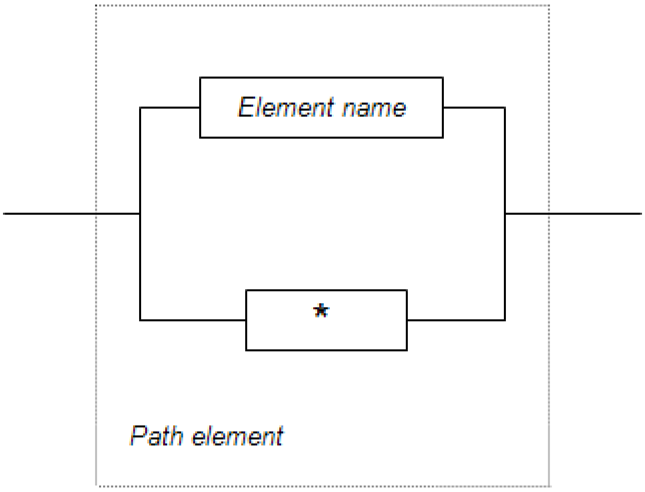
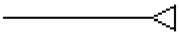
The following notation is used in the complex event processing language syntax:

- Square brackets, "[" and "]", indicate that the part enclosed by the brackets is optional.
- A vertical line, "|", indicates the need to make a selection from the items separated by the vertical line. If the boundary between the selection options and surrounding syntax is not clear, the list of selection options is enclosed in parentheses, "(" and ")".
- Three periods, "..." indicates a continuation of the previous part in a similar way.
- Variable information or content that can be modified is italicized and written in mixed case (for example: *newBkpDir*).
- If square brackets or parentheses are used as part of sentence structure, they are underlined "[" "]" "(" ")" to distinguish them from the square brackets and parentheses used as notation conventions.

Chapter 2, "Filter Rule Language Reference" conventions

The symbols used in the filter rule language format have the following meanings:

Symbol	Explanation
	The part branching below the line indicates elements that are optional. Example:

Symbol	Explanation
	
	The parts branching vertically in parallel indicate that one of these elements must be selected. Example: 
	The arrow line pointing backwards over the top of the line indicates that the element is repeated. Example: 
	Indicates that the multiple syntax elements are grouped. Example: 
	Indicates the end of the syntax.

Trademarks

- Adobe, Adobe Reader, and Flash are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Red Hat, RPM, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.
- Microsoft, Windows, MS, MS-DOS, Windows XP, Windows Server, Windows Vista, Windows 7, Excel, and Internet Explorer are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- Software AG and Terracotta, and all Software AG/Terracotta products, are either trademarks or registered trademarks of Software AG.
- Interstage, PRIMECLUSTER, ServerView, Symfoware, and Systemwalker are registered trademarks of Fujitsu Limited.
- Other company names and product names used in this document are trademarks or registered trademarks of their respective owners.

Note that registration symbols (TM or R) are not appended to system names or product names in this manual.

Export restrictions

If this document is to be exported or provided overseas, confirm legal requirements for the Foreign Exchange and Foreign Trade Act as well as other laws and regulations, including U.S. Export Administration Regulations, and follow the required procedures.

Copyright

```
February 2014: Second edition
October 2012: First edition
```

Contents

Chapter 1 Complex Event Processing Language Reference

This chapter explains the complex event processing rule descriptions for Complex Event Processing provided by Interstage Big Data Complex Event Processing Server (hereafter referred to as "BDCEP").

Complex Event Processing uses the complex event processing language to describe the processing of events.

The complex event processing language is an extended SQL-based language for processing events. Whereas SQL performs processing of database tables, the complex event processing language performs processing of event streams.

1.1 Basic Items in the Complex Event Processing Language

1.1.1 Event Stream Name

In the complex event processing language, an event stream name is expressed as the event types that are included in an event stream. In this manual, event stream name and event type (name) have similar meanings.

The following rules apply to event stream names:

- The first character must be alphabetic (a to z or A to Z).
- The second and subsequent characters can be alphabetic (a to z or A to Z), numerics (0 to 9), or underscores (_).
- The reserved words listed in the table at "1.1.5 Reserved Words" cannot be used regardless of case.

Note that event stream names are case-sensitive.

Complex Event Processing uses the "development asset ID" specified in an "event type definition" as the input event stream name. If the "development asset ID" does not conform to the above rules, enclosing it between backquote symbols (`) allows its use as an event stream name.

1.1.2 Event Properties

Events are normally constructed from multiple properties. Complex Event Processing handles event properties in the same way as table columns are handled in SQL.

The rules applying to event property names are the same as the event stream name rules.

Input events are in either CSV format or XML format, and Complex Event Processing uses the names shown below as input event property names. If these names do not conform to the above rules, enclosing them between backquote symbols (`) allows their use as a property name.

CSV format events

The `name` attributes in the "`column`" elements specified in the "event type definition"

XML format events

Subelement names of the root element in the XML definition ("`xmlSchema`" and "`root`" element) specified in the "event type definition"

1.1.3 Time Expression

The syntax below can be used to express time in the complex event processing language.

```
[yearPart] [monthPart] [weekPart] [dayPart] [hourPart] [minutePart] [secondsPart] [millisecondsPart]
```

These parts have the syntax below. No parts can be omitted.

```
yearPart: numeric (years | year)
monthPart: numeric (months | month)
weekPart: numeric (weeks | week)
dayPart: numeric (days | day)
hourPart: numeric (hours | hour)
minutePart: numeric (minutes | minute | min)
secondsPart: numeric (seconds | second | sec)
millisecondsPart: numeric (milliseconds | millisecond | msec)
```

Point

- Either the singular or plural form can be used for the unit of each *part*, but that is merely for description convenience. Complex Event Processing does not check English grammar.
- In the *monthPart*, one month is equivalent to 30 days.

Information

As with Java, Complex Event Processing internally handles time as long values indicating the number of milliseconds from January 1, 1970, 00:00:00 GMT.

1.1.4 Comments

In the complex event processing language, comments use the same two formats as in Java.

```
// comment

/* comment */
```

1.1.5 Reserved Words

The keywords below are reserved words in the complex event processing language, and cannot be used as event stream names (event types) and event property names.

after	else	is	not	set
all	end	istream	null	snapshot
and	escape	join	offset	some
any	events	last	on	sql
as	every	lastweekday	or	start
asc	every-distinct	left	order	stddev
at	exists	like	outer	sum
avedev	expression	limit	output	terminated
avg	false	match_recognize	partition	then
between	first	matched	pattern	true
by	for	matches	prev	typeof
case	from	max	prevcount	unidirectional
cast	full	measures	prevtail	until

coalesce	group	median	prevwindow	update
context	having	merge	prior	using
count	hour	metadatasql	regexp	variable
create	hours	millisecond	retain-intersection	week
current_timestamp	in	milliseconds	retain-union	weekday
dataflow	index	min	right	weeks
day	initiated	minute	rstream	when
days	inner	minutes	schema	where
define	insert	month	sec	while
delete	instanceof	months	second	window
desc	into	msec	seconds	year
distinct	irstream	new	select	years

1.1.6 Data Type

The complex event processing language can handle the same data types as Java character strings and primitive data types. Data type descriptions in the complex event processing language are not case-sensitive.

Data type	Explanation	Method for expressing constants (literals)
string	Character string	Enclose with double quotation marks (") or single quotation marks ('). If you want to include double quotation marks (") or single quotation marks (') in a character string, you can use either the method in which the backslash symbol (in a Japanese language environment, the Yen symbol) is placed before a double quotation mark or single quotation mark, or the Unicode method (double quotation mark is \u0022, and single quotation mark is \u0027).
char/character	Character	There is no constant (literal) expression indicating a single character.
bool/boolean	Boolean value	true or false
byte	8-bit signed integer	Express using 0x followed by 2 hexadecimal characters.
short	16-bit signed integer	There is no constant (literal) expression indicating 16-bit signed integers.
int/integer	32-bit signed integer	Enter the integer value as is.
long	64-bit signed integer	Add L or l after the integer value.
float	32-bit float	Add F or f after the numeric value.
double	64-bit double precision float	Enter as is a numeric value, including the decimal point or the exponent portion (specify using E or e), or add D or d at the end.

The event properties entered to Complex Event Processing are converted to the complex event processing language data types shown below in accordance with the data types of the elements defined at "xmlSchema" element in the "event type definition".

XML schema standard primitive data type, derived data type		Complex event processing language data type
Primitive data type	string	string
	boolean	bool
	decimal	double

XML schema standard primitive data type, derived data type		Complex event processing language data type
		(If fractionDigit is specified as 0 in the derived data type: int)
	float	float
	double	double
	duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName, NOTATION	string
Derived data type	string derived data type (normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName, ID, IDREF, IDREFS, ENTITY, ENTITIES)	string
	integer, nonPositiveInteger, negativeInteger, nonNegativeInteger, positiveInteger	int
	long, unsignedLong	long
	int, unsignedInt	int
	short, unsignedShort	short
	byte, unsignedByte	byte

Note

- Even if the integer type is derived from the decimal type and the long type is derived from the integer type in the XML schema data types, the decimal and integer types both correspond to the int type (the effective number of digits is less than for the long type) in the corresponding complex event processing language data type. Therefore, if long values are used, do not use the integer or decimal type in XML schema.
- In addition to true and false, 1 and 0 are permitted as XML schema Boolean values. In the complex event processing language, only true and false can be used as bool (Boolean) values, so do not use 1 and 0 as Boolean values.

1.1.7 Complex Event Processing Statement

The statements described using the complex event processing language contain the items in the table below. The SELECT statement is the core of rule descriptions. Use named windows if you want to cache event data in memory and use it for subsequent processing.

Category	Complex event processing statement type	Explanation
Event stream queries	SELECT statement	The complex event processing statement at the core of rule descriptions. It performs queries to event streams.
Named window operations	CREATE WINDOW statement	Generates named windows for event data caching.
	ON SELECT statement	Performs queries to named windows when an event occurs.
	ON UPDATE statement	Updates event data in named windows when an event occurs.
	ON DELETE statement	Deletes events in named windows when an event occurs.
	ON MERGE statement	Performs event addition, update, and deletion operations all at once for named windows when an event occurs.

Point

By placing a semicolon (`;`) at the end of complex event processing statements, multiple complex event processing statements can be described in a complex event processing rule described in the rule definitions.

1.1.8 Annotation

Annotations attach additional information to individual complex event processing statements. Describe an annotation before the complex event processing statement to which you want to add information.

Annotation type	Notation	Explanation
Name	@Name("*name*") or @Name('*name*')	Attaches *name* to a complex event processing statement. Attaching a unique name makes it easier to distinguish output information during debugging. When you attach the same name to multiple complex event processing statements, two hyphens "--" and a number will be automatically appended to make unique names. If the @Name annotation is not specified, a unique name among the rules, such as "0b0562a2-56e7-4cf3-a520-cb1e16ef2992", is automatically assigned.
SoapListener	@SoapListener("*listenerDefinition*") or @SoapListener('*listenerDefinition*')	Associates a listener definition with a SELECT statement. For *listenerDefinition*, specify the "development asset ID" specified at "listener definition". This passes the SELECT statement output as a SOAP message to the user-developed Web service specified in the listener definition.
DebugLogListener	@DebugLogListener	Outputs logs for debugging complex event processing statements.
LoggingListener	@LoggingListener(table="*logStorageArea*", properties="*propertyNamesToBeOutput*") or the same format but with single quotation marks (') instead of double quotation marks (")	Logs the complex event processing statement output results to *logStorageArea*. Specify the output destination of the Hadoop system in *logStorageArea* using a full path. Even if events are to be recorded in the engine log, specify with a virtual path name beginning with a slash, such as "*/eventName*", to distinguish events. In *propertyNamesToBeOutput*, specify the property names to be output by the complex event processing statement, separated by commas.
CustomListener	@CustomListener(mainClass="*userDevelopedJavaClassName*"[, args={"*argument1*", "*argument2*", ...}]) or the same format but with single quotation marks (') instead of double quotation marks (")	Passes the output of the complex event processing statement to a user-developed Java class. In *userDevelopedJavaClassName*, specify the name of the Java class that accepts the result of the complex event processing statement. Specify this name in FQCN format (which includes the package name). The CustomListener interface must have been implemented for this Java class.

Annotation type	Notation	Explanation
		Specify *argument1* and *argument2* to pass certain arguments to a user-developed Java class. If there is no need to pass arguments, you can omit args.

Information

Annotation execution sequence

If the @SoapListener, @DebugLogListener, @LoggingListener, and @CustomListener annotations are specified simultaneously for one complex event processing statement, the output processing of each annotation is executed in the sequence in which they are specified.

```
@SoapListener('LISTEN01')
@DebugLogListener
select gatewayId, value from EVENT_01;
```

In the above example, the sequence of output execution is @SoapListener, then @DebugLogListener.

1.2 SELECT Statement

The SELECT statement of the complex event processing language describes continuous queries to the event stream, similar to the way queries to database tables are coded in SQL SELECT statements.

An overview of the SELECT statement syntax is shown below. Syntax details are shown under the various clauses.

Syntax:

```
[annotation]
[insert into insertDefinition]
select propertyAndExpressionList
from eventStreamDefinitionAndItsJoin
[where conditionExpression]
[group by groupingExpressionList]
[having groupingCondition]
[output outputDefinition]
[order by sortExpressionList]
[limit numberOfRows]
```

1.2.1 SELECT Clause

For the SELECT clause, specify all properties, or specify *property* or *expression* lists. The SELECT statement output event consists of the items specified here.

Syntax:

```
select [distinct] (* | (property | expression) [as name])
 [, (property | expression) [as name]] [, ...]
```

If an asterisk (*) is specified, this is interpreted as all properties being specified.

For AS, a *name* (alias) can be attached to a *property* or *expression*.

Output of duplicate output events can be suppressed by specifying DISTINCT.

1.2.2 FROM Clause

For the FROM clause, specify one or more event streams or named windows as the input.

Syntax:

```
from eventStreamDefinition [as name] [unidirectional] [retain-union | retain-intersection]
 [,eventStreamDefinition [as name]] [, ...]
```

The *`eventStreamDefinition`* is a filter-based event stream definition or a pattern-based event stream definition. For AS, a *`name`* (alias) can be attached to a stream definition.

Refer to "1.2.11 JOIN Clause" for information on `UNIDIRECTIONAL`.

Note

Notes on using Virtual Data Windows

For a simple SELECT statement that specifies Virtual Data Window in a FROM clause or a join that does not specify UNIDIRECTIONAL, an event inserted into the Virtual Data Window by the INSERT INTO clause using the complex event processing rules on the same CEP engine propagates. However, an event (cache entry) added from a Terracotta application or a different CEP engine does not propagates. To access cache data, use an ON SELECT statement, a subquery, or a join that specifies UNIDIRECTIONAL.

The following example shows a simple SELECT statement:

```
select W.high, W.low from MarketWindow;
```

The following example shows a join that does not specify UNIDIRECTIONAL:

```
select W.high, W.low from TicketEvent.std:lastevent() as Input, MarketWindow as W
    where W.code = Input.code;
```

If the INSERT INTO clause inserts an event into `MarketWindow`, the inserted event is reported to the SELECT statements. However, events added outside the CEP engine are not reported.

1.2.2.1 Filter-based Event Stream Definition

Syntax:

```
eventStreamName [( filterCondition )] [.view] [.view]
```

The *`eventStreamName`* is an event type name, an event stream name specified in the INSERT INTO clause of a different complex event processing statement, or the name of a named window.

For the *`filterCondition`* specification, use properties and operators, functions (except for the aggregate function), or similar, to describe the condition. Commas (`,`) can be used in a *filterCondition* with the same meaning as the `AND` logical operator.

In addition, event expiry policy specifications (if a data window view) and data derivations can be specified by specifying a *`view`*, as described at "1.7 Views".

If more than one *`view`* is specified, the common parts of those views are retained by default (same as the `RETAIN-INTERSECTION` specification). The union parts of multiple views can be retained by specifying `RETAIN-UNION`.

1.2.2.2 Pattern-based Event Stream Definition

Syntax:

```
pattern [ patternExpression ] [.view] [.view]
```

Pattern-based event streams are specified by describing a *`patternExpression`* inside square brackets, "`[`" and "`]`", following the PATTERN keyword. A *`view`* can also be specified for a pattern in the same way as for filter-based event stream definitions.

Refer to "1.4 Patterns" for information on how to describe a *`patternExpression`*.

1.2.3 WHERE Clause

The WHERE clause can be used to specify the joining conditions that apply when multiple event streams are joined or to specify event filtering conditions.

Syntax:

```
where conditionExpression
```

The comparison operators =, <, >, >=, <=, `!=`, <>, `IS NULL`, and `IS NOT NULL`, and logical combinations using AND and OR are supported for a WHERE clause *`conditionExpression`*.

Note

Notes on using Virtual Data Windows

The WHERE clause, which can be used for accessing information stored in a Virtual Data Window, must contain a condition for uniquely identifying cache entries. This condition uses "=" to perform a comparison with the key property specified using vdw:ehcache(). An example is shown below.

Definition example

In the following example, `W.code = T.code` is valid, because it uniquely identifies a cache entry.

```
create window MarketWindow.vdw:ehcache("MARKET", "code") as (code string, high int, low int);

on TicketEvent as T
    select W.high, W.low from MarketWindow as W
        where W.code = T.code and ( T.price > W.high or T.price < W.low);
```

The table below shows valid and invalid definition examples of using the above rule to change the WHERE clause only.

No	Definition example	Valid/ Invalid	Explanation
1	`where code = '1111'`	Valid	Valid because the condition can uniquely identify a cache entry by using "=" to perform a comparison with the key property
2	`where ( T.price > W.high or T.price < W.low) and W.code=T.code`	Valid	Preceded by a different condition but valid because it includes "=" for performing a comparison with the key property and can uniquely identify a cache entry
3	`where high = 1000`	Invalid	Invalid because a comparison using "=" is not performed for the key property
4	`where code > '1111'`	Invalid	Invalid because an operation other than "=" is performed for the key property
5	`where code = '1111' or high = 1000`	Invalid	Includes "=" for performing a comparison with the key property but is invalid because there is an OR condition and a cache entry cannot be uniquely identified

1.2.4 GROUP BY Clause

The GROUP BY clause splits the output of a complex event processing statement into groups. The output can be split in accordance with event properties or in accordance with expression calculation results.

Syntax:

```
group by groupingExpression [,groupingExpression] [, ...]
```

In a *groupingExpression*, specify the property, or an expression that includes properties, that is the basis for splitting into groups. The *groupingExpression* cannot include the aggregate function. Event properties used by the aggregate function within a SELECT clause can also not be included in a *groupingExpression*.

1.2.5 HAVING Clause

Like a WHERE clause specification in relation to a SELECT clause, the HAVING clause specifies a *groupingCondition* for a GROUP BY clause. Whereas a WHERE clause cannot include the aggregate function, the HAVING clause can include the aggregate function.

Syntax:

```
having groupingCondition
```

1.2.6 OUTPUT Clause

The OUTPUT clause controls the event output speed and suppresses output.

The syntax below performs output each time the specified *number* of output events have arrived.

Syntax:

```
output [after suppressionDefinition]
[[all | first | last | snapshot] every number events]
```

The syntax below performs output each time the specified *time* elapses.

Syntax:

```
output [after suppressionDefinition]
[[all | first | last | snapshot] every time]
```

The syntax below specifies an output schedule similar to crontab.

Syntax:

```
output [after suppressionDefinition]
[[all | first | last | snapshot] at ( minute, hour, day, month, dayOfWeek[, seconds] )]
```

Refer to the TIMER:AT explanation under "1.4.11 Time-based Observer" for information on schedule specification (*minute*, *hour*, *day*, *month*, *dayOfWeek*, *seconds*).

The ALL keyword is the default and specifies to output all targeted events. The FIRST keyword specifies to output only the first event. The LAST keyword specifies to output only the last event. The SNAPSHOT keyword specifies to output the calculation results that take into account all events in the specified view.

The syntax below specifies AFTER and the *suppressionDefinition*.

Syntax:

```
output after (time | number events ) [...]
```

From the start of complex event processing statement processing until the specified *time* has elapsed, or until the specified *number* of output events arrive, all output events are discarded without being output.

1.2.7 ORDER BY Clause

The ORDER BY clause orders output events in accordance with a property or in accordance with the values of expressions that include properties.

Syntax:

```
order by sortExpression [asc | desc] [,sortExpression [asc | desc]] [, ...]
```

In the *sortExpression*, specify the property, or the expression that includes properties, on which ordering is to be based.

ASC and DESC specify whether ascending or descending order is used.

1.2.8 LIMIT Clause

The LIMIT clause restricts the number of output events.

Syntax:

```
limit numberOfRows [offset offsetNumber]
```

Only the number of events specified at *numberOfRows* are output.

By specifying an *offsetNumber*, the number of rows of events that are output starts from a specified position rather than from the start of the results.

As with SQL, the syntax below can also be used.

Syntax:

```
limit offsetNumber [,numberOfRows]
```

1.2.9 INSERT INTO Clause

The INSERT INTO clause can be used when making SELECT statement results usable as an event stream, when inserting into a named window, and when merging multiple event streams.

Syntax:

```
insert into eventStreamName [ ( propertyName [,propertyName] [, ...] ) ]
```

At *eventStreamName*, specify the identifier used as the name of the event stream that outputs results. This can also be the event type names in the event stream. This *eventStreamName* can also be used when describing processing in other complex event processing statements. *propertyName* can also be specified at the same time as an *eventStreamName*.

The results of a SELECT statement can be inserted in a named window by specifying the name of a named window in an *eventStreamName*.

Streams can be merged by specifying an existing event stream name in an *eventStreamName*.

Note

Notes on using Virtual Data Windows

A Terracotta cache holds only one event (cache entry) for the value of a key property. Therefore, if a new event is inserted into a Virtual Data Window using an INSERT INTO clause and the cache already contains an event that has the same key property value, the event is updated using the new event.

1.2.10 Subqueries

The SELECT statement can be written as a subquery within a complex event processing statement. A subquery can be written in a SELECT clause, a WHERE clause, and in an event stream and pattern filter expression. Describe the subquery enclosed between parentheses, "(" and ")".

A data window or another view must be specified in the event stream definition in the subquery. Only SELECT, FROM, and WHERE clauses can be described in subqueries. The GROUP BY clause, HAVING clause, JOIN, and OUTPUT clause cannot be specified.

The table below shows the keywords that can be used when describing a subquery in a WHERE clause condition.

Keyword	Syntax	Explanation
EXISTS, NOT EXISTS	[not] exists (*subquery*)	If the *subquery* returns at least one row, the EXISTS condition is TRUE. If the *subquery* returns no rows, the NOT EXISTS condition is TRUE.
IN, NOT IN	*expression* [not] in (*subquery*)	If at least one value returned by the *subquery* matches the *expression* value, the IN condition is TRUE. If no values returned by the *subquery* match the *expression* value, the NOT IN condition is TRUE.
ANY, SOME	*expression operator* any (*subquery*) expression *operator* some (*subquery*)	The *operator* evaluates the *expression* value and the *subquery* result, and is TRUE if even one is TRUE. The *subquery* only need return one property. The *operator* is either =, !=, <>, <, <=, >, or >=. ANY and SOME have the same meaning.
ALL	*expression operator* all (*subquery*)	The *operator* evaluates the *expression* value and the *subquery* result, and is TRUE if ALL are TRUE. The *subquery* only needs to return one property. The *operator* is either =, !=, <>, <, <=, >, or >=.

1.2.11 JOIN Clause

The same operation as SQL JOIN can be described for an event stream in complex event processing language. In addition to event streams, named windows and relational database (RDB) data can also be joined.

Syntax:

```
... from eventStreamDefinition [as name] [unidirectional]
  ((left|right|full) outer | inner) join eventStreamDefinition [as name] [unidirectional]
  on property = property [and property = property] [and ...]
[ ((left|right|full) outer | inner) join eventStreamDefinition on ...] ...
```

For JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, and INNER JOIN can be specified. Use the ON clause to specify the *property* for joining each event stream.

In addition, INNER JOIN can be executed by tying event stream definitions together with commas (,). In this case, an ON clause need not be specified and the condition can be specified in a WHERE clause.

Each of the event streams being joined must specify a data window view or another view. A view does not need to be specified for event streams for which UNIDIRECTIONAL is specified, and named windows.

Generally, JOIN is executed for the event streams in the FROM clause regardless of which of the event streams the event arrives at. If the UNIDIRECTIONAL keyword is specified, JOIN is executed only when an event arrives at that event stream. The UNIDIRECTIONAL keyword can be specified for only one event stream.

1.2.12 Database Access

The results of an SQL query sent to a relational database (RDB) can be joined to an event stream. Specify this using the following syntax in a FROM clause:

Syntax:

```
sql:databaseName [" sqlQuery "] or
sql:databaseName [' sqlQuery ']
```

In *databaseName*, specify the "development asset ID" specified in the RDB reference definition.

Enclose *sqlQuery* in double quotation marks (") or single quotation marks ('), and enclose this specification in square brackets "[" and "]".

Alternate parameters can be included in *sqlQuery*. Specify alternate parameters in the ${*expression*} format. The expression is evaluated when the statement is executed.

Note

- Minimize RDB referencing, because it may cause a decline in the performance of Complex Event Processing.
- Multibyte characters cannot be used for definition names in a relational database such as table names and table item names to be referenced from the complex event processing language. Multibyte characters can be used for item values.
- To specify a nonnumeric literal enclosed in single quotation marks (') in an SQL query that is further enclosed in single quotation marks ('), use the escape notation (\') or the Unicode notation (\u0027).

Example

Example of using relational database referencing

This example uses relational database referencing in a complex event processing rule (SELECT statement).

```
@Name("PutRecommend")
@SoapListener("soap-001")
select tvevnt.gatewayId as gatewayId,
    case when db.DRY_FUNC = '1' then 'USE_DRY_FUNC' else 'HANG_LAUNDRY_INSIDE' end as recommendId
    from TVControlRain as tvevnt,
        sql:app_db[ 'SELECT DRY_FUNC FROM PRODUCTFUNC_TBL WHERE HGW_ID=${tvevnt.gatewayId}' ] as db;
```

- The development asset ID of the RDB reference definition to be used is set to "app_db".
- The relational database table being referenced is "PRODUCTFUNC_TBL".
- The alternate parameter "${tvevnt.gatewayId}" is specified in the condition for referencing the relational database table.
- The alias "db" is assigned to the reference results and is used in the SELECT statement.

1.2.13 Data Type Mapping

Complex event processing cannot be specified using SQL data type when RDB referencing is used.

The JDBC driver temporarily converts SQL data types to generic SQL types (java.sql.Types), and then to complex event processing language data types. The following table shows the complex event processing language data types corresponding to the generic SQL types.

Refer to the manual of the relevant JDBC driver for information on the generic SQL types corresponding to each SQL data type. Refer to "1.1.6 Data Type" for information on the data types for the complex event processing language.

Generic SQL types defined using java.sql.Types	Complex event processing language data type
CHAR VARCHAR LONGVARCHAR	string
BOOLEAN BIT	bool/boolean
SMALLINT	short
INTEGER	int/integer
BIGINT	long
REAL	float
DOUBLE FLOAT	double

Some generic SQL types that do not appear in the above correspondence table can be used if you convert the type.

General SQL types defined using java.sql.Types	Type conversion method	Complex event processing language data type
NUMERIC DECIMAL	Cast using bool, short, byte, int, long, float, or double. If the data size is greater than the conversion destination data type, data may be lost.	Type specified using cast
DATE TIME TIMESTAMP	Add the toString() method.	string

Other SQL types that do not appear above cannot be used with the complex event processing language.

Example

Example of mapping between Symfoware Server (referencing with native interface) data types and complex event processing language data types

Symfoware Server (referencing with native interface) SQL data types and complex event processing language data types are converted as shown below.

Refer to the manual of Symfoware Server for information on conversion between Symfoware Server SQL data types and generic SQL types.

Category	Symfoware Server SQL data type	Complex event processing language data type
Exact numeric type	SMALLINT	short
	INTEGER	int/integer
	NUMERIC	Cast using bool, short, byte, int, long, float, or double. If the data size is greater than the cast destination type, data may be lost. Therefore, select the type to be specified using cast according to the data size.
Approximate numeric type	REAL	float
	FLOAT(p) p=1 to 23	float
	FLOAT(p) p=24 to 52	double

Category	Symfoware Server SQL data type	Complex event processing language data type
	DOUBLE PRECISION	double
Datetime type	DATE	Add the toString() method.
	TIME	
	TIMESTAMP	
String type	CHARACTER	string
	VARCHAR	string
Language string types	NCHAR	string
	NCHAR VARYING	string
Other	BLOB	Not supported
	ROW_ID	
	INTERVAL YEAR TO MONTH	
	INTERVAL YEAR	
	INTERVAL MONTH	
	INTERVAL DAY TO HOUR	
	INTERVAL DAY TO MINUTE	
	INTERVAL DAY TO SECOND	
	INTERVAL DAY	
	INTERVAL HOUR TO SECOND	
	INTERVAL HOUR	
	INTERVAL MINUTE TO SECOND	
	INTERVAL MINUTE	
	INTERVAL SECOND	

Example

The following example shows complex event processing for outputting "M_DATE" when the input event "E1", which has the "id" and "e_date" string types as properties, is input and the "M_DATE" date in the "MY_DB" database is earlier than "e_date".

```
@Name('EPL1')
select E1.id, db1.M_DATE.toString() from E1, sql:my_db ['SELECT M_DATE FROM MY_TBL'] as db1
  where db1.M_DATE.toString()>E1.e_date;
```

The following example shows complex event processing for outputting "PRICE" when the input event "E2", which has the "id" string type and the "value" int type as properties, is input and the value of "PRICE" in the "MY_DB" database is greater than "value".

```
@Name('EPL2')
select E2.id, cast(db2.PRICE, double) from E2, sql:mydb_db ['SELECT PRICE FROM MY_TBL'] as db2
  where cast(db2.PRICE, double)>E2.value;
```

1.3 Named Window Operations

Data windows, referred to as named windows, can be created in complex event processing language to cache event data for implementing processing that is similar to other event streams.

1.3.1 CREATE WINDOW Statement

Generate a named window. There are two generation methods: generation from an existing event type, and generation in which a property name and type are defined.

1.3.1.1 Generation from Existing Event Type

Syntax:

```
create window windowName.view
  [as] [select property [, ...] from] eventTypeOrWindowName
  [insert [where filterExpression]]
```

To generate a named window, specify the name of the named window being created (*windowName*), and specify one or more data window views (*view*). An existing event type (event stream) name cannot be used for the *windowName*.

Use the SELECT clause to specify to quote *property* from an existing *eventTypeOrWindowName.*

If quoting properties from an existing named window, the INSERT clause can be used to fetch data from an existing named window when the new named window is generated. Filtering conditions can be specified at *filterExpression*.

1.3.1.2 Generation with Property Name and Type Specification

Syntax:

```
create window windowName.view [as] ( propertyName propertyType [,propertyName propertyType] [, ...] )
```

At *propertyName* and *propertyType*, specify the property names and data types of the events to be entered in the named window.

1.3.1.3 Virtual Data Window Generation

A Virtual Data Window that references the Interstage Terracotta BigMemory Max (hereafter referred to as "Terracotta") cache can be created by specifying vdw:ehcache() in the CREATE WINDOW statement. There are two methods of specification:

- Setting type information in the event type definition
- Setting by direct description of type information

Setting type information in the event type definition

Syntax:

```
create window windowName.vdw:ehcache("cacheName", "nameOfPropertyCorrespondingToCacheKey") as
eventTypeDevelopmentAssetID;
```

This definition method uses the event type definition. Event type definitions defined in XML format cannot be specified.

Only CSV format event type definitions can be specified. (An event is cached as a java.util.HashMap<String,Object> object.)

As the arguments of `vdw:ehcache()`, specify the name of the cache accessed by a Virtual Data Window and the name of the property corresponding to the cache key.

Setting by direct description of type information

Syntax:

```
create window windowName.vdw:ehcache("cacheName", "nameOfPropertyCorrespondingToCacheKey") as
(propertyName1 type1, propertyName2 type2, ...);
```

This definition method specifies type information directly. Refer to "1.3.1.4 Supported Terracotta Cache Formats" for information on types that can be specified for each property.

Example

Virtual Data Window creation example

```
create window CustomerWindow.vdw:ehcache("CustomerCache", "id") as (id string, name string, address
string);
```

This Virtual Data Window creation example shows direct description of type information.

This example references the Terracotta cache `CustomerCache`. The key is the `id`. The name of the window being generated is `CustomerWindow`, and the properties are as follows:

Property name	Type
`id`	string
`name`	string
`address`	string

Note

Do not specify another view at the same time as a Virtual Data Window

Do not specify another view such as win:length(1) at the same time as the vdw:ehcache() specification in a CREATE WINDOW statement that defines a Virtual Data Window. Specifying another view does not cause a syntax error, but the view specification cannot be used to operate Terracotta cache data (even if you specify win:length(1), the number of events in the cache will not be "1").

1.3.1.4 Supported Terracotta Cache Formats

A Terracotta cache consists of key-value pairs. The cache referenced by BDCEP must have the following configuration:

Type specified for the key	Type used for the value
java.lang.String	java.util.HashMap<java.lang.String, java.lang.Object>

Each property to be specified in a complex event processing rule corresponds to each "HashMap" element above.

The table below shows the type that is compatible for the value of each "HashMap" element and the corresponding type in complex event processing rules.

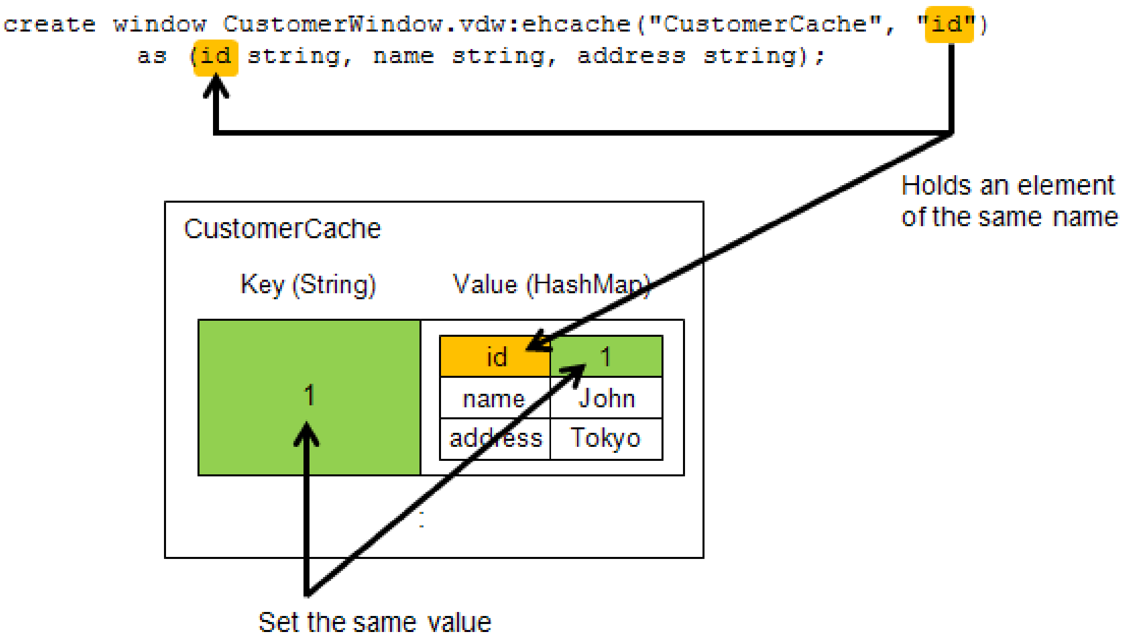
Type used in each HashMap element	Corresponding type in complex event processing rules
java.lang.String	string
java.lang.Character	char/character
java.lang.Boolean	bool/boolean
java.lang.Byte	byte
java.lang.Short	short
java.lang.Integer	int/integer
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double

The Terracotta cache key and "HashMap" to be specified for the value must have the following relationship:

- Ensure that the name of the property to be specified as the cache key is the same as the key of the corresponding "HashMap" element.

- Set the value of the above "HashMap" element as the value of the cache key.

The following figure illustrates this relationship.



1.3.2 ON SELECT Statement

Execute a query to a named window when an event occurs.

Syntax:

```
on eventType[(filterCondition)] [as name]
  [insert into insertDefinition]
  select propertyAndExpressionList
  from windowName [as name]
  [where conditionExpression]
  [group by groupingExpressionList]
  [having groupingCondition]
  [order by sortExpressionList]
```

This syntax executes a query (SELECT statement) to the named window specified at *windowName* when an event of *eventType* occurs. A *filterCondition* can also be specified for the event. The AS keyword can be used to assign a *name* (alias).

The explanations for the other parts are the same as for the SELECT statement.

1.3.3 ON UPDATE Statement

Update an event in a named window when an event occurs.

Syntax:

```
on eventType [(filterCondition)] [as name]
  update windowName [as name]
  set property = expression [,property = expression] [, ...]
  [where conditionExpression]
```

When an event of *eventType* occurs, this syntax updates the value of the *property* specified in the SET clause to the value of *expression* for events in the named window specified at *windowName*. If a WHERE clause *conditionExpression* is specified, only events that match that condition are targeted.

1.3.4 ON DELETE Statement

Delete an event from a named window when an event occurs.

Syntax:

```
on eventType [(filterCondition)] [as name]
  delete from windowName [as name]
  [where conditionExpression]
```

This syntax deletes events in the named window specified at `windowName` when an event of `eventType` occurs. If a WHERE clause `conditionExpression` is specified, only events that match that condition are targeted.

1.3.5 ON MERGE Statement

Perform event addition, update, and deletion operations on a named window as a batch when an event occurs.

Syntax:

```
on eventType[(filterCondition)] [as name]
  merge [into] windowName [as name]
  [where conditionExpression]
    when [not] matched [and condition]
      then (
        insert [into insertDefinition]
          select propertyAndExpressionList
          [where conditionExpression]
        |
        update set property = expression [,property = expression] [, ...]
          [where conditionExpression]
        |
        delete
          [where conditionExpression]
      )
      [then (insert|update|delete) ...] [then ...]
    [when ...  then ...] [...]
```

This syntax executes various types of operations on the named window specified at `windowName` when an event of `eventType` occurs.

The THEN clause action (INSERT, UPDATE, DELETE) specified after "`when matched`" or "`when not matched`" is executed in accordance with whether or not events are in the named window (or, if there is a WHERE clause specified after the MERGE clause, in accordance with that `conditionExpression`). A further `condition` can be added to "`when [not] matched`" by using AND. With "when matched", either INSERT, UPDATE, or DELETE can be specified as the THEN clause action. With "`when not matched`", only INSERT can be specified.

1.4 Patterns

An event pattern matches when a single event or multiple events that match the pattern definition occur. Patterns can also be described based on time.

Pattern expressions consist of pattern atoms and pattern operators.

Pattern atoms contain filter expressions and observers for time-based events. Filter expressions specify filter conditions for event streams.

There are the following four types of pattern operators:

- Operators that control repetitions of the subexpressions that comprise the pattern expression: EVERY, EVERY-DISTINCT, [COUNT], UNTIL
- Logical operators: AND, OR, NOT
- Time operator that operates event sequences: `->` (followed-by)
- Guards specified in WHERE clauses for controlling subexpression life cycles: TIMER:WITHIN, TIMER:WITHINMAX, WHILE expression

1.4.1 Pattern Operators and Priorities

Pattern operators have the following calculation priorities:

Priority	Operator type	Operator
1	Monadic operator	EVERY NOT EVERY-DISTINCT

2	Guard	WHERE TIMER:WITHIN
		WHERE TIMER:WITHINMAX
		WHILE (*expression*)

3	Repetitive	[COUNT] UNTIL
4	Logical conjunction	AND
5	Logical disjunction	OR
6	followed-by	->

By using parentheses, "(" and ")", to enclose the subexpressions that comprise a pattern expression, calculation of the enclosed parts can be given priority.

1.4.2 EVERY Operator

The EVERY operator specifies to evaluate a *subexpression* repeatedly. If the EVERY operator is not used, pattern evaluation ends at the point when the *subexpression* is evaluated once (as either TRUE or FALSE).

Syntax:

```
every subexpression
```

1.4.3 EVERY-DISTINCT Operator

The EVERY-DISTINCT operator treats patterns that return the same value for the specified expression as duplicates and excludes them from pattern evaluation.

Syntax:

```
every-distinct ( distinctExpression [,distinctExpression] [, ...] [, period] ) subexpression
```

Patterns having the same value for the *distinctExpression* are excluded as duplicates after being detected once. If a time sets a *period*, information held internally for duplicate exclusion purposes is discarded after the specified time has elapsed, and the pattern is once again targeted for evaluation.

1.4.4 Repetition Operator

When the pattern *subexpression* is evaluated *repetitionCount* times as TRUE, the repetition operator is evaluated as being TRUE.

Syntax:

```
[ repetitionCount ] subexpression
```

For the *repetitionCount*, specify a positive integer enclosed in square brackets, "[" and "]".

1.4.5 UNTIL Operator

A repetition end condition can be specified using the UNTIL operator.

Syntax:

```
[ range ] subexpression until endPatternExpression
```

If a `range` is not specified, `subexpression` evaluation is repeated until the `endPatternExpression` becomes TRUE. At that point the expression becomes TRUE.

If a `range` is specified, the `subexpression` must become TRUE within the count specified as the *range*.

The syntax for the range is as follows:

Syntax:

```
[minimumNumber] : [maximumNumber]
```

At least one, either the `minimumNumber` or the `maximumNumber`, must be specified.

The `minimumNumber` is the minimum required number of TRUE `subexpression` repetitions for this expression to become TRUE.

When the `maximumNumber` of repetitions is reached, the expression is evaluated as being TRUE and `subexpression` evaluation stops.

1.4.6 AND Operator

If the `subexpression` on both sides of the AND operator are TRUE, the entire pattern is TRUE.

Syntax:

```
subexpression and subexpression
```

1.4.7 OR Operator

If either the `subexpression` before or after the OR operator is TRUE, the entire pattern is TRUE.

Syntax:

```
subexpression or subexpression
```

1.4.8 NOT Operator

The NOT operator negates the `subexpression` value.

Syntax:

```
not subexpression
```

1.4.9 Followed-by Operator

The followed-by operator (`->`) specifies that, after the `subexpression` on the left side is evaluated as TRUE, the `subexpression` on the right side is evaluated for event matching.

Syntax:

```
subexpression -> subexpression
```

1.4.10 Pattern Guard

The pattern guard controls a subexpression by specifying a condition in a WHERE clause or a WHILE clause.

The pattern guards available for specification in the WHERE clause are TIMER:WITHIN and TIMER:WITHINMAX. A WHERE clause that specifies a pattern guard is different from a WHERE clause that specifies event filtering in complex event processing language.

Syntax:

```
timer:within( timeExpression )
```

If the pattern expression does not become TRUE within the specified time, evaluation of the expression ends. In the *timeExpression*, specify either a time representation or an expression providing the number of seconds.

Syntax:

```
timer:withinmax( timeExpression, maximumNumberExpression )
```

In addition to the TIMER:WITHIN processing, the number of matches are counted. Evaluation of the expression ends when either the time specified in the *timeExpression* has elapsed or when the number specified in the *maximumNumberExpression* is reached.

The pattern guard specified by the WHILE clause evaluates the *guardExpression* for each pattern detected, and evaluation of the pattern expression ends at the point when it is evaluated as being FALSE.

Syntax:

```
while ( guardExpression )
```

Any expression that returns Boolean (TRUE or FALSE) can be written in a *guardExpression*.

1.4.11 Time-based Observer

Time-based observers observe time-based events based on the internal timer of the complex event processing engine.

Two observers, TIMER:INTERVAL and TIMER:AT, are available.

Observer	Syntax	Explanation
TIMER:INTERVAL	`timer:interval(`*timeExpression*`)`	Waits the specified time. In the *timeExpression*, specify either a time representation or an expression providing the number of seconds.
TIMER:AT	`timer:at(`*minuteSpecification*, *hourSpecification*, *daySpecification*, *monthSpecification*, *dayOfWeekSpecification* `[,`*secondsSpecification*`])`	Has a function like the Unix crontab command. The expression becomes TRUE at the specified time. An asterisk (`*`) can be specified as a wild card in each *specification*. A range can be specified by using a colon (`:`) to tie together a lower limit and an upper limit. The division expression `*/x` indicates that the value is enabled after every *x*. Specifications in accordance with combinations are possible by enclosing specifications in square brackets, "`[`" and "`]`", and by comma (`,`) separation.

The table below shows the values that can be specified for each TIMER:AT *specification*.

Specification location	Specifiable value	Specifiable keyword
minuteSpecification	0-59	
hourSpecification	0-23	
daySpecification	1-31	`last`: Indicates the last day of the target month.

Specification location	Specifiable value	Specifiable keyword
		`weekday`: Indicates the nearest working day (Monday to Friday) to the specified date. `lastweekday`: Indicates the last working day of the target month.
`monthSpecification`	1-12	
`dayOfWeekSpecification`	0 (Sunday) - 6 (Saturday)	`last`: If only this specification is used, this simply indicates Saturday. If `last` is used together with a numeric indicating a day of the week, indicates the last occurrence of that day of the week in the target month.
`secondsSpecification` (optional)	0-59	

1.5 Functions

1.5.1 Single-row Functions

A single-row function returns a single value for each event (row) that is the output of a complex event processing statement. These functions can be described at any position where expressions are permitted.

The table below shows the built-in single-row functions.

Function	Syntax	Explanation

CASE	`case` *`value`*
	`when` *`comparisonValue`* `then` *`result`*
	`[when` *`comparisonValue`* `then` *`result`*`] [...]`
	`[else` *`result`*`]`
	`end`
		Returns *`result`* at the first position where *`value`* is equivalent to *`comparisonValue`*.

	`case`
	`when` *`condition`* `then` *`result`*
	`[when` *`condition`* `then` *`result`*`] [...]`
	`[else` *`result`*`]`
	`end`
		Returns *`result`* at the first position where *`condition`* is TRUE.

CAST	`cast(`*`expression`*`,` *`dataType`*`)`	Converts the *`expression`* result to *`dataType`*. For *`dataType`*, `int`, `long`, `byte`, `short`, `char`, `double`, `float`, or `string` can be specified.
COALESCE	`coalesce(`*`expression`*`,` *`expression`* `[,`*`expression`*`] [, ...])`	Returns the value of the first *`expression`* in the list that is not null. If all are null, null is returned.
CURRENT_TIMESTAMP	`current_timestamp[()]`	Returns the current time using long milliseconds.

Function	Syntax	Explanation
MAX	max(*expression*, *expression* [,*expression*] [, ...])	Returns the maximum value out of all the *expression* values.
MIN	min(*expression*, *expression* [,*expression*] [, ...])	Returns the minimum value out of all the *expression* values.
PREV	prev(*expression*, *property*)	Returns the specified *property* value of the event at the *expression* value position, counting from the end in the data window, or returns all properties. If the data window has been sorted, it is the position based on that sequence. Specifying a stream name as the *property* returns all properties.
PREVTAIL	prevtail(*expression*, *property*)	Returns the specified *property* value of the event at the *expression* value position, counting from the start in the data window, or returns all properties. If the data window has been sorted, it is the position based on that sequence. Specifying a stream name as the *property* returns all properties.
PREVWINDOW	prevwindow(*property*)	Returns the specified *property* value for all events in the data window, or returns all properties. Specifying a stream name as the *property* returns all properties.
PREVCOUNT	prevcount(*property*)	Returns the number of events in the data window. For *property*, specify a property name or a stream name.
PRIOR	prior(*expression*, *property*)	Returns the specified *property* value of the event at the *expression* value position, counting from the end of arrived events, or returns all properties. The sequence is the event arrival sequence. Specifying a stream name as the *property* returns all properties.

1.5.2 Aggregate Functions

SQL standard functions

The table below shows the SQL standard aggregate functions that can be used by the complex event processing language.

Function	Syntax	Explanation
AVEDEV	avedev([all | distinct] *expression*)	Returns a double value showing the mean deviation of the *expression* value.
AVG	avg([all | distinct] *expression*)	Returns a double value showing the *expression* value average.
COUNT	count([all | distinct] *expression*)	Returns a long value showing the number of *expression* values that are not null.
	count(*)	Returns a long value showing the number of events.
MAX	max([all | distinct] *expression*)	Returns the maximum value of the *expression* value.
MEDIAN	median([all | distinct] *expression*)	Returns a double value showing the *expression* value median value. Non-numeric values (Not-a-Number: NaN) are ignored when calculating the median value.

Function	Syntax	Explanation
MIN	min([all | distinct] *expression*)	Returns the minimum value of the *expression* value.
STDDEV	stddev([all | distinct] *expression*)	Returns a double value showing the standard deviation of the *expression* value.
SUM	sum([all | distinct] *expression*)	Returns the sum of the *expression* values.

If distinct is specified, duplicated values are not included in calculations.

Data window aggregate functions

The table below shows the aggregate functions for data windows.

Function	Syntax	Explanation
FIRST	first (* | *eventStreamName*.* | *valueExpression* [, *indexExpression*])	Returns the property of the first event in the data window, in event arrival sequence, or the evaluation value at *valueExpression*. If multiple event streams are joined or if subqueries are included, use *eventStreamName* to specify the event stream for which you want properties returned. If *indexExpression* is specified, the property of the event at the expression value position, counting from the first event, is returned.
LAST	last(* | *eventStreamName*.* | *valueExpression* [,*indexExpression*])	Returns the property of the most recent event in the data window, in event arrival sequence, or the evaluation value at *valueExpression*. If multiple event streams are joined or if subqueries are included, use *eventStreamName* to specify the event stream for which you want properties returned. If *indexExpression* is specified, the property of the event at the expression value position, counting from the most recent event, is returned.
WINDOW	window(* | *eventStreamName*.* | *valueExpression*)	Returns the properties of all events in the data window, or the evaluation value at *valueExpression*. If multiple event streams are joined or if subqueries are included, use *eventStreamName* to specify the event stream for which you want properties returned.

The differences between the FIRST, LAST, and WINDOW aggregate functions and the PREVTAIL, PREV, and PREVWINDOW functions are that aggregate functions can operate using GROUP BY, and that the aggregate functions are based on the event arrival sequence rather than the sort sequence.

1.6 Operators

The table below shows the operators that can be used in complex event processing language expressions. The operator priority sequence is the same as Java standard.

Type	Operator	Explanation
Arithmetic	+, -	Monadic operators indicating positive and negative values. Dyadic operators for performing addition and subtraction.
	*, /	Dyadic operators for performing multiplication and division
	%	Dyadic operator for performing modulo (division remainder) operation
Logical	NOT	Negation of a logical value
	OR	Logical conjunction of two logical values

Type	Operator	Explanation
	AND	Logical disjunction of two logical values
Comparison	=, !=, <, >, <=, >=	Comparison of two values
Join	||	Joining of two character strings
Binary	&	AND operation for each bit
	|	OR operation for each bit
	^	Exclusive logical disjunction (XOR) operation for each bit

In addition, the keywords shown in the table below can be used in complex event processing language expressions.

Keyword	Syntax	Explanation
IN	*evaluationExpression* [not] in (*expression* [,*expression*] [, ...])	Returns TRUE if the value of the *evaluationExpression* is the same as any of the *expression* values within parentheses. If NOT is present, the negated value (if TRUE, FALSE is returned, and if FALSE, TRUE is returned) is returned.
	evaluationExpression [not] in ([| () *lowerLimitValue* : *upperLimitValue* () |])	Returns TRUE if the value of the *evaluationExpression* is within the *lowerLimitValue* and *upperLimitValue* range. If square brackets, "[" and "]", are used, the *lowerLimitValue* and the *upperLimitValue* are included in the range. If parentheses, "(" and ")", are used, the *lowerLimitValue* and the *upperLimitValue* are not included. If NOT is present, the negated value is returned.
BETWEEN	*evaluationExpression* [not] between *startExpression* and *endExpression*	Returns TRUE if the value of the *evaluationExpression* is within the *startExpression* and *endExpression* range. The *startExpression* and *endExpression* values are both included in the range. If NOT is present, the negated value is returned.
LIKE	*evaluationExpression* [not] like *patternRepresentation* [escape *character*]	Provides the SQL standard pattern matching function. Returns TRUE if the character string value of the *evaluationExpression* matches the pattern shown at *patternRepresentation*. If NOT is present, the negated value is returned. In the *patternRepresentation*, an underscore (_) indicates any single character, and the percent symbol (%) indicates any character string (includes 0 characters). The underscore (_) and the percent symbol (%) can be used as ordinary characters in a *patternRepresentation* by preceding them with the *character* specified as the ESCAPE.
REGEXP	*evaluationExpression* [not] regexp *patternRepresentation*	The same regular expressions as those implemented by the Java java.util.regex package are used in the *patternRepresentation* to perform *evaluationExpression* pattern matching.

Keyword	Syntax	Explanation

ANY, SOME	`expression operator any (expression [,expression] [, ...] )`
	`expression operator some (expression [,expression] [, ...] )`
		Uses `operator` to compare the left-side `expression` against all the `expression` values within parentheses on the right side, and returns TRUE if the result of any of the comparisons is TRUE.
		SOME and ANY are the same.

ALL	`expression operator all (expression [,expression] [, ...] )`	Uses `operator` to compare the left-side `expression` against all the `expression` values within parentheses on the right side, and returns TRUE if the result of all of the comparisons is TRUE.

1.7 Views

In complex event processing language, the events targeted for operations can be restricted (data window views) and values can be derived from event streams (derived value view) by specifying a view in relation to an event stream.

The table below shows a list of data window views.

View	Syntax	Explanation
Length window	`win:length(size)`	A length window that slides and holds the specified `size` amount of the most recent events.
Length batch window	`win:length_batch(size)`	A repetitive window that processes events when the specified `size` of events have accumulated, then releases them all.
Time window	`win:time(time)`	A time window that holds the specified `time` amount of events. Specify a time representation or the number of seconds.
Time batch window	`win:time_batch(time [, startTime])`	A repetitive window that stockpiles the specified `time` events and processes them when the time is reached, then releases them all. If a `startTime` is not specified, the `time` count starts from when the first event arrives. If a `startTime` is specified, the `time` count starts from that `startTime`. For the `startTime`, specify the number of milliseconds from January 1, 1970, 00:00:00 GMT.
Time-length combination batch window	`win:time_length_batch(time, size)`	A combination of the time window and length window. Events are processed and released when either the `time` or `size` condition is matched.
Keep-all window	`win:keepall()`	This window holds all events. Care must be taken with memory consumption because events are not released.
First length	`win:firstlength(size)`	Holds the first event to arrive that is the `size` amount.
First time	`win:firsttime(time)`	Holds the first event to arrive within the specified `time.`

View	Syntax	Explanation
Unique	`std:unique(`*`uniqueExpression`*`)`	Calculates the *`uniqueExpression`* for events and holds just the most recent events for each value.
Grouped data window	`std:groupwin(`*`groupingExpression`*`)`	Calculates the *`groupingExpression`* for events and holds a data window for each value.
Last event	`std:lastevent()`	Holds just the most recent event. This is equivalent to a length window that set 1 to size.
First event	`std:firstevent()`	Holds just the event that arrived first.
First unique	`std:firstunique(`*`uniqueExpression`*`)`	Calculates the *`uniqueExpression`* for events and holds just the event that arrived first for each value.
Sorted window	`ext:sort(`*`size`*`, `*`sortExpression`*` [asc|desc] [,`*`sortExpression`*` [asc |desc]] [, ...])`	Starting from the top of the results sorted in accordance with the *`sortExpression`*, holds the number of events specified at *`size`*. Ascending or descending order can be specified using ASC/DESC.

The table below shows the derived value view.

View	Syntax	Explanation
Size	`std:size([`*`property`*` [, ...]])`	This view enables the number of events received from the event stream to be referenced using the SIZE property having long values. In addition to SIZE, if *`property`* is specified, that property can also be referenced.

Chapter 2 Filter Rule Language Reference

This chapter describes how to describe filter rules for the High-speed Filter.

2.1 What are Filter Rules?

The High-speed Filter can extract data from events received by input adapters (extraction process), and can join them to master data (join processing). After input events are processed by the high-speed filter, they are passed to the complex event processing.

Filter rules are the rules for describing the extraction process and the join processing. Input events can be in CSV format or XML format, and rules (the ON statement described later in this document) are described for each event type that indicates the structure of event data.

Filter rules are described in the rule definitions deployed to the CEP engine, and the syntax is checked when the CEP engine starts.

Point

- Input events are passed to the complex event processing only if the setting for using complex event processing is set in the event type definition.
- If filter rules are omitted from the rule definitions, input events are passed directly to the complex event processing.

Note

CEP engine startup fails if the syntax check detects an error.

2.2 Basic Filter Rule Items

2.2.1 Spaces

For filter rule syntax, the following characters are treated as spaces and ignored:

- Space character (' ')
- Horizontal tab (HT)
- Line feed (LF)
- Carriage return (CR)

2.2.2 Keywords

The following keywords, written entirely in lower-case, are specified in filter rule syntax:

- on
- if
- then
- join
- output
- as

2.2.3 Comments

If two successive single-byte slashes (//) are used at the start of a row in filter rule syntax, the entire row is treated as being a comment row and is ignored.

```
// comment
```

2.2.4 Master ID

In filter rule syntax, master data can be referenced by specifying the development asset ID of a master definition deployed to the CEP engine.

In this manual, the development asset ID of a master definition is referred to as a master ID.

2.2.5 Item Names and Attribute Names

The single-byte characters (as indicated below) and multi-byte characters can be specified for the item names and attribute names used in filter rule syntax. The character encoding is UTF-8.

!	-	.	0	1	2	3	4	5	6	7	8	9	:
@	A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z	

Note

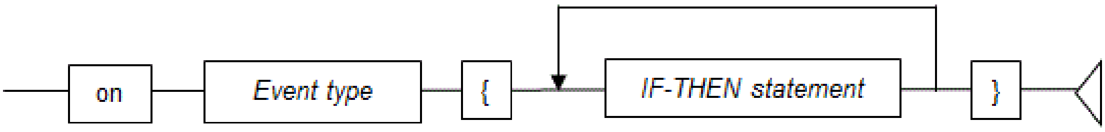
- Names starting with a hyphen (-) or period (.) cannot be specified.
- Single-byte spaces cannot be specified. (The far-right cell in the bottom row of the table does not indicate a space character.)

2.3 Filter Rule Syntax

This section explains the filter rule syntax (grammar).

2.3.1 ON Statement

Indicates a rule to be described for a particular event type.



Event type

Specify the development asset ID of the event type definition.

Point

- Only one ON statement can be described for one event type.

- ON statements for multiple event types can be described in one rule definition.
- Input events having an event type for which an ON statement is not described are passed directly to the Complex Event Processing.

Note

Multiple rule definitions can be deployed to one CEP engine, but the CEP engine will fail to start if these rule definitions contain more than one ON statement described for the same event type.

IF-THEN statement

Describe the input event processing for the specified type.

If multiple IF-THEN statements are described, the output of the previous IF-THEN statement will be the input for the next IF-THEN statement. The output of the IF-THEN statement described last is passed to the Complex Event Processing.

Point

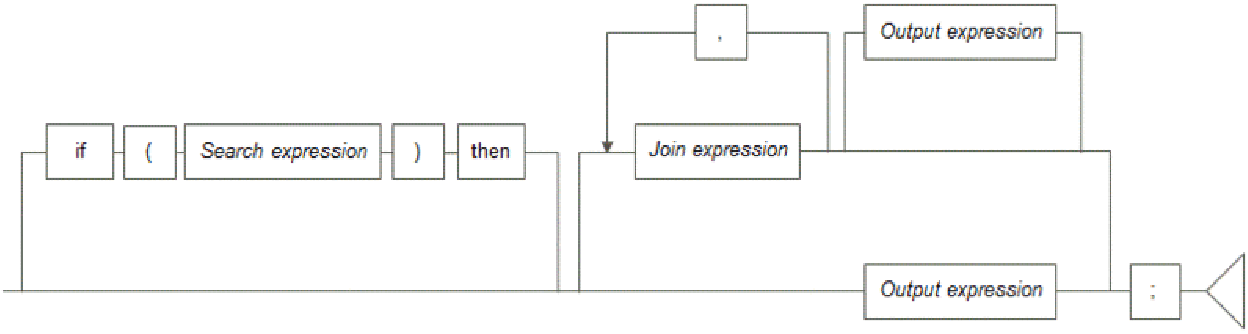
- If only an extraction process is described for the previous IF-THEN statement (if a join expression is not described and an output() without arguments is specified), the event type that is the input for the next IF-THEN statement does not change.
- If the previous IF-THEN statement is other than the above (there is a join expression or an output() with arguments is specified), the input event of the next IF-THEN statement is in CSV format.

Note

A maximum of 10 IF-THEN statements can be described in one ON statement. If more than 10 IF-THEN statements are described, CEP engine startup fails.

2.3.2 IF-THEN Statement

Describe the extraction process and the join processing for events passed from an input adapter or a previous IF-THEN statement (only if multiple IF-THEN statements are described).



If only an extraction process is to be used, do not describe a join expression and describe output() without arguments.

If only join processing is to be used, omit the part between IF and THEN and describe just the join processing and output processing.

Search expression

Describe the extraction process (conditions) that filters events.

See

Refer to "2.5 Search Expression Format" for details.

Join expression

Describe the join processing for joining to master data.

If multiple join expressions are described, events can be joined to more than one master data.

See

Refer to "2.6 Join Expression Format" for details.

Output expression

Specify the output item from within an input event or a joined master data.

If join processing is described, the output is in CSV format. When output is in CSV format, each item is tied together by double quotation marks (").

Point

- The output expression can be omitted if the input event is in CSV format and is not the last IF-THEN statement.
- If only an extraction process is implemented, the input event is output as is (the event type does not change).

See

Refer to "2.7 Output Expression Format" for details.

2.4 Common Formats

This section explains the common formats used in filter rule syntax.

- Item Expressions
- Path Expressions
- Text Expressions
- Attribute Expressions
- Data Types
- Literals
- Comparison Operators
- Logical Operators

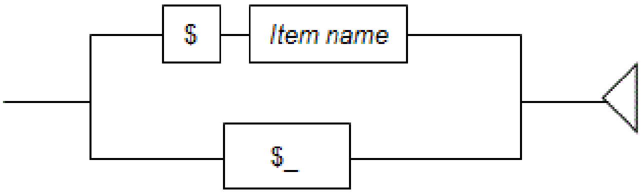
- Item References

2.4.1 Item Expressions

The structure of a CSV type event is expressed by the item name in column elements of the event type definition.

An item expression specifies the location of elements in a CSV type event using the item name.

The format used by item expressions is shown below.



$_

Is specified in a search expression when all items are search targets.

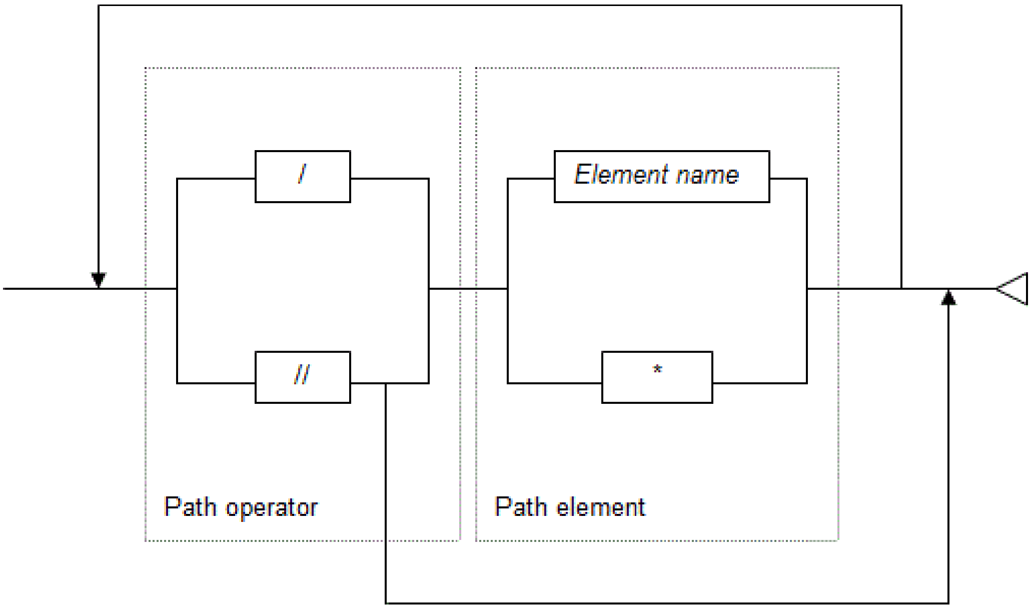
Note

Use "$_" only when a pattern is specified in a search expression. At any other time, using "$_" will return items with an underscore ("_") as the item name.

2.4.2 Path Expressions

An XML type input event's structure is represented as a tree. Path expressions are used to identify the position of nodes within an XML tree structure.

The format used by path expressions is shown below.



Path element

Path elements are used to identify element nodes in an XML data.

Path element	Explanation
Element name	Specifies the name of an element node
*	Signifies all element nodes below the upper node

Path operator

Path operators express the relationship between path elements.

Path operator	Explanation
/	Target is the node below the upper node
//	Target is all descendant nodes below the upper node

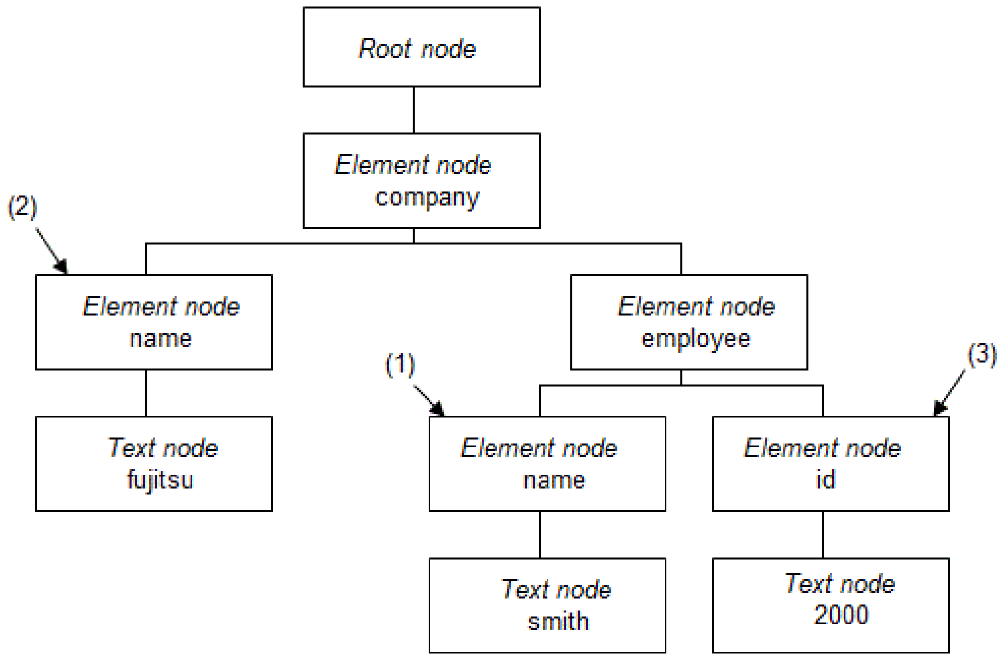
Note

- Do not specify "//" and "*" consecutively in a path expression.
- Do not specify a path expression as "/*".
- If a pattern is specified in a search expression, specify the "//" path operator at the end of the expression.

Example

A sample path expression is shown below.

```
<company>
    <name>fujitsu</name>
    <employee>
        <name>smith</name>
        <id>2000</id>
    </employee>
</company>
```



```
/company/employee/name
```

This path expression contains the "name" element node below the "employee" element node, which is further aligned below the "company" element node under the root node. This node is indicated by (1) in the above figure.

```
//name
```

This path expression indicates all "name" element nodes below the root node. These nodes are indicated by (1) and (2) in the above figure.

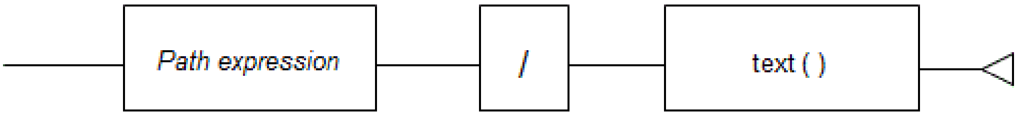
```
/company/*/id
```

This path expression refers to the "id" element node, which can be under any element node ("name" or "employee") below the "company" element node under the root node. This node is indicated by (3) in the above figure.

2.4.3 Text Expressions

Text expressions specify the (string) value of a text node below element nodes in an XML type input event specified using path expressions.

The format used by text expressions is shown below.



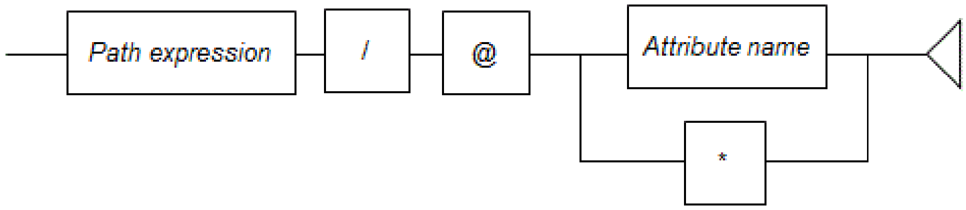
Note

- The '*' path element cannot be specified in the path expression.
- The "//" path operator cannot be specified in the path expression.

2.4.4 Attribute Expressions

Attribute expressions specify the value of attribute nodes of element nodes in an XML type input event specified using path expressions.

The format used by attribute expressions is shown below.



Attribute name

Specify the names of attribute nodes of element nodes in the path expression. The at sign (@) must be specified before the attribute name. Using the asterisk (`*`) selects all attribute nodes of element nodes specified in the path expression.

Point

If the '`//`' path operator is specified at the end of a path expression, the '`/`' after the path expression is omitted.

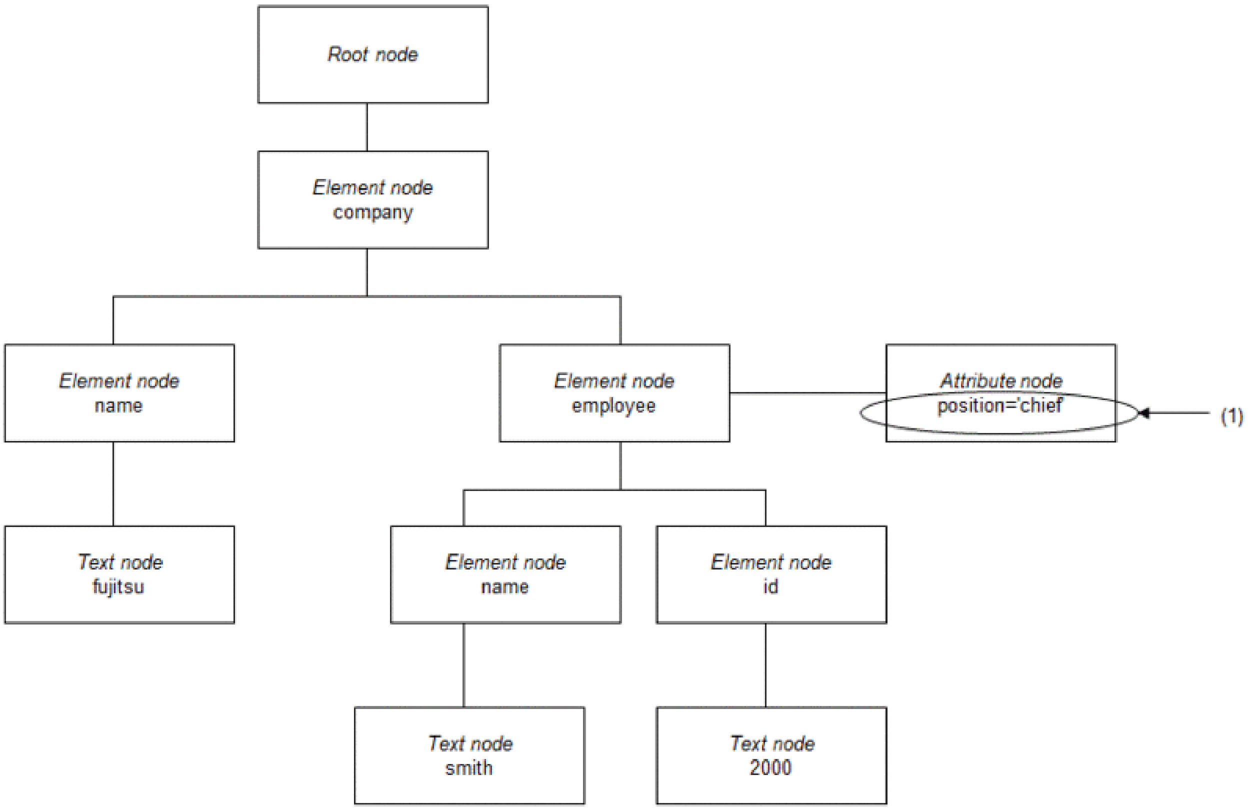
Note

- Attribute expressions with an asterisk (`*`) cannot be used as the argument for a function.
- You can specify an asterisk (`*`) in an attribute expression only when a pattern is specified in a search expression.
- Element names starting with the at sign (@) cannot be specified as the path element in an attribute expression.

Example

A sample attribute expression is shown below.

```
<company>
    <name>fujitsu</name>
    <employee position="chief">
        <name>smith</name>
        <id>2000</id>
    </employee>
</company>
```



```
/company/employee/@position
```

This expression shows the value of the attribute node "`position`" of the "`employee`" element node. The value is "`chief`" indicated by (1) in the above figure.

2.4.5 Data Types

This table lists the data types supported by the filter rule.

Data type	Explanation
String	- Represents a series of alphabetic characters, such as "ABC" - Input data specified in an item name or text expression is considered to be of the string data type
Numeric	- Represents integers as well as decimal values; for example, 10 and -0.5 - Up to 18 digits and 18 decimal places can be specified for integers and decimal values, respectively (*1)

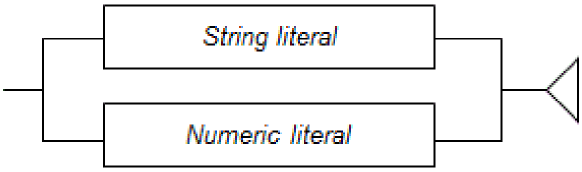
*1: There is no restriction on the number of digits that can be specified for a numeric type value in a condition expression.

2.4.6 Literals

Literals represent values that are directly used in a search expression without any computation involved.

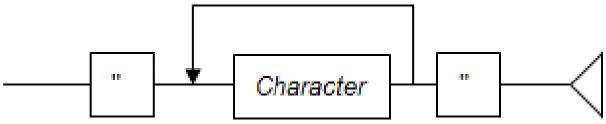
Literals are used in various condition expressions.

The format used by literals is shown below.



2.4.6.1 String Literal

The format used by string literals is shown below.



Character

Specify single-byte characters or multi-byte characters (UTF-8 encoding).

Any spaces within quotation marks are treated as valid values, and the characters appear exactly as they have been specified.

If the backslash ("\") symbol precedes a character in a string literal, the character is replaced by an alternative character in accordance with the following table.

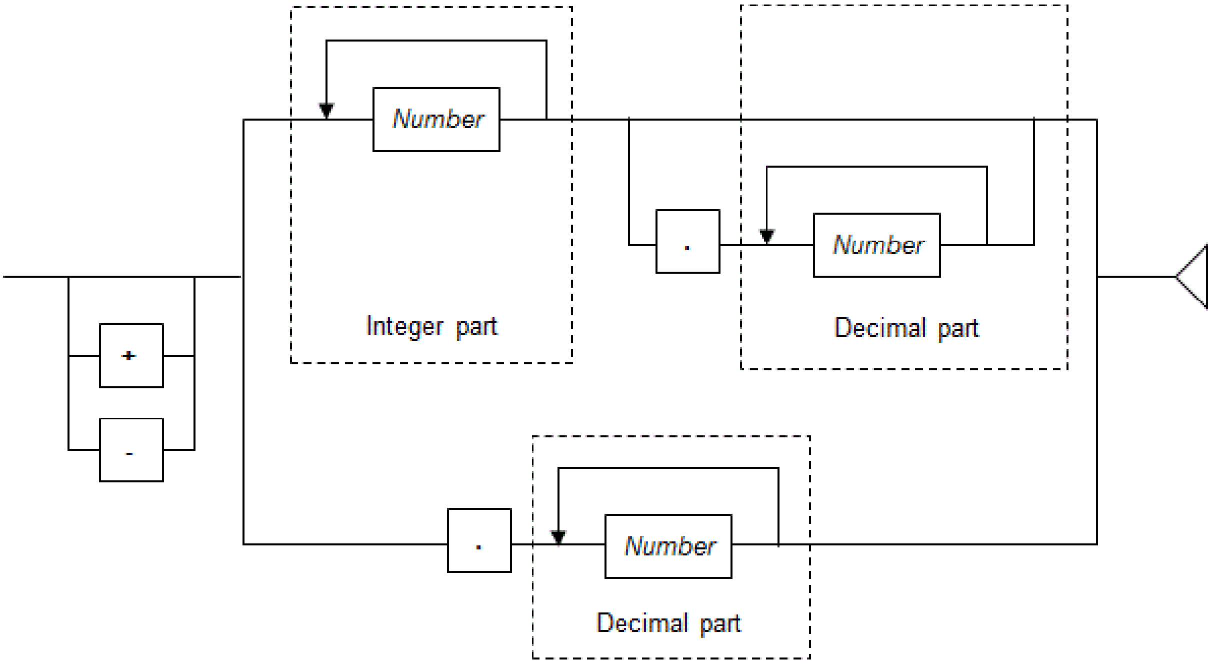
Character within a literal	Replacement character
`\s`	Single-byte space
`\S`	Double-byte space
`\n`	Line feed
`\t`	Horizontal tab
`\"`	Quotation marks
`\\`	\ (backslash)

Point

If the character immediately following the backslash ("\") is not listed in the above table, the replacement character is the character itself. For example, "`\a`" will be replaced by "`a`".

2.4.6.2 Numeric Literal

The format used by numeric literals is shown below.



Number

Specify a digit from 0 through 9.

Note

- When specifying a numeric literal in a condition expression:
  - There is no limit to the number of digits that can be specified.
  - Spaces cannot be specified in a numeric literal. The only exception is spaces contained in a prefix or suffix.
- In all other cases:
  - The integer part and decimal places of a numeric literal can be up to a maximum of 18 digits. However, this excludes any 0 values at the beginning of the integer part.
  - Spaces cannot be specified in a numeric literal. The only exception is spaces contained in a prefix or suffix.

2.4.7 Comparison Operators

Comparison operators that can be used in condition expressions are listed in the following table.

Comparison operator				Search type	Explanation
=				Partial match (*1)	TRUE if the search keyword is included in the element value
!=					TRUE if the search keyword is not included in the element value
==				Complete match	TRUE if the search keyword and the element value exactly match
!==					TRUE even if a portion of the search keyword differs from the element value
<	<=	>	>=	Size comparison	Compares the size of the search keyword and the element value

*1: If the keyword is a numeric value, it will exactly match.

2.4.8 Logical Operators

Logical operators define the relationship between two adjacent condition expressions, when multiple expressions are specified.

Logical operators that can be used in condition expressions are listed in the following table.

Logical operator	Logical operation	Explanation
`AND`	AND operation	Links pairs of condition expressions with the AND operator Evaluates to TRUE if the results of both condition expressions is TRUE Evaluates to FALSE if either or both condition expressions are FALSE
`OR`	OR operation	Links pairs of condition expressions with the OR operator Evaluates to TRUE if either or both condition expressions are TRUE Evaluates to FALSE if both condition expressions are FALSE

Point

- The AND operator is evaluated first in condition expressions that contains both AND and OR operators.
- Use parentheses "`()`" to change the order of logical operators' evaluation. In the following example, `(`*`conditionExpression2`* OR *`conditionExpression3`*`)` is evaluated first.

```
conditionExpression1 AND (conditionExpression2 OR conditionExpression3)
```

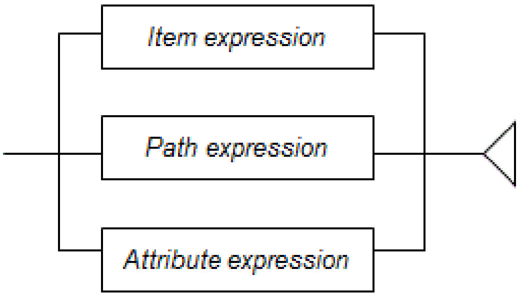
Note

Lower-case "`and`" and "`or`" cannot be used as logical operators.

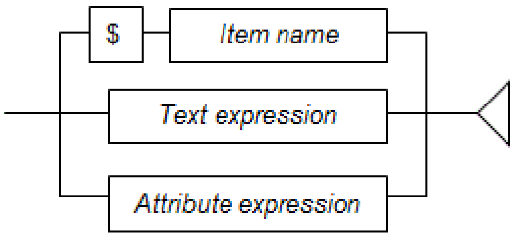
2.4.9 Item References

Item references return the values of input data and variables.

The format of item references used by condition expressions is shown below.



The format of item references used by join-relational expressions, output items of output expressions, and lookup functions is shown below.



Note

If an item reference that does not exist is specified

If an item reference that does not exist is specified in a search expression, join-relational expression, or similar, it is processed as having a "null" value.

See

- Refer to "2.4.1 Item Expressions" for details.
- Refer to "2.4.2 Path Expressions" for details.
- Refer to "2.4.3 Text Expressions" for details.
- Refer to "2.4.4 Attribute Expressions" for details.

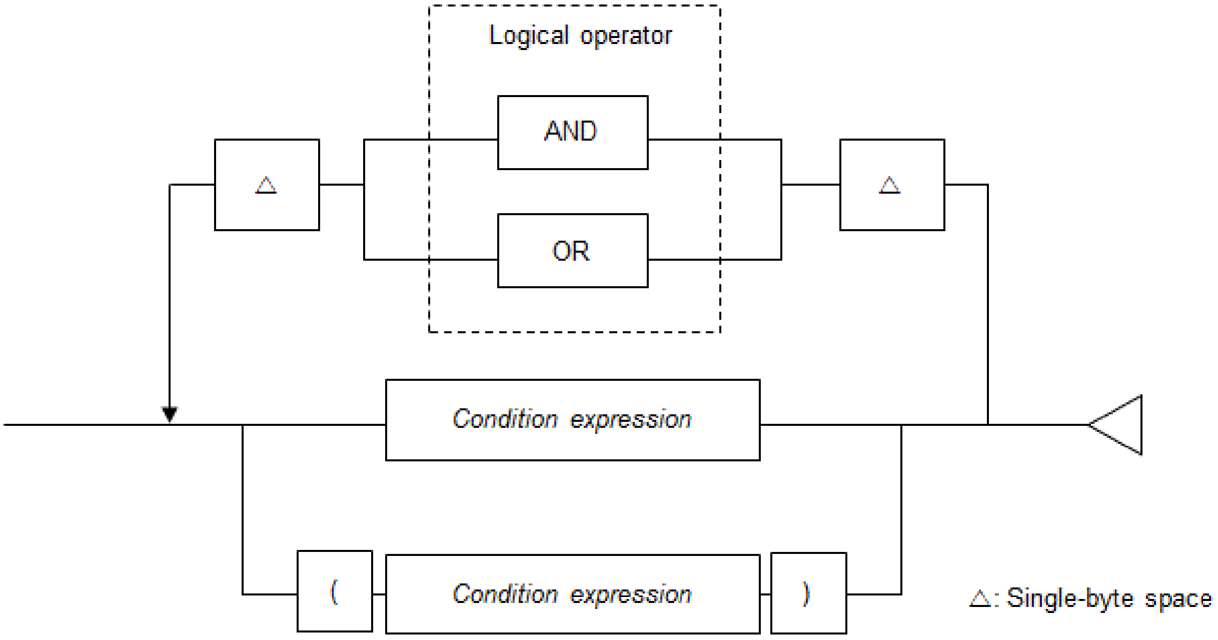
2.5 Search Expression Format

Search expressions are used to specify conditions that apply to input event to be retrieved.

A search expression consists of one or more condition expressions.

Use logical operators to specify multiple condition expressions.

The format used by search expressions is shown below.



See

- Refer to "2.4.8 Logical Operators" for details.

Note

If the search expression size (character string length in UTF-8 encoding) exceeds 65535 bytes, CEP engine startup fails.

2.5.1 Condition Expressions

Condition expressions are used when performing a comparison between items specified on left and right sides of a comparison operator.

There are the following three types of condition expressions:

Keyword search

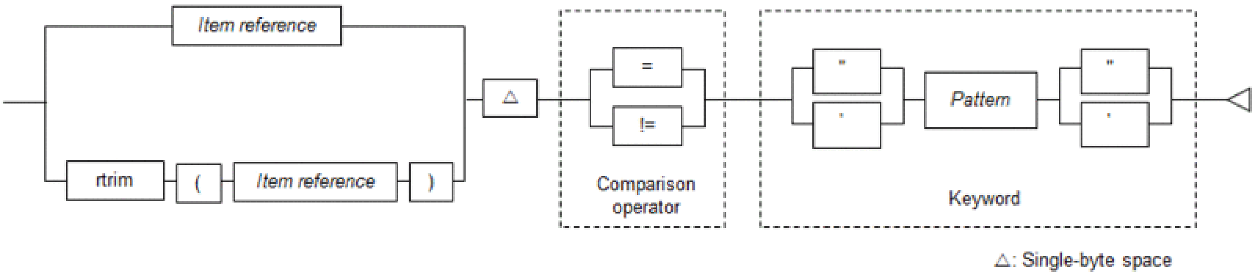
Compares the keyword with the input event.

In the search expression, specify an item reference (left side) and a keyword (right side).

Pattern search, string search, and numeric search can be used.

Example

Example of keyword search format (pattern search)



See

- Refer to "2.5.5 Keyword Search" for information on keyword search formats.
- Refer to "2.4.9 Item References" for details.

Comparison between items

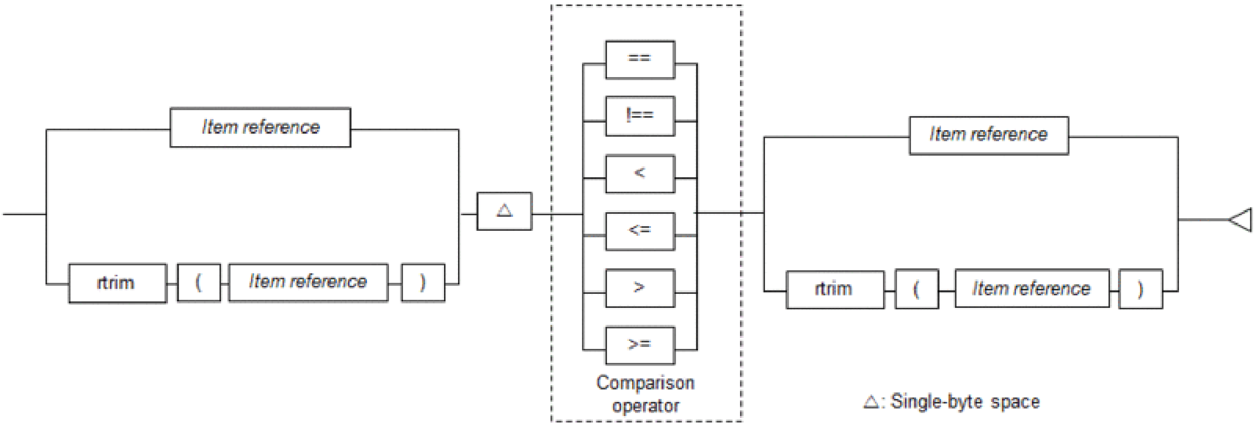
Compare items in input events.

In the search expression, specify an item reference (left side) and the value of another item reference (right side).

String comparisons and numeric comparisons can be used.

Example

Example of comparison between items format (string comparison)



See

- Refer to "2.5.6 Comparison between Items" for information on comparison between items formats.
- Refer to "2.4.9 Item References" for details.

Lookup search

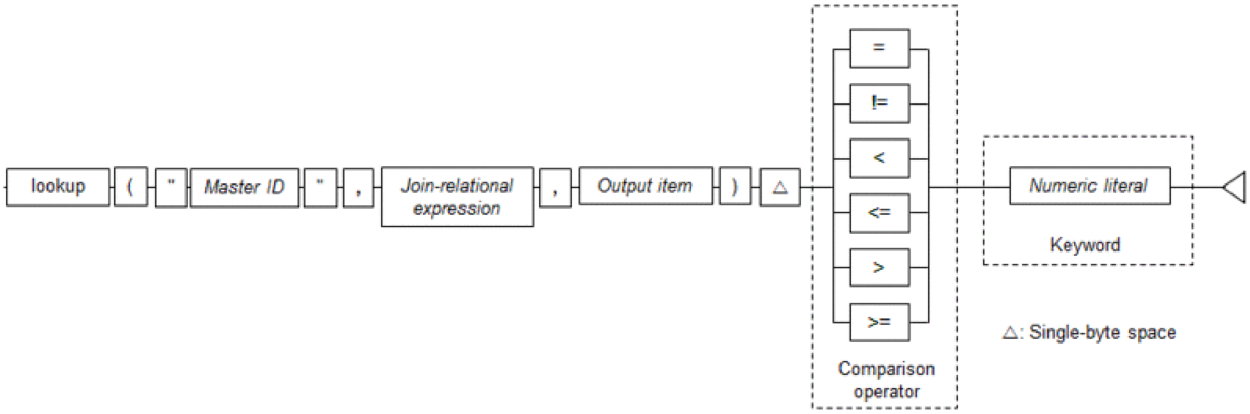
Compare master data items with a keyword.

In the search expression, specify the master data content (left side) and the keyword (right side).

Pattern search, string search, numeric search, master data search, lookup sum matching, and lookup count matching can be used.

Example

Example of lookup search format (numeric search)



See

Refer to "2.5.7 Lookup Search" for information on lookup search formats.

Point

Ensure both the left and right sides of a condition expression are of the same data type.

Note

- The "//" path operator can be specified at the end of path expressions only when a pattern is specified for the keyword. Specifying the "//" path operator at the end of a path expression selects all the element nodes under the element node specified by the path expression.
- The "*" path element can be specified at the end of path expressions only when a pattern is specified for the keyword.
- The "$_" path element can be specified in item expressions only when a pattern is specified for the keyword.
- The "*" path element can be specified in the attribute name of attribute expressions only when a pattern is specified for the keyword.

2.5.2 Escape Characters

To specify following characters in pattern and strings, precede them with the escape character '\'.

Escape character is '\'.

The following table lists characters that require the escape character "\".

Table 2.1 Characters that must be preceded by the escape character

<table>
<tr><th>Character</th><th>Specified as:</th></tr>
<tr><td>"</td><td>\"</td></tr>
<tr><td>$</td><td>\$</td></tr>
<tr><td>&</td><td>\&</td></tr>
<tr><td>'</td><td>\'</td></tr>
<tr><td>(</td><td>\(</td></tr>
<tr><td>)</td><td>\)</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>+</td><td>\+</td></tr>
<tr><td>,</td><td>\,</td></tr>
<tr><td>-</td><td>\-</td></tr>
<tr><td>.</td><td>\.</td></tr>
<tr><td>?</td><td>\?</td></tr>
<tr><td>[</td><td>\[</td></tr>
<tr><td>\</td><td>\\</td></tr>
<tr><td>]</td><td>\]</td></tr>
<tr><td>^</td><td>\^</td></tr>
<tr><td>{</td><td>\{</td></tr>
<tr><td>|</td><td>\|</td></tr>
<tr><td>}</td><td>\}</td></tr>
<tr><td>~</td><td>\~</td></tr>
</table>

Example

If the search target string is to be 'abc\', specify 'abc\\'.

2.5.3 Entity References

When searching an entity reference string within an XML type input event, specify that entity reference string as the search keyword.

To search for a symbol represented by an entity reference, specify the symbol as the search keyword.

Table 2.2 Example of entity references

Entity reference	Symbol represented
&lt;	<
&gt;	>
&amp;	&
&apos;	'
&quot;	"

Note

- When specifying entity references for keywords, the ampersand (&) must be preceded by escape character.
- To specify symbols represented for keywords, the ampersand (&), single quotation marks ('), and quotation marks (") must be preceded by the escape character.

2.5.4 Special Characters

When specifying special characters in a pattern or string, use the values listed in the table below.

Table 2.3 Special characters

Character	Specified as:
Single-byte space	\s
Double-byte space	\S
Line feed	\n
Horizontal tab	\t

2.5.5 Keyword Search

This section explains the condition expressions that compare input event items with keywords.

- Pattern Search
- String Search
- Numeric Search

2.5.5.1 Pattern Search

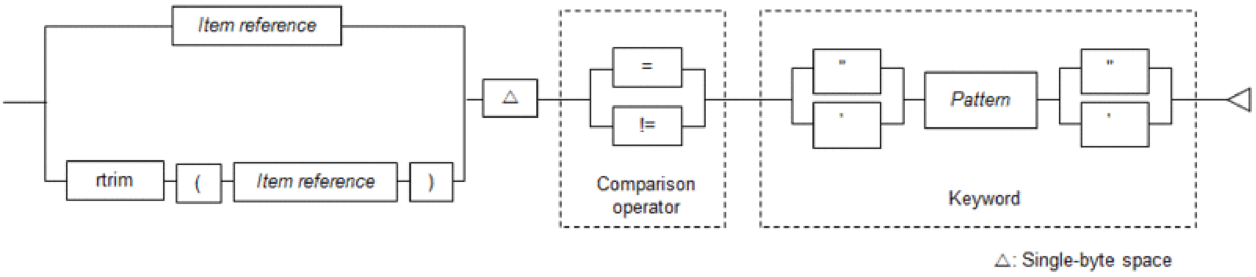
Various conditions can be specified for patterns. Complex conditions, such as searches for partial matches and word searches, can be described for searches.

The following types of pattern search are available:

Classification	Type
Pattern search (string)	String match specification
	Prefix match specification
	Suffix match specification
	Free Character Specification
	Character Interval Specification
	Partial Character Specification
	Character Range Specification
	Numeric Range Specification
Pattern search (word)	Word match specification
	Word interval specification
Logical conjunction, logical disjunction, and negation in pattern searches	Logical conjunction
	Logical disjunction
	Negation

Pattern Search format

The format used by the pattern search is shown below.



Point

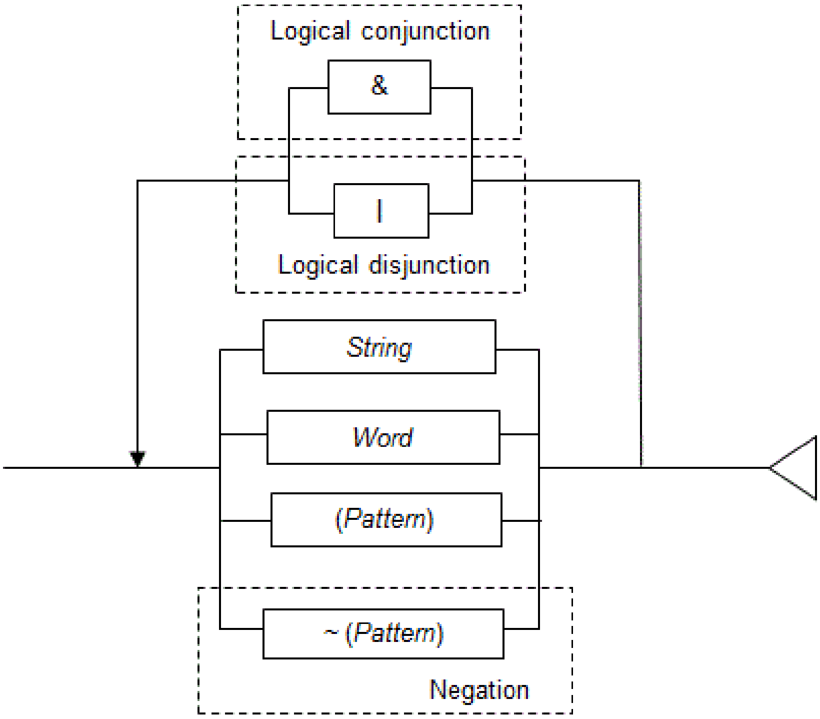
- A pattern search is enclosed within quotation marks (") or single quotation marks (').
- The handling of upper-case and lower-case single-byte alphabetics in search target strings can be specified by the rule definition (ANKmix option). The handling of upper-case and lower-case double-byte alphabetics can be specified by the rule definition (KNJmix option). Refer to "2.9 Options" for information on the ANKmix and KNJmix options.

Note

Quotation marks (") and single quotation marks (') cannot be used together.

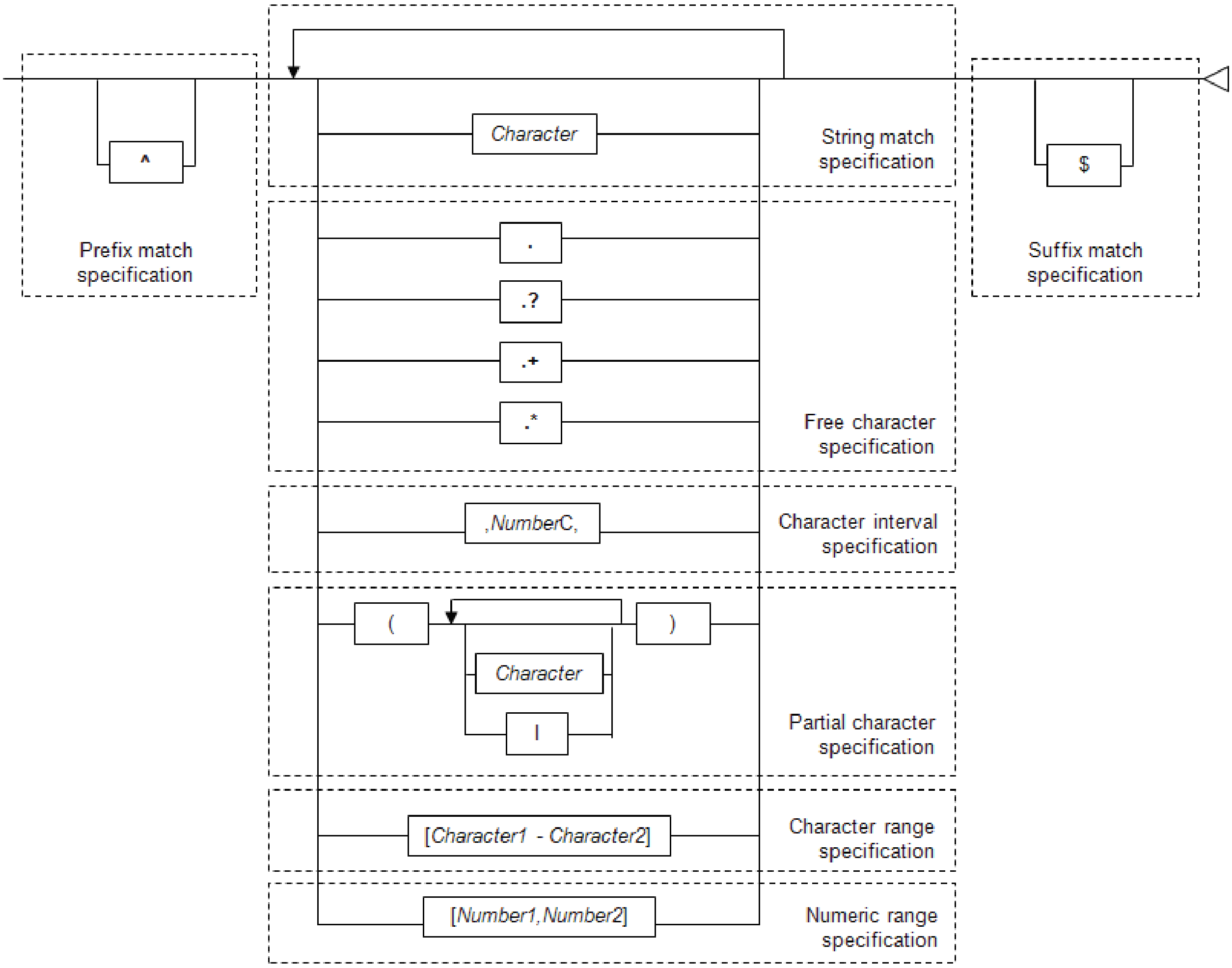
Pattern format

Pattern format is shown below.



2.5.5.1.1 Pattern search (string)

The format of a pattern search (string) is shown below.


Character
String match specification
^
Prefix match specification
$
Suffix match specification
.
.?
.+
.*
Free character specification
,NumberC,
Character interval specification
(
)
Character
|
Partial character specification
[Character1 - Character2]
Character range specification
[Number1,Number2]
Numeric range specification

Point

Characters to be excluded as search targets can be specified in rule definitions (SkipChar option). Refer to "2.9 Options" for information on the SkipChar option.

String match specification

Finds out whether the value of an element node includes the specified keyword.

Example

Search for data that includes the string "`Fujitsu`" in the element value indicated by `/root/text`.

```
/root/text = 'Fujitsu'
```

Prefix match specification

Finds out whether the specified keywords exist at the start of an element node's value.

Example

Search for data that begins with the string "`Fujitsu`" in the element value indicated by `/root/text`.

```
/root/text = '^Fujitsu'
```

Suffix match specification

Finds out whether the specified keywords exist at the end of an element node's value.

Example

Search for data that ends with the string "Fujitsu" in the element value indicated by /root/text.

```
/root/text = 'Fujitsu$'
```

Free character specification

Find out whether the value of an element node and the value of a text node include a keyword that contains free characters.

Free characters included in keywords can be specified in four ways, as shown in the following table.

Symbol	Explanation	Can be used consecutively
.	Any one arbitrary character	Yes
.?	Zero or one arbitrary character	Yes
.+	One or more arbitrary characters	No
.*	Zero or more arbitrary characters	No

Note

If symbols that cannot be used consecutively are used consecutively, CEP engine startup fails.

Example

Search for data that includes the strings "Fujitsu" and "company" in the element value indicated by /root/text, provided the number of characters between these strings is 0 or more.

```
/root/text = 'Fujitsu.*company'
```

Information

Free character specifications can be combined. The following table shows examples of how combinations of free character specifications evaluate to TRUE or FALSE for different data. These results assume that "=" has been specified as the comparison operator.

Keyword example	Data example			
	AB	AXB	AYYB	AZZZB
'A.'	Y	Y	Y	Y
'A.B'	x	Y	x	x
'A.?B'	Y	Y	x	x
'A.+B'	x	Y	Y	Y

Keyword example	Data example			
	AB	AXB	AYYB	AZZZB
'A.*B'	Y	Y	Y	Y
'A..?B'	x	Y	Y	x
'A..+B'	x	x	Y	Y
'A..*B'	x	Y	Y	Y
'A.?.+B'	x	Y	Y	Y
'A.?.*B'	Y	Y	Y	Y

Y: TRUE

x: FALSE

Character interval specification

Finds out whether the two specified keywords appear in succession in an element node's value within an interval of the specified number of characters. The numeric value of character interval specifications must be from 0 through 1024.

Note

- Character interval specifications can only be specified once in string searches.
- Free character specifications cannot be specified immediately before or after character interval specifications.

Example

Search for data that includes the strings "alcohol" and "concentration" in the element value indicated by /root/text, provided the number of characters between these strings is 10 or less.

```
/root/text = 'alcohol,10C,concentration'
```

Partial character specification

Finds out whether the value of an element node and the value of a text node contain the specified keyword.

Part of the keyword consists of one of multiple strings.

Note

Depending on the number of characters specified, a large amount of memory can be used. Insufficient memory can cause the search response to deteriorate. Note that, if a memory overflow is detected, an error message is output and the input event is discarded (processing of the next input event continues).

Refer to Section 6.3.4, "Tuning" in the *User's Guide* for information on the memory estimation method.

Example

Search for data that includes any of the strings-"Jon Smith", "John Smith", or "Jonathon Smith"-in the element value indicated by /root/text.

```
/root/text = 'Jo(n|hn|nathon) Smith'
```

Character range specification

Finds out whether the value of an element node includes the specified keyword where part of the keyword consists of any character in a specific range.

The character code value of the start character must be smaller than the character code value of the end character. Both the start character (character 1) and the end character (character 2) must be single ASCII characters and must not be control characters.

Note

Depending on the specified character range, a large amount of memory can be used. Insufficient memory can cause the search response to deteriorate. Note that, if a memory overflow is detected, an error message is output and the input event is discarded (processing of the next input event continues).

Refer to Section 6.3.4, "Tuning" in the *User's Guide* for information on the memory estimation method.

Example

Search for data that includes the strings "`classA`", "`classB`", and "`classC`" in the element value indicated by `/root/text`.

```
/root/text = 'class[A-C]'
```

Numeric range specification

Finds out whether the value of an element node includes the specified keyword where part of the keyword consists of any numeric value in a specific range.

The start numeric value (numeric value 1) and the end numeric value (numeric value 2) must be specified using single-byte numbers. These values must be from 0 through 999. Also, the start numeric value must be smaller than the end numeric value.

Point

Correct search results can be obtained if characters are specified before and after the numeric value.

Note

Depending on the specified numeric range, a large amount of memory can be used. Insufficient memory can cause the search response to deteriorate. Note that, if a memory overflow is detected, an error message is output and the input event is discarded (processing of the next input event continues).

Refer to Section 6.3.4, "Tuning" in the *User's Guide* for information on the memory estimation method.

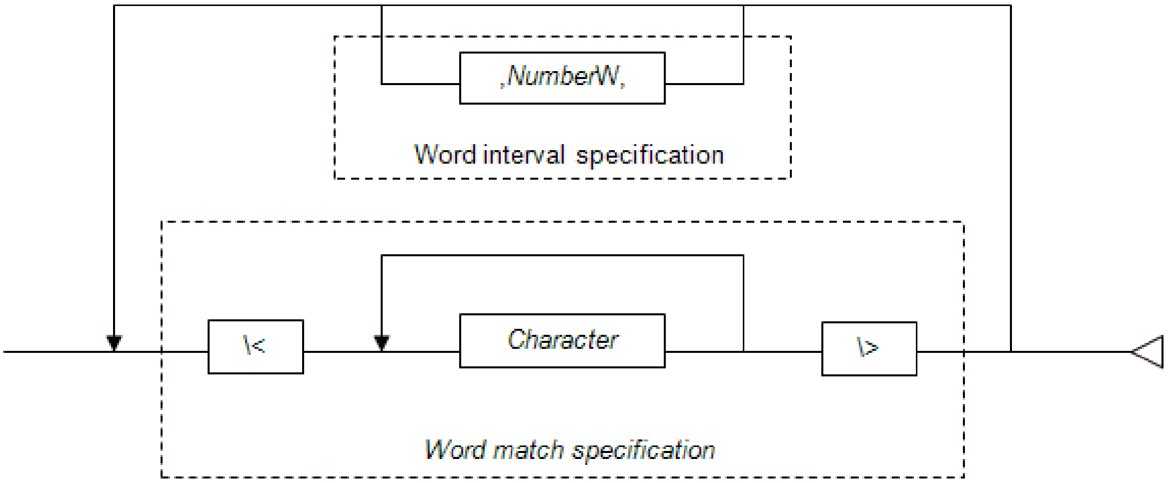
Example

Search for data that includes the strings "`alcohol 9%`", "`alcohol 10%`", and "`alcohol 11%`" in the element values indicated by `/root/text`.

```
/root/text = 'alcohol [9,11]%'
```

2.5.5.1.2 Pattern search (word)

The format of pattern search (word) is shown below.



Point

- The word delimiter character can be specified in the rule definitions (SeparateChar option). Refer to "2.9 Options" for information on the SeparateChar option.
- ASCII characters (except for the word delimiter character) can be described in word searches.

Word match specification

Finds out whether the value of an element node and the value of the text node contain any individual words that match the specified keyword. For word searches, strings separated by the delimiter are considered as individual words.

Example

Search for data containing the word "`the`" in the element value indicated by `/root/text`.

```
/root/text = '\<the\>'
```

The string "the" in "mother" will evaluate to FALSE because it occurs within a larger word.

Word interval specification

Finds out whether the two keywords appear in succession in an element node's value within an interval of the specified number of words.

Numeric values specified for word interval specifications must be from 0 through 1024.

Example

Search for data that includes the words "`search`" and "`AsIs`" in the element value indicated by `/root/text`, provided the number of words between these two words is 10 or less.

```
/root/text = '\<search\>,10W,\<AsIs\>'
```

Note

Word interval specifications can be used only once in word searches.

2.5.5.1.3 Logical conjunction, logical disjunction, and negation in pattern searches

This section explains pattern searches (logical conjunction, logical disjunction, and negation).

Logical conjunction

Finds out whether the value of element nodes specified in a path expression includes all the specified patterns.

Example

Evaluates to TRUE if the value of the element node represented by '`/root/text`' includes the strings "`fast`" and "`search`".

```
/root/text = 'fast&search'
```

Logical disjunction

Finds out whether the value of an element node specified in a path expression includes any of the specified patterns.

Example

Evaluates to TRUE if the value of the element node represented by '`/root/text`' includes either the string "`fast`" or the string "`search`".

```
/root/text = 'fast|search'
```

Negation

Finds out whether the value of an element node specified in a path expression includes none of the specified patterns.

Example

Evaluates to TRUE if the value of the element node represented by '`/root/text`' includes neither the string "`fast`" nor the string "`search`".

```
/root/text = '~(fast|search)'
```

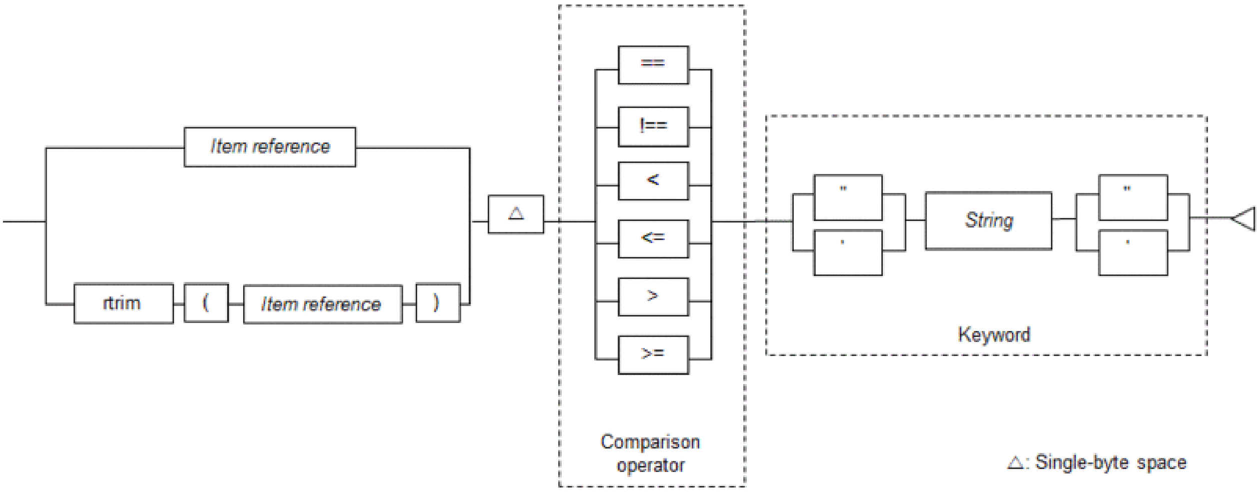
Point

- For pattern searches, you can use logical conjunction, logical disjunction, and negation in combination. When this happens, the order of evaluation is Negation > Logical conjunction > Logical disjunction.
- Parentheses "`(`" and "`)`" may also be used to specify the order of evaluation. Conditions in parentheses are evaluated preferentially.

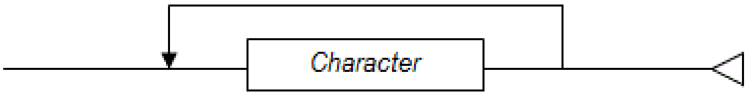
2.5.5.2 String Search

In a string search, a search is performed for events in which the element value exactly matches the value specified in the string or for events in which the element value is in the size relationship. As strings can be used for size comparisons, string searches can be used to search for mixed values containing both numerals and characters.

The format used by the string search is shown below.



String format is shown below.



A string search involves complete match and size comparison.

Complete match

Finds out if the value of an element node is equal to the string.

Example

Search for data equivalent to the string "`North Sydney, Australia`" indicated by the element value in `/root/area`.

```
/root/area == 'North Sydney, Australia'
```

Size comparison

This compares the size of the element value with the string in the encoding value, in sequence from the left of the string to the right.

Note

- It is not possible to specify the "`//`" path operator at the end of a path expression when the string exactly matches or when performing a size comparison.

- It is not possible to specify the "*" path element at the end of a path expression when the string exactly matches or when performing a size comparison.
- It is not possible to specify "$_" in the item expression when the string exactly matches or when performing a size comparison.
- When performing a string comparison, any element value to be searched within an XML event must have the same number of digits as the string specified in the keyword.

Point

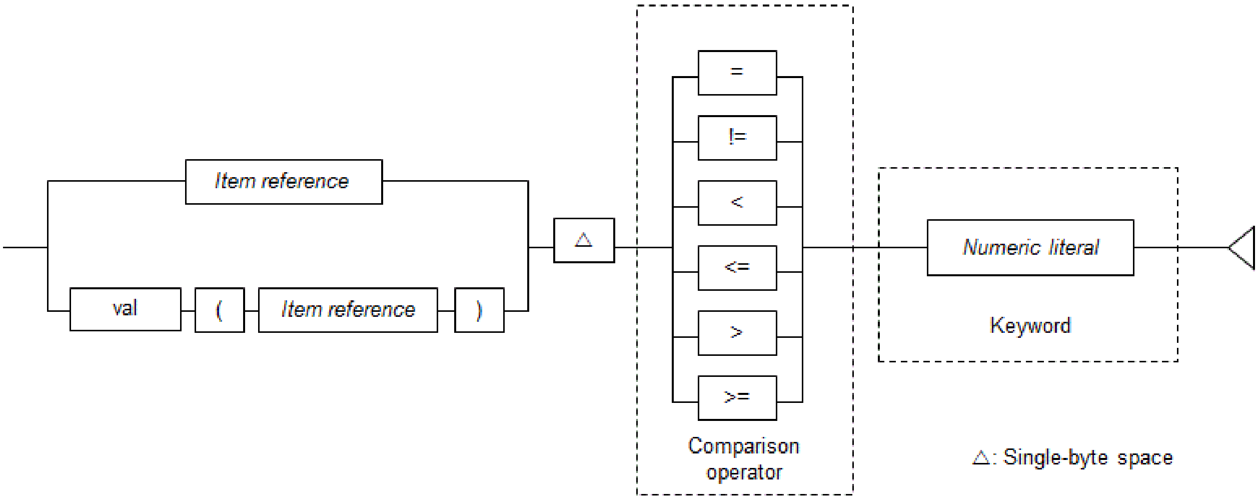
- Characters to be excluded as search targets can be specified in rule definitions (SkipChar option).
- The handling of upper-case and lower-case single-byte alphabets in search target strings can be specified by the rule definition (ANKmix option). The handling of upper-case and lower-case double-byte alphabets can be specified by the rule definition (KNJmix option).
- Refer to "2.9 Options" for details.

2.5.5.3 Numeric Search

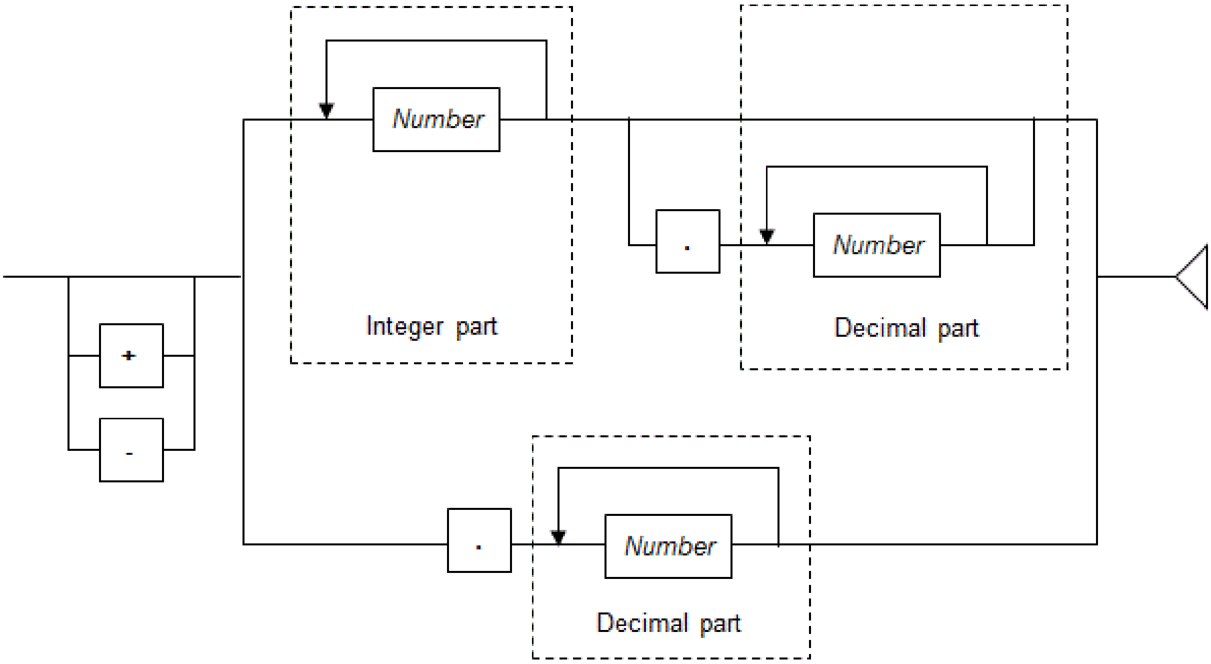
In a numeric search, a search is performed by extracting the numeric part from an element value and searching for events in which the extracted value matches a specified numeric value or for events in which the extracted value is in the size relationship. As the numeric portion of the element value is extracted automatically, this search can be used to extract numeric values that have been written in a variety of ways.

In addition, it is also possible to specify a numeric function on the left side of the comparison operator to perform comparisons with numeric values.

The format used by the numeric search is shown below.



Numeric literal format is shown below.



Number

For numbers, specify a digit from 0 through 9. There is no limit to the number of digits that may be specified.

Spaces may not be specified in a numeric literal, with the exception of spaces in a prefix or suffix.

The first string in the above format found from the element value will be treated as a numeric value.

Any commas (`,`) appearing in the integer part are ignored. If a decimal point is specified, the decimal places include all characters up to the first instance of a non-numeric character.

Example

This example evaluates to TRUE if the numeric component extracted from the value of the element node represented by '`/doc/money`' matches `1000`.

```
/doc/money = 1000
```

In the following examples, the value of the element node specified in the path expression contains multiple numeric values. In such cases, only the first numeric value is extracted.

Event A

```
<money>ABC123,456@789</money>
```

123456 is extracted.

Event B

```
<money>123456 7890123</money>
```

123456 is extracted.

Event C

```
<money>1,500yen</money>
```

1500 is extracted.

If the search data does not contain a valid numeric value string, the conditions evaluate to FALSE.

The following search target string does not contain a valid numeric value string.

```
<money></money>
```

Point

- The number of digits in a numeric value specified as the keyword need not match the value of the element node specified in a path expression.
- There is no need to make the number of digits in the integer or decimal part of element node values consistent across multiple XML events.

  EventA

  ```
  <money>1000.1</money>
  ```

  EventB

  ```
  <money>2000.05</money>
  ```

  EventC

  ```
  <money>10.5</money>
  ```

Note

- The '//' path operator cannot be specified at the end of a path expression when performing numeric search.
- The '*' path element cannot be specified at the end of a path expression when performing numeric search.
- In numeric search, "$_" cannot be specified as an item expression.

Example

Search for data greater than 1000 in the element value indicated by /root/money.

```
/root/money > 1000
```

See

- Refer to "2.8.3.1 val() Function" for details.
- Refer to "2.4.6.2 Numeric Literal" for details.

2.5.6 Comparison between Items

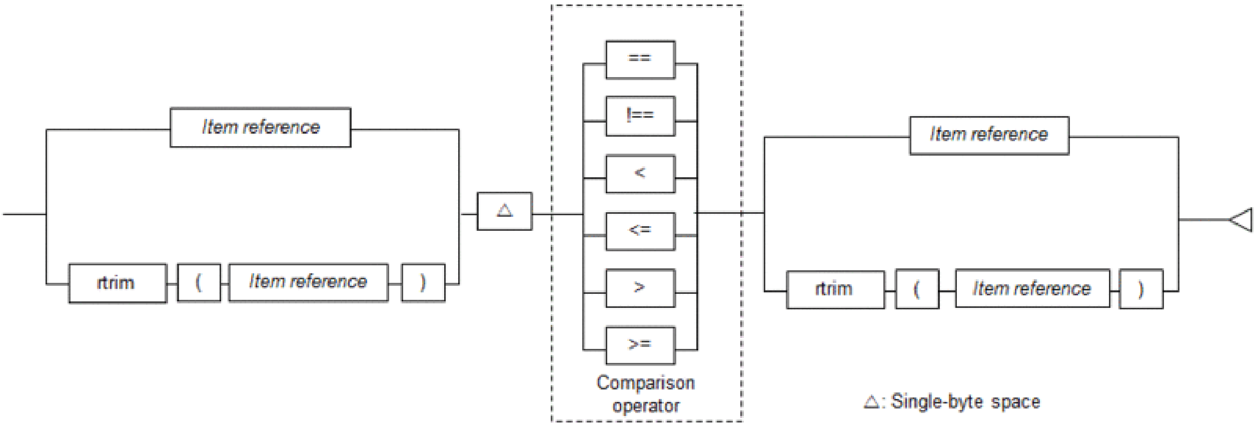
This section explains condition expressions that compare input event items with other input event items. There are two types: string comparison and numeric comparison.

- String Comparisons
- Numeric Comparisons
- Notes Common to String Comparisons and Numeric Comparisons

2.5.6.1 String Comparisons

Compares strings.

The format used for string comparisons is shown below.



Example

Search for data of employees who are not in the Management team and where the applicant and approver is the same.

```
$Position != "Management" AND $Applicant == $Approver
```

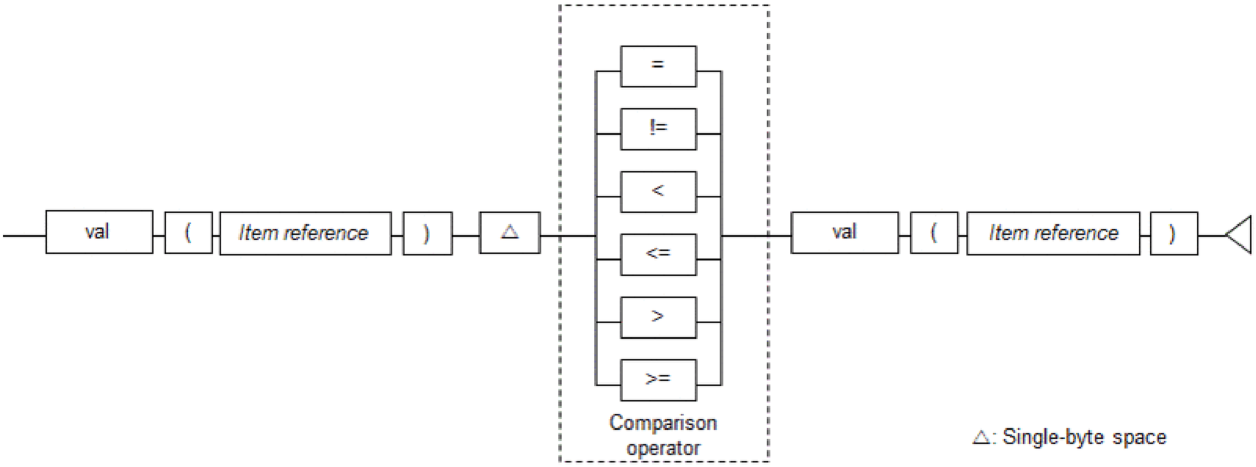
Note

It is not possible to specify a partial match in the comparison operator for string comparisons.

2.5.6.2 Numeric Comparisons

Compares numeric values with other numeric values.

The format used for numeric comparisons is shown below.



Example

Search for data of employees, with a tendency towards obesity, who have gained weight since last year and who have a waist measurement of more than 80 cm.

```
val($WeightLastYear) < val($Weight) AND val($WaistMeasurement) > 80.0
```

2.5.6.3 Notes Common to String Comparisons and Numeric Comparisons

This section explains considerations that apply to both string comparisons and numeric comparisons.

Note

- For comparisons between items, the SkipChar, ANKmix, and KNJmix options of the rule definitions deployed to the CEP engine are disabled. Refer to "2.9 Options" for details.
- If an input event in XML format includes an element node that repeatedly appears with the same name and that element is specified in a path expression in an item reference, the result of the comparison between items is TRUE if even one item fulfills the conditions. However, if a negation search expression (`!==`) is specified, the result is TRUE if no items fulfill the conditions.
- If a path expression, item expression or attribute expression having an element that is "null" is specified, the result is not TRUE even if the item element on the right side of the condition expression is "null".
- It is not possible to specify characters, such as "`//`" or "`*`", when specifying a path expression for comparison between items.
- It is not possible to specify "`$`" when specifying an item expression for comparison between items.
- It is not possible to specify "`*`" when specifying an attribute expression for comparison between items.

2.5.7 Lookup Search

This section explains condition expressions that compare master data items and keywords.

The lookup functions below are provided for returning master data item contents.

Refer to "2.8.4 Lookup Functions" for details.

lookup()

There are two types of functions: those that specify two arguments and those that specify three arguments.

If two arguments are specified, master item existence (true/false) is returned.

If three arguments are specified, master item contents are returned.

lookup_sum()

Returns the sum of the master item contents.

lookup_count()

Returns the number of master items existing (count).

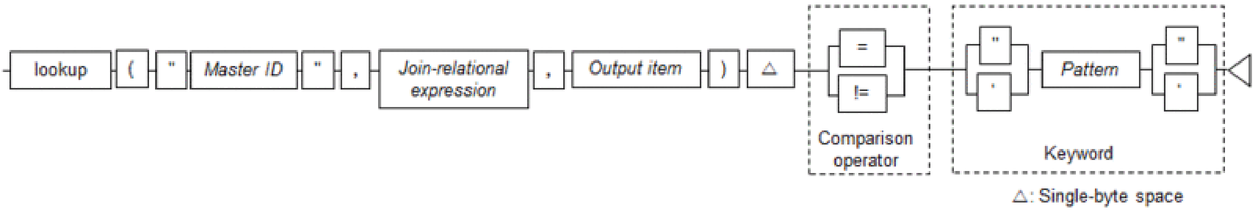
The sections below explain condition expressions in which these lookup functions are specified.

- Pattern Search
- String Search
- Numeric Search
- Master Data Search
- Lookup Sum Matching
- Lookup Count Matching

2.5.7.1 Pattern Search

Complex conditions in relation to master items joined by lookup(), such as searches for partial matches and word searches, can be described for searches.

The format used for pattern search is shown below.



Refer to "2.8.4.1 lookup() Function" for information on the lookup() function specified on the left side.

The keyword specified on the right side (a pattern search type) is similar to keyword search. Refer to "2.5.5.1 Pattern Search" for details.

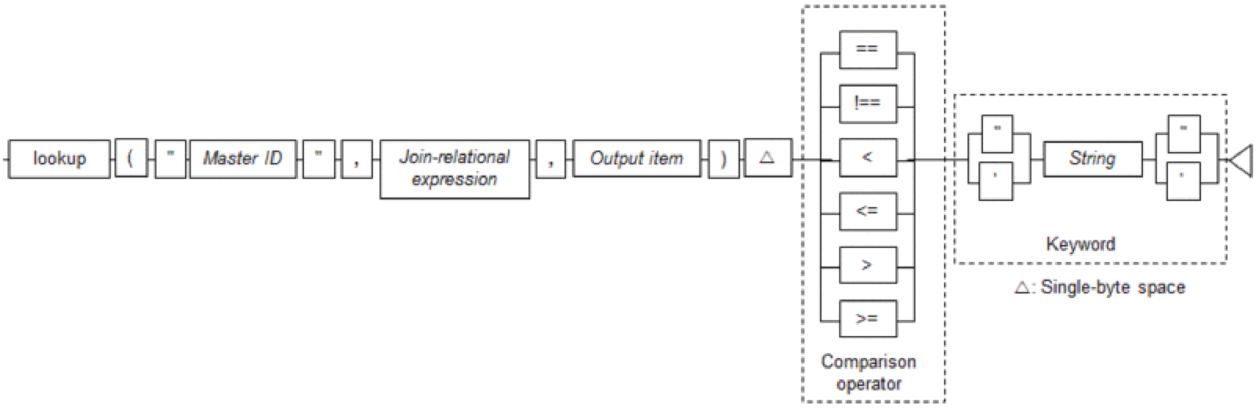
Point

- The handling of upper-case and lower-case single-byte alphabets in search target strings can be specified by the rule definition (ANKmix option). The handling of upper-case and lower-case double-byte alphabets can be specified by the rule definition (KNJmix option).
- If a pattern (string) is used for a searching, characters to be excluded as search targets can be specified in rule definitions (SkipChar option). Refer to "2.5.5.1.1 Pattern search (string)" for details.
- If a pattern (word) is used for a search, the word delimiter character can be specified in the rule definitions (SeparateChar option). Refer to "2.5.5.1.2 Pattern search (word)" for details.
- Refer to "2.9 Options" for details.

2.5.7.2 String Search

String search searches master items joined by lookup() for events which are a complete match with the value specified in the string or for events in a size relationship.

The format used for string search is shown below.



Refer to "2.8.4.1 lookup() Function" for information on the lookup() function specified on the left side.

The keyword specified on the right side (string) is similar to keyword search. Refer to "2.5.5.2 String Search" for details.

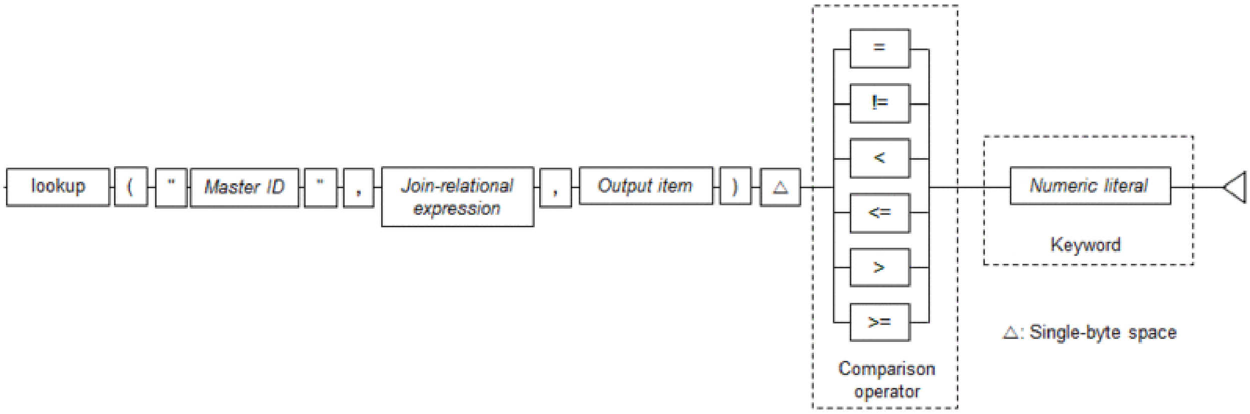
Point

- Characters to be excluded as search targets can be specified in rule definitions (SkipChar option).
- The handling of upper-case and lower-case single-byte alphabets in search target strings can be specified by the rule definition (ANKmix option). The handling of upper-case and lower-case double-byte alphabets can be specified by the rule definition (KNJmix option).
- Refer to "2.9 Options" for details.

2.5.7.3 Numeric Search

Numeric search searches master items joined by lookup() for events which match the specified numeric value or for events in a size relationship.

The format used for numeric search is shown below.



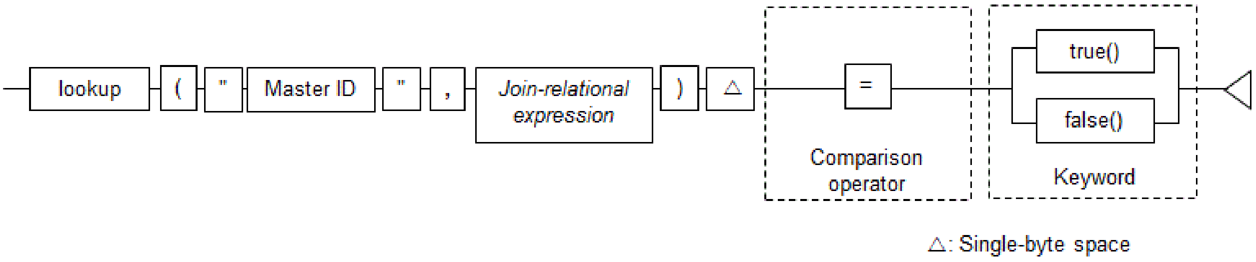
Refer to "2.8.4.1 lookup() Function" for information on the lookup() function specified on the left side.

The keyword specified on the right side (numeric) is similar to keyword search. Refer to "2.5.5.3 Numeric Search" for details.

2.5.7.4 Master Data Search

Evaluates the existence of master data joined by lookup().

The format used for master data search is shown below.



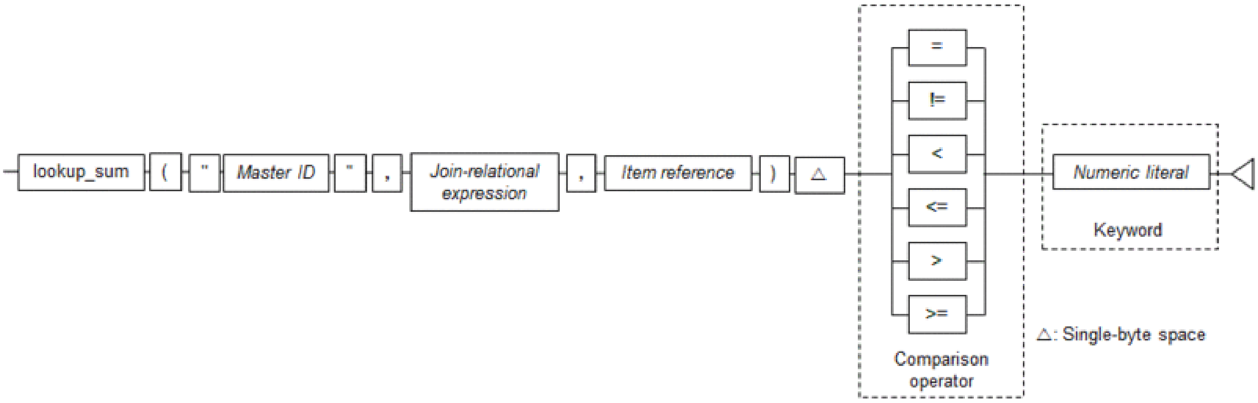
Refer to "2.8.4.1 lookup() Function" for information on the lookup() function specified on the left side.

A true() function or a false() function is specified in the keyword specified on the right side. Refer to "2.8.5 Boolean Functions" for information on the format of these functions.

2.5.7.5 Lookup Sum Matching

Searches the sum of master items joined by lookup_sum() for events which match the specified numeric or for events in a size relationship.

The format used for lookup sum matching is shown below.



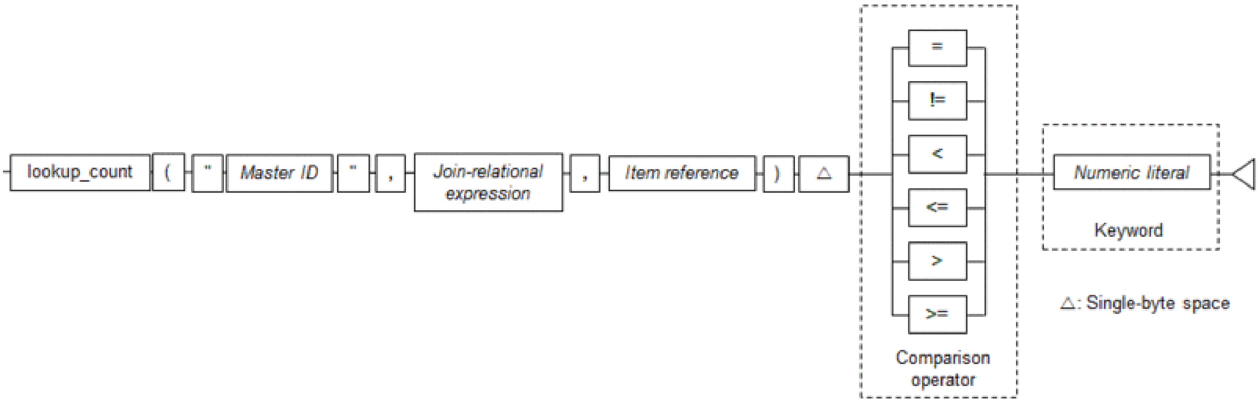
Refer to "2.8.4.2 lookup_sum() Function" for information on the lookup_sum() function specified on the left side.

The keyword specified on the right side (numeric) is similar to keyword search. Refer to "2.5.5.3 Numeric Search" for details.

2.5.7.6 Lookup Count Matching

Searches the count for master items joined by lookup_count() for events which match the specified numeric or for events in a size relationship.

The format used for lookup count matching is shown below.



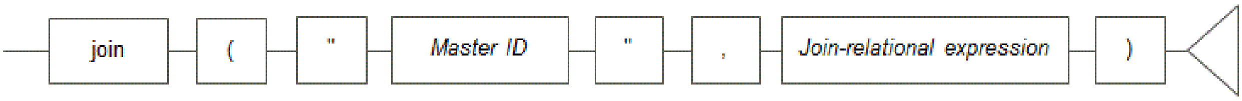
Refer to "2.8.4.3 lookup_count() Function" for information on the lookup_count() function specified on the left side.

The keyword specified on the right side (numeric) is similar to keyword search. Refer to "2.5.5.3 Numeric Search" for details.

2.6 Join Expression Format

A join expression is used to join an input event and master data.

The format used for join expression is shown below.



Master ID

Specify the development asset ID of the master definition.

Join-relational expression

Describe the join-relational expression used when joining the event and the master.

Point

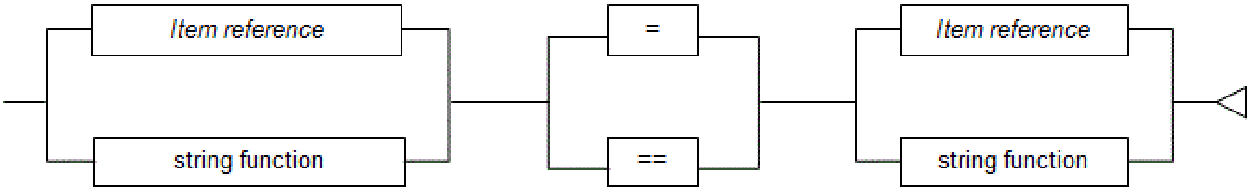
Only one master data can be handled by one join expression. Therefore, add join expressions if you want to join an event to multiple master data.

2.6.1 Join-Relational Expression

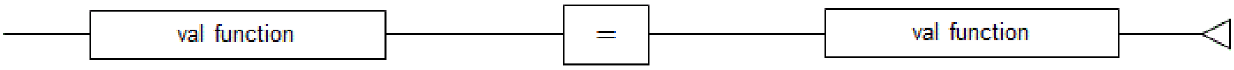
Specify the conditions for joining the input event (left side) and the master file (right side).

The format used for join-relational expression is shown below.

String type



Numeric type

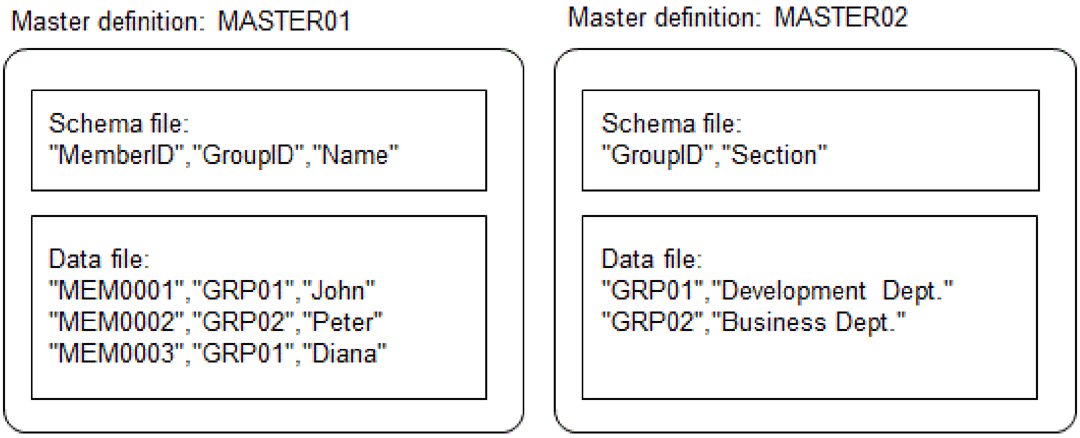


Point

Specify the same type on the left and right sides of the join-relational expression.

Note

- If the item reference and the function result specified on the left side of the join-relational expression are "null", the join-relational expression is not TRUE even if the item reference and the function result specified on the right side of the join-relational expression are "null".
- The join target is only the item within the input event (the input event item joins the master data as a key). Therefore, master data joined by one IF-THEN statement and another master data cannot be joined.
  If you want to join a master data item as a key to another master data, describe a join expression in the next IF-THEN statement as shown below, or prepare master data that is already joined and join that.



Master definition: MASTER01

Schema file:
"MemberID","GroupID","Name"

Data file:
"MEM0001","GRP01","John"
"MEM0002","GRP02","Peter"
"MEM0003","GRP01","Diana"

Master definition: MASTER02

Schema file:
"GroupID","Section"

Data file:
"GRP01","Development Dept."
"GRP02","Business Dept."

The joining of two master data can be described using two IF-THEN statements, as follows:

```
join("MASTER01", $ID == $MemberID) output();
join("MASTER02", $MASTER01.GroupID == $GroupID) output();
```

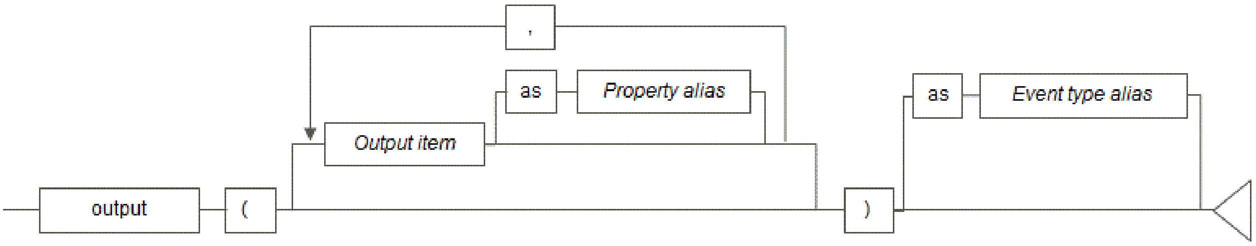
See

Refer to "2.4.7 Comparison Operators" for information on the meaning of the comparison operators.

2.7 Output Expression Format

An output expression is used to output the result of joining an input event and master data.

The format used for output expression is shown below.



Output item

Specify the property of the event to be output and the master data item.

If no master data fulfills the join-relational expression conditions, the output as the master item will be empty.

Operation is as follows if an output item is not specified:

With no join expression:

Input events filtered by a search expression are output as is.

If there is no search expression, CEP engine startup fails.

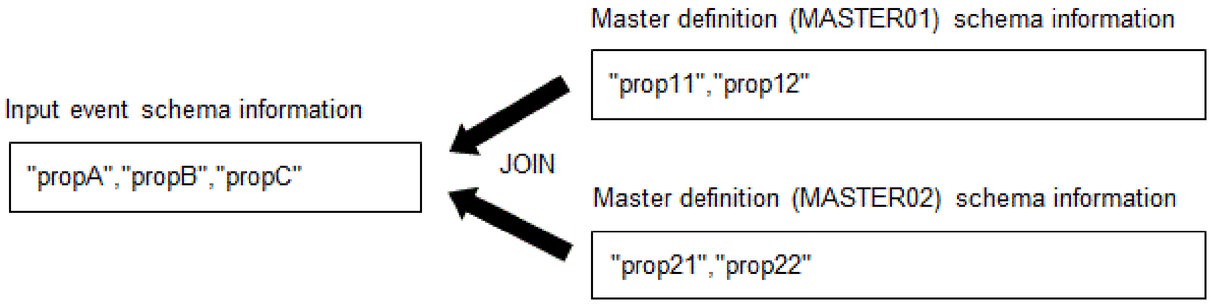
With join expression:

All properties of input events and all items of all joined master data are output in sequence (if input events are in CSV format).

If input events are in XML format, CEP engine startup fails.

Point

If an output expression that does not have an argument (output item) after the join expression is described, the output results are the same as for an output expression in which all items of input events and master data are specified.


Master definition (MASTER01) schema information
"prop11","prop12"
Input event schema information
"propA","propB","propC"
JOIN
Master definition (MASTER02) schema information
"prop21","prop22"

IF-THEN statement:

```
join("MASTER01", $propA == $prop11), join("MASTER02", $propB == $prop22) output();
```

The above output() has the same output result as the following output expression:

```
output($propA,$propB,$propC,"MASTER01".$prop11,"MASTER01".$prop12,"MASTER02".$prop21,"MASTER02".
$prop22);
```

Note

- Output item specification is mandatory if join processing is used for events in XML format.
- Output item specification can be omitted only if a search expression or a join expression is specified.

- There is no limit to the number of output items that can be specified for one output(). However, if the size obtained from the following calculation expression exceeds 65535 bytes, CEP engine startup fails:

  Size calculation expression:

  ```
  sizeRequiredForOutputItems + numberOfOutputItems - 1 (bytes)
  ```

Output item	Required size
Item reference	Item reference string length
Master item reference	Master item reference string length + string length of join-relational expression of the target join expression + 9
lookup_sum() function	lookup_sum argument string length + string length of join-relational expression of the target join expression + 13
lookup_count() function	lookup_count argument string length + string length of join-relational expression of the target join expression + 15

  Each of the string lengths specifies the string length in UTF-8 encoding.

  Refer to "2.7.1 Output Items" for details.

  Sizes (examples) in output() in the following IF-THEN statement:

  ```
  join("MASTER01", $message = $word)
  output($ID, "MASTER01".$word, lookup_sum("MASTER01".$weight));
  ```

  The calculation result is 93 bytes.

  ```
  stringLengthOf"$ID"
   + (stringLengthOf""MASTER01".$word" + stringLengthOf"$message = $word" + 9)
   + (stringLengthOf""MASTER01".$weight" + stringLengthOf"$message = $word" + 13)
   + numberOfOutputItems - 1
  = 3 + (16 + 16 + 9) + (18 + 16 + 13) + 3 - 1 = 93
  ```

Property alias

Attaches an alias to items that are output.

For output expressions other than the last IF-THEN statement, a property alias must be specified.

The property alias can be used as property names of input events for next processing.

A property alias must be specified if the input event is in XML format (except for the last output expression within an ON statement).

If the input event is in CSV format, the following property aliases are generated automatically:

Property	Automatically generated property alias
Input event item	The input event item name is specified as is.
Master data item	The name is generated by using a period (.) to join the master ID and the item name.

Note

- A property alias cannot be specified in the last output expression within an ON statement.
- Restrictions apply to the characters that can be used in a property alias. Refer to "2.2.5 Item Names and Attribute Names" for details.

- The same property alias cannot be used more than once in an output expression.
- If the same item is output multiple times, each must have a different property alias attached.
- If the input format is XML, a property alias must be specified unless it is the last output expression within an ON statement. If not specified, CEP engine startup fails.
- If the same alias is defined more than once in an output expression, automatically generated aliases included, CEP engine startup fails.

Example

If `"Mst1".$item1` is specified as the output item, the property alias is as follows:

```
Mst1.item1
```

Event type alias

Attaches an event type alias to the event to be output.

An event type alias must be specified in the output expression of the last IF-THEN statement.

Specify the development asset ID of the event type definition to be passed to the complex event processing.

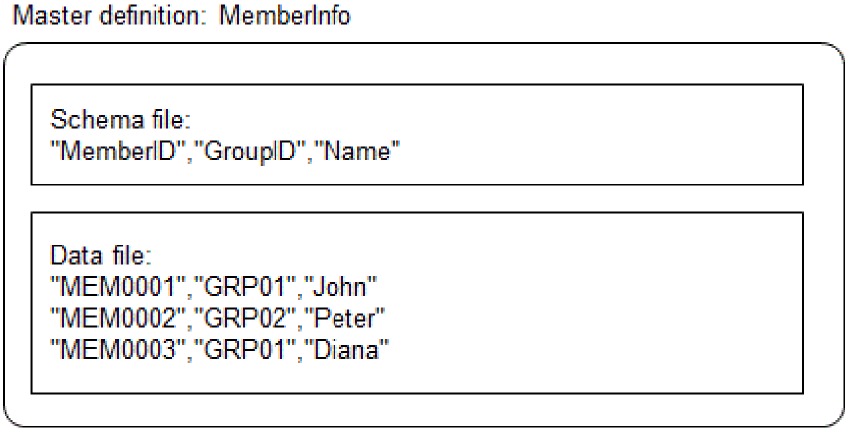
Note

- If an event type alias is specified other than in the last process in an ON statement, CEP engine startup fails.
- If the extraction process is the only processing in an ON statement, the event type alias must be the same as the event type specified in the ON statement.
- The event type alias in the last output expression of an ON statement must be registered as an event type definition.
- If the output is in CSV format, the number of properties in that event type definition must match the number of items to be output. If they do not match, CEP engine startup fails.

Point

- The output expression can be omitted if the input event is in CSV format and the IF-THEN statement is not the last IF-THEN statement.

- If multiple master items in a join expression fulfill the join-relational expression conditions, multiple items are output separated by commas (`,`) and all items are enclosed between double quotation marks (`"`), as shown below. If a double quotation mark is included in the master item contents, another double quotation mark is attached before that double quotation mark.


Master definition: MemberInfo

Schema file:
"MemberID","GroupID","Name"

Data file:
"MEM0001","GRP01","John"
"MEM0002","GRP02","Peter"
"MEM0003","GRP01","Diana"

IF-THEN statement:

```
join("MemberInfo", $group == $GroupID)
output("MemberInfo".$MemberID, "MemberInfo".$Name);
```

If `$group` joins the "`GRP01`" input event to the master data, the output results are as follows:

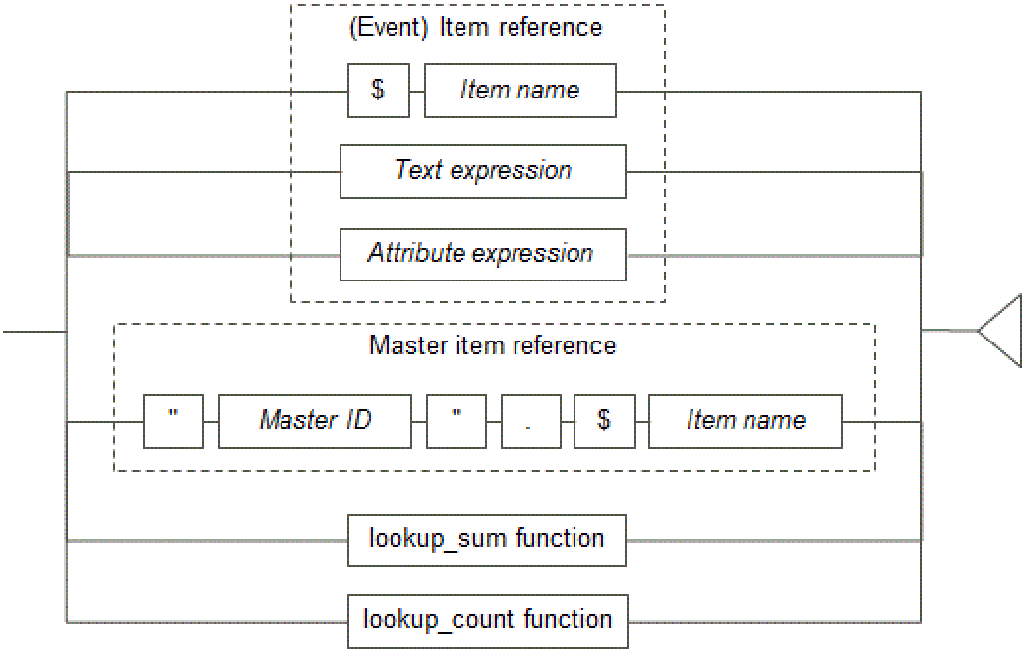
```
"""MEM0001"",""MEM0003""","""John"",""Diana"""
```

- However, if the lookup_sum() function or the lookup_count() function is specified for the output item, items are output without being enclosed between double quotation marks (`"`).

2.7.1 Output Items

Specifies which items in the input events and master data are to be output.

The format used for output item is shown below.



Item reference

Describe an item reference in accordance with the event type format.

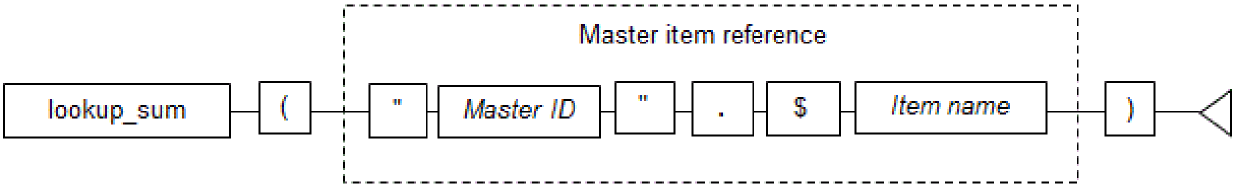
Master item reference

Describe a master item reference in which a master ID and item name are combined.

lookup_sum function

This function returns the sum of the master item contents that results from joining to one master file.

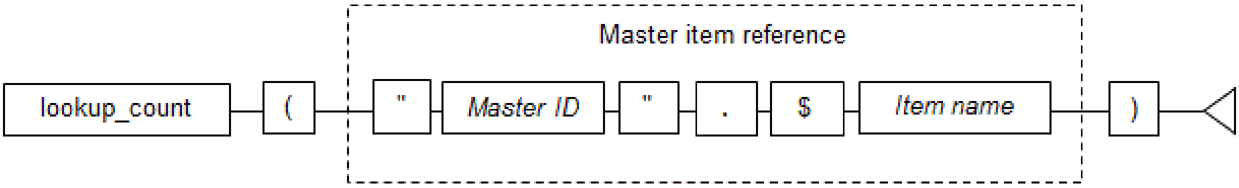
The format used for lookup_sum() function specified in the output expression is shown below.



lookup_count function

This function returns the master item count that results from joining to one master file.

The format used for lookup_count() function specified in the output expression is shown below.



Note

If the lookup_sum() function or the lookup_count() function is specified for an output item, a property alias must be specified after the output item.

See

- Refer to "2.8.4.2 lookup_sum() Function" for details.
- Refer to "2.8.4.3 lookup_count() Function" for details.

2.8 Function Format

This section explains the format of functions.

2.8.1 Function List

The table below lists the function provided by the High-speed Filter.

The table shows the format in which each function can be specified.

Type	Function name	Explanation	Return value (type)	Whether specifiable or not			
				Search expression	Join-relational expression	Output item for lookup()	Output item for output()
String function	rtrim	Removes spaces from the end of a string	String	Y	-	-	-
	string	Converts to a standard format string	String	-	Y	Y	-
Numeric processing function	val	Fetches a numeric value	Numeric	Y	Y	Y	-
lookup function	lookup	Searches master items (without a third argument)	Boolean value	Y	-	-	-
		Searches master items (with a third argument)	Any	Y	-	-	-
	lookup_sum	Calculates the master item sum	Numeric	Y	-	-	Y
	lookup_count	Calculates the master item count	Numeric	Y	-	-	Y
Boolean function	true	Returns Boolean value (true)	Boolean value	Y	-	-	-
	false	Returns Boolean value (false)	Boolean value	Y	-	-	-

Y: Can be specified

-: Cannot be specified

2.8.2 String Functions

This section explains the functions that handle strings.

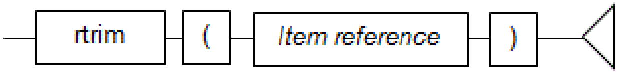
2.8.2.1 rtrim() Function

The rtrim() function returns a string from which the following consecutive characters at the end of a string specified in an item reference have been removed:

- Single-byte space (' ')
- Horizontal tab (HT)
- Line feed (LF)

- Carriage return (CR)

The rtrim() function format is as follows:



See

Refer to "2.4.9 Item References" for details

Return value

If the conversion has operated normally, a string type is returned.

Example

If $*name* is "`Smith Adam `" (where " " is a single-byte space):

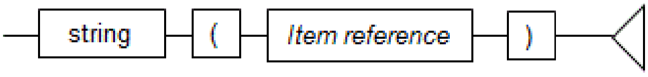
```
rtrim($name)
```

"`Smith Adam`" is output as a string.

2.8.2.2 string() Function

The string() function converts an item reference value to a standard format string.

The string() function format is as follows:



See

Refer to "2.4.9 Item References" for details.

Return value

If the conversion has operated normally, a string type is returned. If the item reference value is "null", "null" is returned.

The string after conversion is in the following format:

Type specified in argument	String after conversion
String type	Not converted
Numeric type	Integer part + decimal part (*1)

*1: The integer part and decimal part are a maximum of 18 digits each.

Example

If the numeric item `$age` is "30":

```
string($age)
```

"30" is output as the string.

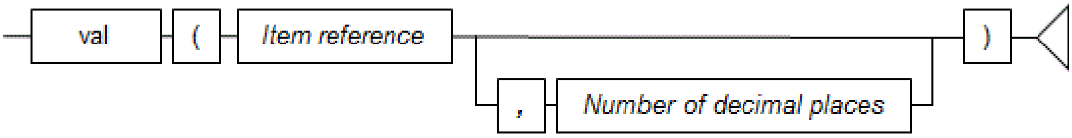
2.8.3 Numeric Processing Functions

This section explains the functions that handle the numeric type.

2.8.3.1 val() Function

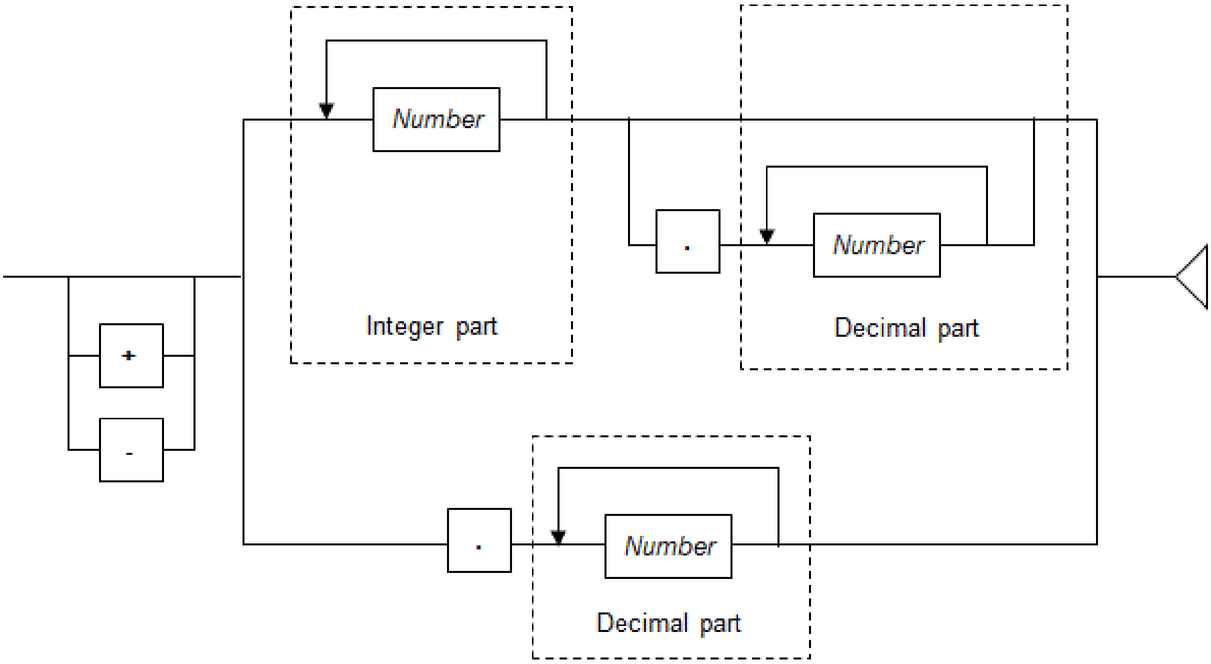
The val() function extracts the numeric values from a string within an item reference.

The format of the val() function is illustrated below.



Point

- The first instance of a string, in the format shown below, found in the string in the text expression is extracted as a numeric value.



- Any commas (`,`) appearing in the integer part are ignored.
- If a decimal point is specified, the decimal places will include all characters appearing from the decimal point onwards till the first instance of a non-numeric character.

- Strings within the text expression that do not contain any numbers are treated as 0.

Note

If the integer part, excluding leading zeros, exceeds 18 digits, an error message is output and the input events are discarded (processing of subsequent input events continues).

See

Refer to "2.4.9 Item References" for details.

Number of decimal places

- If specifying the number of decimal places, digits after the specified number of decimal places are truncated from the return value.
- The range for decimal places is from -18 through 18.
- If the number of decimal places argument is omitted, a value with up to a maximum of 18 decimal places is valid.
- If the number of decimal places is a negative number, the digits to the left of the decimal point (integer part) are truncated.

Example

Consider the following value before truncation.

```
123456789012345678.1234567890123456789
```

Number of decimal places	Value after truncation
When omitted	123456789012345678.123456789012345678
0	123456789012345678
1	123456789012345678.1
-1	123456789012345670
18	123456789012345678.123456789012345678
-18	0

Return value

A numeric value is returned if the function executes successfully.

Example

If `$address` is "`3141 Fairview Park Drive, Falls Church`":

```
val($address)
```

Only the numeric value 3141 is output.

2.8.4 Lookup Functions

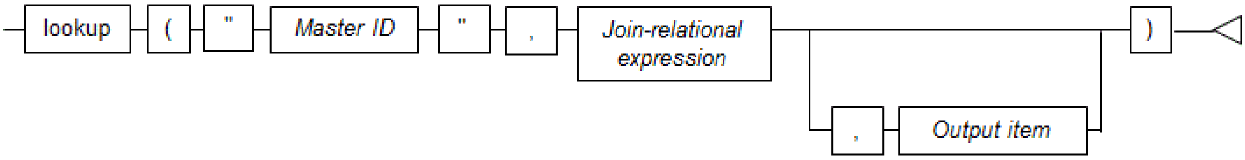
These functions relate to the results joined to one master data.

2.8.4.1 lookup() Function

The lookup() function returns the master item contents from the results joined to the master data.

The function return value type varies in accordance with the contents specified for the third argument output item and in accordance with the existence of specifications.

The lookup() function format is as follows:



Master ID

Specify the development asset ID of the master definition.

Join-relational expression

Describe the join-relational expression used when joining the event and the master.

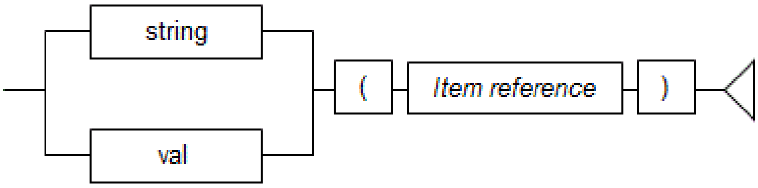
Output item

Specify the master data item to be output.

The string() function format or the val() function format must be used, not the master item reference format.

If an output item is not specified, the existence of master data after the join (true/false) is returned.

The format used for output item is shown below.



Point

If multiple master data fulfill the join-relational expression, search condition evaluation is performed for each of the output item contents. If even one of the multiple master items fulfills the lookup expression, the result of the condition expression is TRUE.

Note

There are no literals indicating Boolean values (true/false). If the lookup() function is used without an output item specification, specify either the true() function or the false() function in the right side of the condition expression.

See

- Refer to "2.6.1 Join-Relational Expression" for information on join-relational expression formats.
- Refer to "2.4.9 Item References" for details.

Return value

If output item specified

If the string() function is specified for the output item, the string type is returned. If the val() function is specified for the output item, the numeric type is returned.

If output item not specified

The following Boolean values are returned:

Return value	Explanation
true	Master data that fulfills the join-relational expression exists. It is the same as the true() function result.
false	Master data that fulfills the join-relational expression does not exist. It is the same as the false() function result.

Example

If the input event C item and the master data X item were joined, that input event is extracted.

```
lookup("Mst", $C == $X) = true()
```

If the input event C item and the master data X item were joined and the Y item of the joined result master matches "Diana", that input event is extracted.

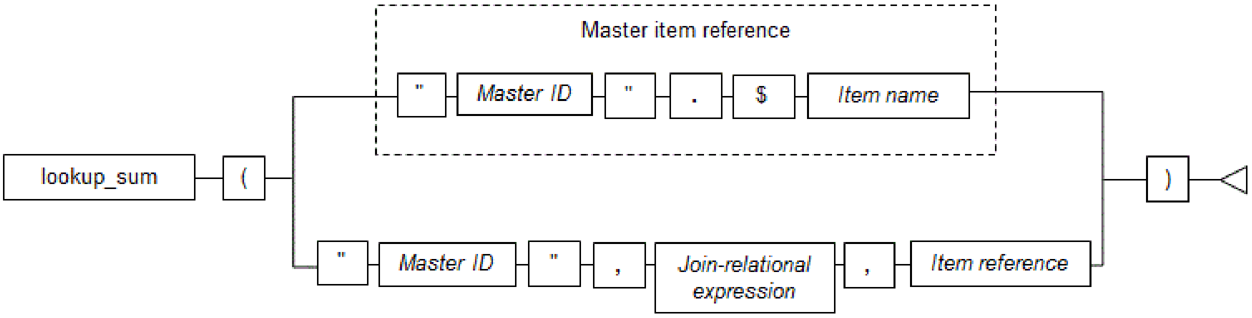
```
lookup("Mst", $C == $X, string($Y)) = "Diana"
```

2.8.4.2 lookup_sum() Function

The lookup_sum() function returns the sum of the contents of master items from the results of joining to master data.

If specified in an output expression, the second and third arguments are omitted and the sum of the contents of the master items joined by the join expression is returned.

The lookup_sum() function format is as follows:



Master ID

Specify the development asset ID of the master definition.

Join-relational expression

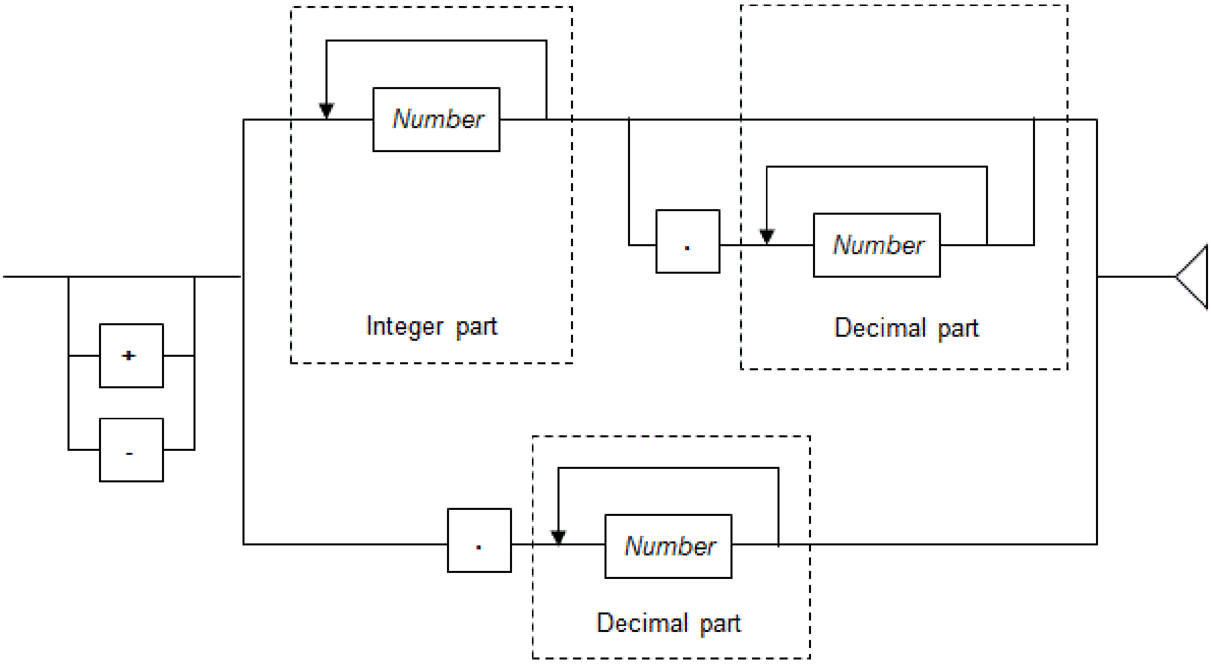
Describe the join-relational expression used when joining events and a master.

Item reference

Specify the master data and items for extracting the sum.

Point

- The first numeric literal format string found is extracted from the master data items as a numeric and added to the sum.



- If strings shown in master data items do not contain numerics, they are handled as 0 (they are not added to the sum).
- If no master data items contain numeric, this function returns "null". If this function is specified in a condition expression, the condition expression is evaluated as being false (conditions not met).
- Commas (`,`) appearing in the integer part are ignored.
- If a decimal point is specified, all subsequent numerics up to the first appearance of a non-numeric character are assumed to be the decimal part.
- Values having a maximum of 18 digits in the decimal part after the decimal point are valid. (Decimal parts that exceed 18 digits are truncated.)

Note

If the integer part, excluding leading zeros, exceeds 18 digits, an error message is output and the input events are discarded (processing of subsequent input events continues).

See

- Refer to "2.6.1 Join-Relational Expression" for information on join-relational expression formats.
- Refer to "2.4.9 Item References" for details.

Return value

If the conversion has operated normally, a numeric type is returned.

Example

If the input event C item and the master data X item were joined and the totaled result for the master data Y item is greater than 100, that input event is extracted.

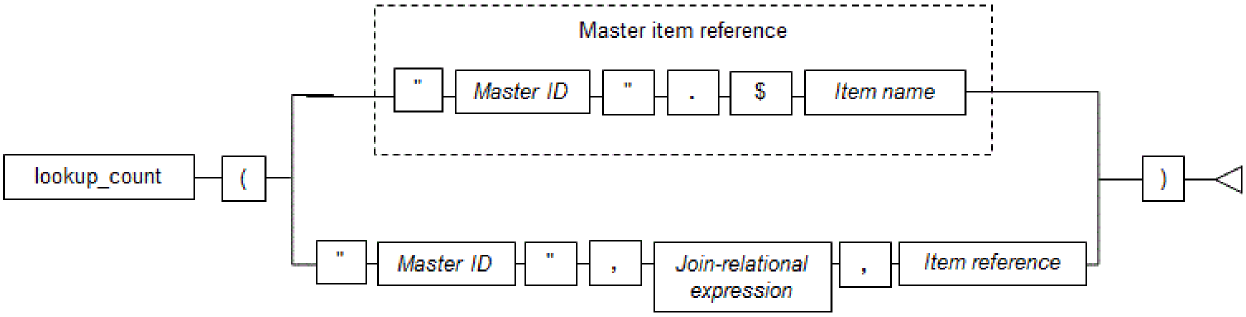
```
lookup_sum("Mst", $C == $X, $Y) > 100
```

2.8.4.3 lookup_count() Function

The lookup_count() function returns the count for the master items from the results of joining to master data.

If specified in an output expression, the second and third arguments are omitted and the count for the master items joined by the join expression is returned.

The lookup_count() function format is as follows:



Master ID

Specify the development asset ID of the master definition.

Join-relational expression

Describe the join-relational expression used when joining events and a master.

Item reference

Specify the master data item to be counted.

Point

- If the content of the specified item is "null", it is not counted.
- If all the contents of the specified item are "null", this function returns 0.

See

- Refer to "2.6.1 Join-Relational Expression" for information on join-relational expression formats.
- Refer to "2.4.9 Item References" for details.

Return value

If the conversion has operated normally, a numeric type is returned.

Example

Example of using the lookup_count() function

If the input event C item and the master data X item were joined, this example detects if the master data Y item count is smaller than 10.

```
lookup_count("Mst", $C == $X, $Y) < 10
```

2.8.5 Boolean Functions

This section explains the functions that return Boolean values.

See

Boolean functions can be specified only in the right side of a condition expression when master data search is performed.

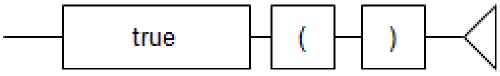
Refer to "2.5.7.4 Master Data Search" for details.

2.8.5.1 true() Function

The true() function returns "true", indicating always TRUE.

Use the true() function as a lookup search (master data search) keyword.

The true() function format is as follows:



Return value

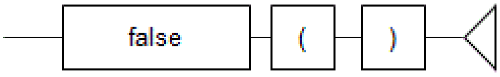
The Boolean value (true) is returned.

2.8.5.2 false() Function

The false() function returns "false", indicating always FALSE.

Use the false() function as a lookup search (master data search) keyword.

The false() function format is as follows:



Return value

The Boolean value (false) is returned.

2.9 Options

This section explains the filter rule options.

2.9.1 Options Overview

The handling of characters during pattern search and string search can be changed by specifying options in the filter rules.

Describe the options before the first ON statement.

Example

Example of describing options

The following is an example of an option description:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rule xmlns="urn:xmlns-fujitsu-com:cspf:bdcep:v1" id="RULE_01 ">
  <comment> Rule definition used by CEP</comment>
  <filter>
    <![CDATA[
      @SkipChar("\n")
      @SeparateChar("\t")
      @ANKmix(true)
      @KNJmix(true)

      on EventType1 {
          ...(...)...
      }
      ...(...)...
    ]]>
  </filter>
  <statements>
     ...(...)...
  </statements>
</rule>
```

Note

Notes on setting filter rule options

- Options are valid for all filter rules described in the <filter> element.
- Describe options before the first ON statement. Options cannot be described after an ON statement.

2.9.2 Options List

The options that can be used are shown below.

Table 2.4 Specifiable options

Option	Item name	Parameter	Example	Explanation
SkipChar	Skip character	String to be excluded from search target (More than one string can be specified.)	`@SkipChar("\n")`	If performing pattern search (string) and complete match of string search, specify characters to be excluded from the search target.
SeparateChar	Separate character	Word separator character	`@SeparateChar("\t")`	If performing pattern search (word), specify the word separator character.
ANKmix	Distinguish between upper-case and lower-case single-byte alphabetics	true or false	`@ANKmix(true)`	Specify the handling of upper-case and lower-case for single-byte alphabetics in search target strings. true If not case-sensitive false If case-sensitive
KNJmix	Distinguish between upper-case and lower-case double-byte alphabetics	true or false	`@KNJmix(true)`	Specify the handling of upper-case and lower-case for double-byte alphabetics in search target strings. true If not case-sensitive false If case-sensitive

2.9.2.1 SkipChar

If filter rules are used to perform pattern search (string) and string search, specify the strings (skip characters) that are excluded as search targets.

Syntax

```
@SkipChar("parameter")
```

Values specified in parameter

- Specify characters, excluding control characters. The characters specified are case-sensitive.
- Specify spaces, horizontal tabs, and line feeds as follows:

Character	Specification method
Single-byte space	`\s`
Double-byte space	`\S`
Horizontal tab	`\t`
Line feed	`\n`

Example

Example when specifying a single-byte space

```
@SkipChar("\s")
```

- There can be more than one character specified. If there are multiple specifications, specify with each string separated by a comma (`,`).

Example

Example when specifying single-byte spaces and double-byte spaces

```
@SkipChar("\s,\S")
```

- Characters can be specified in UTF-8 encoding. If using the character codes to specify characters, express the code with an escape character added. The escape character is "\". If character codes are expressed using multi-byte characters, use single-byte spaces to separate them.

Example

Example when specifying @ (single-byte character)

```
@SkipChar("\40")
```

Example when specifying @ (double-byte character)

```
@SkipChar("\EF \BC \A0")
```

- Use up to 4096 bytes to specify skip characters.

Note

- This option is not valid for condition expression of comparisons between items.
- The following characters cannot be specified:

Prohibited characters
"
, (*1)
\n (*1)
< (*2)
> (*2)
] (*2)
' (*2)

*1: Can be specified as an exception if the input event type to all IF-THEN statements in filter rules is XML.

*2: Can be specified as an exception if the input event type to all IF-THEN statements in filter rules is CSV.

Operation when option is omitted

If this option is omitted, all characters are treated as search targets.

2.9.2.2 SeparateChar

If pattern search (word) is specified in a filter rule search expression, specify the delimiter character (separator character).

Syntax

```
@SeparateChar("parameter")
```

Values specified in parameter

- Specify ASCII characters, excluding control characters, and line feeds and horizontal tabs.
- If the characters shown below are specified, express them by adding an escape character. The escape character is "\".

Character	Specification method
Single-byte space	`\s`
Line feed	`\n`
Horizontal tab	`\t`
Comma	`\,`
Double-quotation marks	`\"`
\ (backslash)	`\\`

Example

Example when specifying a single-byte space

```
@SeparateChar("\s")
```

- There can be more than one character specified. If there are multiple specifications, specify with each string separated by a comma (`,`).

Example

Example when specifying a single-byte space and a horizontal tab

```
@SeparateChar("\s,\t")
```

- Use up to 4096 bytes to specify separator characters.

Note

The following characters cannot be specified:

Prohibited characters
`"  (*1)`
`,  (*1)`
`\n (*1)`

Prohibited characters
< (*2)
> (*2)
] (*2)

*1: Can be specified as an exception if the input event type to all IF-THEN statements in filter rules is XML.

*2: Can be specified as an exception if the input event type to all IF-THEN statements in filter rules is CSV.

Operation when option is omitted

If this option is omitted, it is assumed that the following "separator characters" have been specified:

\t	\n (*1)	\s	\" (*1)	!	$
%	&	'	(	)	*
+	\, (*1)	-	.	/	:
;	< (*2)	=	> (*2)	?	@
[	\\	] (*2)	^	_	`
{	|	}	~		

*1: Not assumed to be a separator character if the input event type to IF-THEN statements is CSV.

*2: Not assumed to be a separator character if the input event type to IF-THEN statements is XML.

2.9.2.3 ANKmix

Specify how upper-case and lower-case are handled for single-byte alphabetic search target strings.

Syntax

```
@ANKmix(parameter)
```

Values specified in parameter

true

Single-byte alphabetics are not case-sensitive.

false

Single-byte alphabetics are case-sensitive.

Operation when option is omitted

If this option is omitted, it is assumed that false is specified.

Note

This option is not valid for condition expression of comparisons between items.

Example

Examples of search processing results when the ANKmix parameter is false (case-sensitive) and when it is true (not case-sensitive)

Search keyword	Search target characters	false: case-sensitive	true: not case-sensitive
ab	ab	Y	Y
	AB	x	Y
	aB	x	Y
	Ab	x	Y
AB	ab	x	Y
	AB	Y	Y
	aB	x	Y
	Ab	x	Y

Y: Hit

x: Not hit

2.9.2.4 KNJmix

Specify how upper-case and lower-case are handled for double-byte alphabetic search target strings.

Syntax

```
@KNJmix(parameter)
```

Values specified in parameter

true

Double-byte alphabetics are not case-sensitive.

false

Double-byte alphabetics are case-sensitive.

Operation when option is omitted

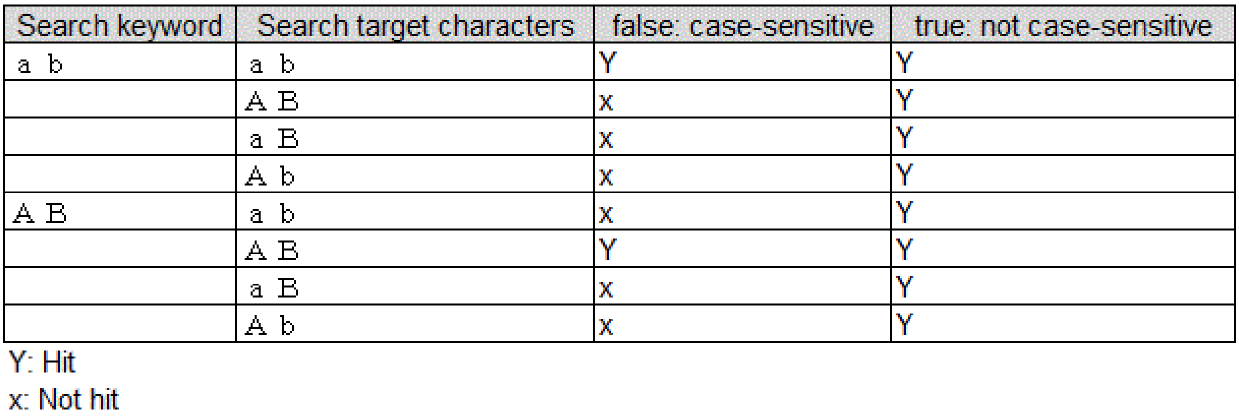
If this option is omitted, it is assumed that false is specified.

Note

This option is not valid for condition expression of comparisons between items.

Example

Examples of search processing results when the KNJmix parameter is false (case-sensitive) and when it is true (not case-sensitive)



Search keyword	Search target characters	false: case-sensitive	true: not case-sensitive
ａ　ｂ	ａ　ｂ	Y	Y
	Ａ　Ｂ	x	Y
	ａ　Ｂ	x	Y
	Ａ　ｂ	x	Y
Ａ　Ｂ	ａ　ｂ	x	Y
	Ａ　Ｂ	Y	Y
	ａ　Ｂ	x	Y
	Ａ　ｂ	x	Y

Y: Hit

x: Not hit

Chapter 3 Input Adapter Reference

This chapter explains the input adapter features provided by Interstage Big Data Complex Event Processing Server (hereafter referred to as "BDCEP") and also explains the event sender application samples.

3.1 Input Adapter Overview

The input adapter receives event data from an event sender application, performs format analysis and log output (only if logging is used), then passes the event data to a high-speed filter.

The input adapters provided by BDCEP are a SOAP adapter, an HTTP adapter, and a socket adapter. The user selects which adapter to use in accordance with the terminal used as the event sender application and the service format.

The table below shows the features of these input adapters and the communication protocol used for data transmission.

Input adapter type	Features	Communication protocol	Characteristics
SOAP adapter	Receives SOAP messages and extracts event data	SOAP (HTTP)	**Versatility: Very good** **Performance: Poor** SOAP communication enables easy linkage to existing SOA systems
HTTP adapter	Receives HTTP requests and extracts event data	HTTP	**Versatility: Good** **Performance: Good** Lightweight compared with SOAP, but response is better
Socket adapter	Receives messages using the proprietary data format of BDCEP, and extracts event data	Proprietary protocol (TCP/IP)	**Versatility: Poor** **Performance: Very good** High throughput enables sending large quantities of events

3.2 About Event Data

This section explains the event data sent to input adapters.

Point

Event sender applications must send event data in accordance with the communication method of each input adapter. Refer to "3.3 Communication Method" for information on input adapter communication methods.

3.2.1 Event Data Contents

The event data posted to an input adapter contains the following information:

Information	Explanation	Mandatory/Optional
Event data format	The format of the event data (CSV or XML) * Not case-sensitive	Mandatory

Information	Explanation	Mandatory/Optional
Event type ID	The development asset ID of the event type definition that indicates the event data structure	Mandatory
Character set	The event data encoding (character code)	(*1)
Event data content	The event data content (data in CSV or XML format) * Maximum data size is 32,000,000 bytes	Mandatory

*1: Posting is mandatory if the character encoding is other than UTF-8

3.2.2 Supported Character Sets

The event data character sets (encoding) supported by input adapters are shown below.

If a character set other than UTF-8 is specified, the input adapter converts the event data contents to UTF-8 for the CEP service. Character sets are not case-sensitive.

Supported character sets	Explanation
Shift_JIS	Shift JIS
EUC-JP	Japanese-language EUC
UTF-8	Unicode (UTF-8)

Note

- If a character set other than the above is specified, the input adapter does not respond with an error if the CEP Server system can recognize the character set. However, in this case, operation is not guaranteed.
- Operation is not guaranteed if the event data encoding does not match the specified character set.

3.3 Communication Method

This section explains the end points (send destination address or port number), send message contents and response message contents (including error contents when an error occurs) for each of the following input adapters:

- SOAP Adapter
- HTTP Adapter
- Socket Adapter

3.3.1 SOAP Adapter

This section explains the SOAP adapter communication method.

3.3.1.1 End Point

A SOAP adapter uses CEP service Web server features to receive event data.

The end point address (URL) is as follows:

```
http://cepServerHostName/cepEngineNameFrontServerService/SoapReceiverService
```

Example

If the CEP Server host name is "bdcep", and the CEP engine name is "CEPengine1":

```
http://bdcep/CEPengine1FrontServerService/SoapReceiverService
```

3.3.1.2 Send Message

Use HTTP protocol (HTTP binding) to send a SOAP message in which event data is stored to the CEP Server.

Information

The SOAP adapter of BDCEP conforms to SOAP 1.1 specification.

SOAP message

The format of SOAP messages posted to the CEP engine is as follows:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header />
    <S:Body>
        <a:notify xmlns:a="http://adapter.front.cep.cspf.fujitsu.com/">
            <type>eventDataFormat</type>
            <eventTypeId>eventTypeID</eventTypeId>
            <data>eventData</data>
        </a:notify>
    </S:Body>
</S:Envelope>
```

SOAP header (S:Header)

Specify a blank element.

SOAP body (S:Body)

Describe the event data information below in the notify element value and specify it in the SOAP body.

Specify "`http://adapter.front.cep.cspf.fujitsu.com/`" in the `xmlns` attribute (XML namespace) of the notify element.

Send information	Specification method	Specification example
Event data format	Specify as the **`type`** element value	If the event data is in CSV format: **`<type>CSV</type>`**
Event type ID	Specify as the **`eventTypeId`** element value	If the event type (development asset ID of the event type definition) is EVENTTYPE_01: **`<eventTypeId>EVENTTYPE_01</eventTypeId>`**
Event data content	Specify as the **`data`** element value	**`<data>MEM0001,1010,1</data>`**

HTTP request

Specify the CEP engine that posts the event data.

```
POST path HTTP/version
```

path

The path part of the end point address.

version

The HTTP protocol version.

Point

BDCEP supports HTTP protocol versions 1.0 and 1.1.

Example

If the CEP engine name is "`CEPengine1`":

```
POST /CEPengine1FrontServerService/SoapReceiverService HTTP/1.1
```

Request header

Specify event data information as follows:

Send information	Specification method	Specification example
Character set	Specify in the **Content-Type** header (Can be omitted for UTF-8)	If the event data encoding is Shift JIS: **Content-Type: text/xml; charset=Shift_JIS**

Note

If a character set is not specified, the encoding is not converted. Therefore, even if the encoding of the sent event data is not UTF-8, the high-speed filter and complex event processing operate interpreting the input event as being in UTF-8.

Message body

Specify the SOAP message (as above).

Example

The send message for the following scenario is shown below:

- CEP Server host name: bdcep
- CEP engine name: CEPengine1
- Event data format: CSV format
- Event type ID: EVENTTYPE_01
- Character set: Shift JIS

```
POST /CEPengine1FrontServerService/SoapReceiverService HTTP/1.1
Host: bdcep
Content-Type: text/xml; charset=Shift_JIS
Content-Length: nnnn

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header />
    <S:Body>
```

```
        <a:notify xmlns:a="http://adapter.front.cep.cspf.fujitsu.com/">
            <type>CSV</type>
            <eventTypeId>EVENTTYPE_01</eventTypeId>
            <data>MEM0001,1010,1</data>
        </a:notify>
    </S:Body>
</S:Envelope>
```

3.3.1.3 Response Message

After an HTTP request is sent, a response message (SOAP message) posted from the CEP Server is received.

The HTTP response and the information posted in the message body are shown below.

SOAP message

The format of the SOAP message posted from the CEP engine is shown below.

Note that newlines have been inserted here for this example, but newlines are not inserted in the actual messages.

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/">
<return>returnMessage</return>
</ns2:notifyResponse>
</S:Body>
</S:Envelope>
```

returnMessage

This message shows the SOAP adapter processing results.

If processed normally, the message is "Code=0 Message=Message sending is completed normally."

HTTP response

An HTTP response message is posted.

```
HTTP/version statusCode supplementaryMessage
```

version

HTTP protocol version.

statusCode

HTTP request result.

supplementaryMessage

Supplementary message in accordance with the status code.

Example

If event data send ended normally:

```
HTTP/1.1 200 OK
```

Message body

A SOAP message (as above) is posted.

Point

When an error occurs, a message is output to the syslog and the engine log. Refer to "3.4 Error Processing" for information on error processing.

Various messages, in accordance with the processing result, are as follows:

Processing result	Message (*1)
Normal end	**200 OK**
	Code=0 Message=Sending message completed normally.
	None
Format error The event data format does not exist.	**200 OK**
	Code=-100 Message=Event type format is not defined.
	cep10306e: Event Type Format is not defined. EngineId=*cepEngineName*
Processing result error Log output processing failed.	**200 OK**
	Code=-200 Message=Logging failed.
	cep10401e: The Log was not able to be output. EngineId=*cepEngineName*, EVENT=*receivedEvent*, ERRORINFO=*internalInformation* Note: Newlines in received events are converted to &#012; before output. If an event exceeds the length that can be output to the syslog, only partial event information is output to the syslog.
Format error The event type ID does not exist.	**200 OK**
	Code=-300 Message=Event type id is not defined.
	cep10305e: Event Type ID is not defined. EngineId=*cepEngineName*
Format error A format other than CSV or XML is specified for the event data format.	**200 OK**
	Code=-400 Message=Unknown event format [*eventFormat*].
	cep10114e: Unknown Event format. EngineId=*cepEngineName*, format=*eventFormat*
Settings content discrepancy The event type ID is not known.	**200 OK**
	Code=-500 Message=Unknown event type id [*eventType*].
	cep10108e: Event type is not found. EngineId=*cepEngineName*, eventType=*eventType*
Format error The character set is not known.	**415 Unsupported Media Type**
	None
	None
Settings content discrepancy The event data type does not match the deployed event type definition.	**200 OK**
	Code=-700 Message=It differs from the registered format [*eventFormat*].
	cep10116e: It differs from the registered format. EngineId=*cepEngineName*, format=*eventFormat*
Busy status The CEP Server is temporarily or permanently overloaded.	**200 OK**
	Code=-800 Message=Server is busy now.
	cep10301w: FrontServer is busy. Event is aborted. EngineId=*cepEngineName* or

Processing result	Message (*1)
	cep10302e: FrontServer is continuously busy. Event is aborted. EngineId=*cepEngineName*
Format error The event data size exceeds 32,000,000 bytes.	**200 OK**
	Code=-900 Message=Bad Event data size [*eventDataSize*].
	cep10310e: Event Data Size is over. EngineId=*cepEngineName*, Size=*eventDataSize*
CEP Server not running	**200 OK**
	Code=-1000 Message=Server is not Running.
	cep10300w: FrontServer is not running. Event is aborted. EngineId=*cepEngineName*

*1: The contents of each row are as follows:

Top row: HTTP status code + supplementary message

Middle row: Return message included in SOAP message (variable information shown in *italic*)

Bottom row: Message output to syslog and engine log (variable information shown in *italic*)

3.3.1.4 Notes

This section explains notes related to the use of the SOAP adapter.

Number of simultaneous connections

If multiple event sender applications connect to a SOAP adapter simultaneously, the maximum number of simultaneous connections to the SOAP adapter might be reached. In this case, the suspended status temporarily occurs for subsequent connection requests from event sender applications.

Since the SOAP adapter responds to suspended status connection requests in accordance with the processing of the received event data, no action is required by the event sender application.

Information

The maximum number of simultaneous connections to a SOAP adapter is 50.

Since SOAP adapters and HTTP adapters share use of the Web server features of the CEP Server, the actual number of connections is the combined number of SOAP adapter and HTTP adapter connections.

Communication protocol

The SOAP adapter communication protocols do not support HTTPS (HTTP over SSL/TLS).

3.3.2 HTTP Adapter

This section explains the HTTP adapter communication method.

3.3.2.1 End Point

An HTTP adapter uses CEP service Web server features to receive event data.

The end point address (URL) is as follows:

```
http://cepServerHostName/cepEngineNameFrontServerService/HttpReceiver
```

Example

In this example, the CEP Server host name is "bdcep", and the CEP engine name is "CEPengine1".

```
http://bdcep/CEPengine1FrontServerService/HttpReceiver
```

3.3.2.2 Send Message

Use an HTTP request to send event data to the CEP Server.

HTTP request

Specify the CEP engine that posts the event data.

```
POST path HTTP/version
```

path

The path part of the end point address.

version

The HTTP protocol version.

Point

BDCEP supports HTTP protocol versions 1.0 and 1.1.

Example

If the CEP engine name is "CEPengine1":

```
POST /CEPengine1FrontServerService/HttpReceiver HTTP/1.1
```

Request header

Specify event data information as follows:

Send information	Specification method	Specification example
Event data format	Specify in the **TYPE** header	If the event data is in CSV format: **TYPE: CSV**
Event type ID	Specify in the **EVENT-TYPE-ID** header	If the event type (development asset ID of the event type definition) is EVENTTYPE_01: **EVENT-TYPE-ID: EVENTTYPE_01**
Character set	Specify in the **Content-Type** header (Can be omitted for UTF-8)	If the event data encoding is Shift JIS, and the format is CSV: **Content-Type: text/plain; charset=Shift_JIS** If the event data encoding is Japanese-language EUC, and the format is XML: **Content-Type: text/xml; charset=EUC-JP**

Note

If a character set is not specified, the encoding is not converted. Therefore, even if the encoding of the sent event data is not UTF-8, the high-speed filter and complex event processing operate interpreting the input event as being in UTF-8.

Message body

Specify the event data content.

Example

The send message for the following scenario is shown below:

- CEP Server host name: bdcep
- CEP engine name: CEPengine1
- Event data format: CSV format
- Event type ID: EVENTTYPE_01
- Character set: Shift JIS

```
POST /CEPengine1FrontServerService/SoapReceiverService HTTP/1.1
Host: bdcep
TYPE: CSV
EVENT-TYPE-ID: EVENTTYPE_01
Content-Type: text/plain; charset=Shift_JIS
Content-Length: nnnn

MEM0001,1010,1
```

3.3.2.3 Response Message

After an HTTP request is sent, a response message posted from the CEP Server is received.

The HTTP response and the information posted in the message body are shown below.

HTTP response

An HTTP response message is posted.

```
HTTP/version statusCode supplementaryMessage
```

version

HTTP protocol version.

statusCode

HTTP request result.

supplementaryMessage

Supplementary message in accordance with the status code.

Example

If event data send ended normally:

```
HTTP/1.1 200 OK
```

Message body

A message showing the HTTP adapter processing results is posted.

Example

If event data send ended normally:

```
Code=0 Message=Message sending is completed normally.
```

Point

When an error occurs, a message is output to the syslog and the engine log. Refer to "3.4 Error Processing" for information on error processing.

Various messages, in accordance with the processing result, are as follows:

Processing result	Message (*1)
Normal end	**200 OK**
	Code=0 Message=Sending message completed normally.
	None
CEP Server not running	**200 OK**
	Code=-1000 Message=Server is not Running.
	cep10300w: FrontServer is not running. Event is aborted. EngineId=*cepEngineName*
Format error The event data format does not exist.	**400 Bad Request**
	Code=-100 Message=Event type format is not defined.
	cep10306e: Event Type Format is not defined. EngineId=*cepEngineName*
Format error The event type ID does not exist.	**400 Bad Request**
	Code=-300 Message=Event type id is not defined.
	cep10305e: Event Type ID is not defined. EngineId=*cepEngineName*
Format error A format other than CSV or XML is specified for the event data format.	**400 Bad Request**
	Code=-400 Message=Unknown event format [*eventFormat*].
	cep10114e: Unknown Event format. EngineId=*cepEngineName*, format=*eventFormat*
Format error The event data size exceeds 32,000,000 bytes.	**400 Bad Request**
	Code=-900 Message=Bad Event data size [*eventDataSize*].
	cep10310e: Event Data Size is over. EngineId=*cepEngineName*, Size=*eventDataSize*
Settings content discrepancy The event type ID is not known.	**400 Bad Request**
	Code=-500 Message=Unknown event type id [*eventType*].

Processing result	Message (*1)
	cep10108e: Event type is not found. EngineId=*cepEngineName*, eventType=*eventType*
Settings content discrepancy The event data format does not match the deployed event type definition.	**400 Bad Request**
	Code=-700 Message=It differs from the registered format [*eventFormat*].
	cep10116e: It differs from the registered format. EngineId=*cepEngineName*, format=*eventFormat*
Format error The character set is not known.	**415 Unsupported Media Type**
	Code=-600 Message=Charset is Abnormal [*characterSetName*].
	cep10304e: Charset Name is abnormal. EngineId=*cepEngineName*,CharsetName=*characterSetName*
Processing result error Log output processing failed.	**500 Internal Server Error**
	Code=-200 Message=Logging failed.
	cep10401e: The Log was not able to be output. EngineId=*cepEngineName*, EVENT=*receivedEvent*, ERRORINFO=*internalInformation* Note: Newlines in received events are converted to &#012; before output. If an event exceeds the length that can be output to the syslog, only partial event information is output to the syslog.
Busy status The CEP Server is temporarily or permanently overloaded.	**503 Service Unavailable**
	Code=-800 Message=Server is busy now.
	cep10301w: FrontServer is busy. Event is aborted. EngineId=*cepEngineName* or cep10302e: FrontServer is continuously busy. Event is aborted. EngineId=*cepEngineName*

*1: The contents of each row are as follows:

Top row: HTTP status code + supplementary message

Middle row: HTTP response message body (variable information shown in *italic*)

Bottom row: Message output to syslog and engine log (variable information shown in *italic*)

3.3.2.4 Notes

This section explains notes related to the use of the HTTP adapter.

Number of simultaneous connections

If multiple event sender applications connect to an HTTP adapter simultaneously, the maximum number of simultaneous connections to the HTTP adapter might be reached. If so, the suspended status temporarily occurs for subsequent connection requests from event sender applications.

Since the HTTP adapter responds to suspended status connection requests in accordance with the processing of the received event data, no action is required by the event sender application.

Information

The maximum number of simultaneous connections to an HTTP adapter is 50.

Since SOAP adapters and HTTP adapters share use of the Web server features of the CEP Server, the actual number of connections is the combined number of SOAP adapter and HTTP adapter connections.

Communication protocol

The HTTP adapter communication protocols do not support HTTPS (HTTP over SSL/TLS).

3.3.3 Socket Adapter

This section explains the socket adapter communication method.

Information

Due to the following performance advantages, use of a socket adapter can achieve higher throughput than with SOAP and HTTP adapters:

- The event sender application can send multiple event data while connected to the CEP Server (input adapter). Since this can eliminate overheads associated with establishing and ending TCP/IP connections, throughput improves.
- Event sender applications can send event data continuously without waiting for reception messages from the CEP Server (input adapter). Since this eliminates the time lag associated with waiting for a response from the CEP Server, throughput improves. (Only if the input adapter does not use logging)

3.3.3.1 End Point

The socket adapter end point is a combination of the CEP Server IP address and the socket adapter port (any TCP port).

If the socket adapter is used, the listening port for event data must be specified as the socket adapter port in the engine configuration file.

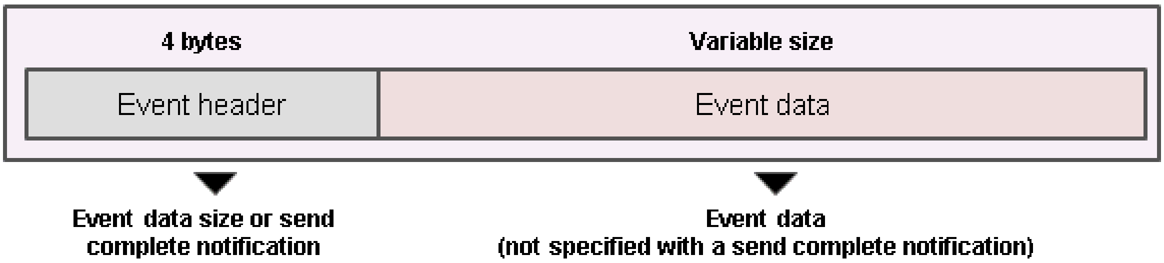
See

Refer to Section 9.1.1, "Engine Configuration File" in the *User's Guide* for information on how to specify the socket adapter port.

3.3.3.2 Send Message

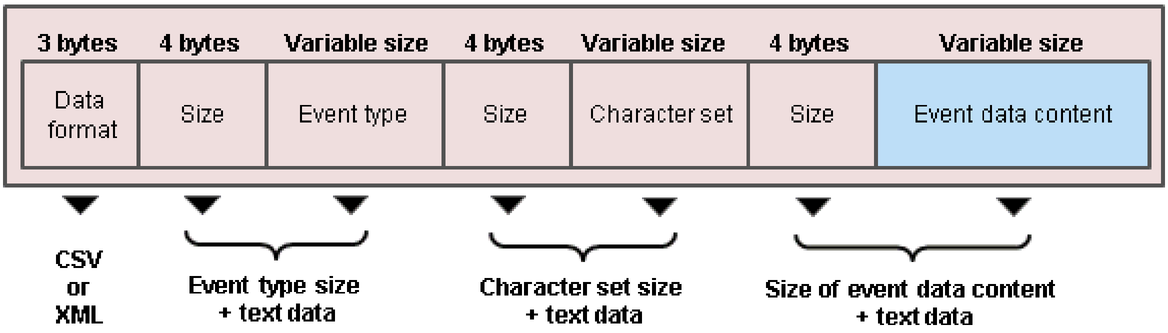
This section explains the messages sent to the socket adapter.

Send message format


4 bytes
Variable size
Event header
Event data
Event data size or send complete notification
Event data
(not specified with a send complete notification)

Data field	Size	Specified value
Event header	4 bytes	Specify the data size of the event data field (bytes). Or, specify 0 to post a **send complete notification** (indicating event sending is completed) to the socket adapter. Possible range is 0 to 33,000,000.
Event data	Variable size	Store the event data (described below). Specify this field if the event header field is other than 0 (send complete notification).

Event data field format



Data field		Size	Specified value
Event data format		3 bytes	Specify CSV or XML (string). The format can also be specified in lower-case (csv or xml).
Event type ID	Size	4 bytes	Specify the string length of the event type ID (bytes).
	Data	Variable size	Specify the event type ID text data.
Character set	Size	4 bytes	Specify the string length of the character set (bytes). If the event data encoding is UTF-8, specify 0.
	Data	Variable size	Specify the character set text data. If UTF-8, omit this field.
Event data content	Size	4 bytes	Specify the size of the event data content (bytes).
	Data	Variable size	Specify the event data text data.

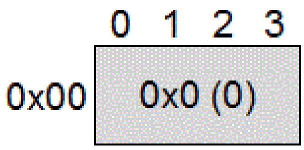
Example

In the following case, the send message byte array is as follows:

- Event data format: CSV format
- Event type ID: EVENTTYPE_01
- Character set: Shift JIS

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	0x32 (50)				C	S	V	0xC (12)				E	V	E	N	T
0x10	T	Y	P	E	_	0	1	0x9 (9)				S	h	i	f	t
0x20	_	J	I	S	0xE (14)				M	E	M	0	0	0	1	,
0x30	1	0	1	0	,	1										

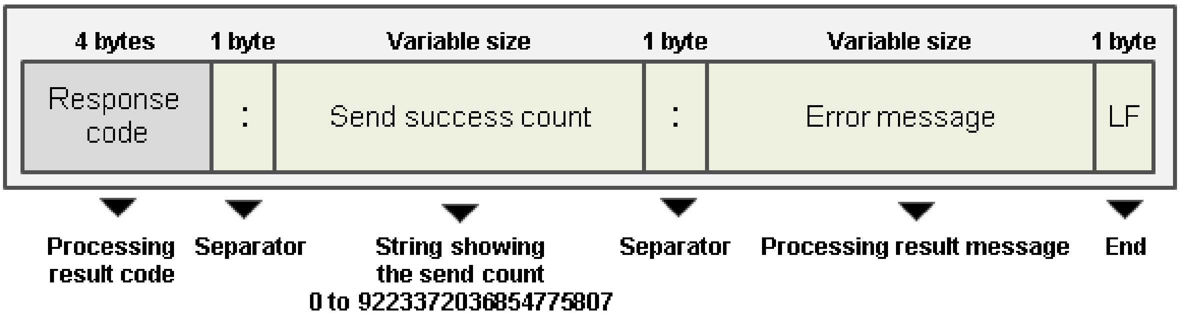
The send complete notification byte array is as follows:



3.3.3.3 Response Message

This section explains the messages posted from the socket adapter.

Response message format



Data field	Size	Specified value
Response code	4 bytes	Stores a 4-digit string showing the input adapter processing results.
<Separator>	1 byte	Stores a colon (:) (0x3A) as the separator character between the previous and next fields.
Send success count	Variable size (0 to 19 bytes)	Stores the processing success count for event data at the input adapter (upper limit: LONG_MAX). This field is posted as a decimal string. For example, if the count is 0, one byte indicating "0" (0x30) is stored. If the maximum count, 19 bytes indicating 2^{63} - 1 is stored.
<Separator>	1 byte	Stores a colon (:) (0x3A) as the separator character between the previous and next fields.
Error message	Variable size	Stores the error message string.
<End>	1 byte	Stores a line feed (LF) (0x0A) as the error message end character.

Point

When an error occurs, a message is output to the syslog and the engine log. Refer to "3.4 Error Processing" for information on error processing.

Response codes and various messages, in accordance with the processing result, are as follows:

Processing result	Response code and message (*1)
Normal end	**0000**
	Sending message completed normally.
	None
Format error	**M001**
	Bad Message data size [*dataSize*].

Processing result	Response code and message (*1)
The data size exceeds 33,000,000 bytes.	cep10308e: Message Data Size is over. EngineId=*cepEngineName*, Size=*dataSize*
Settings content discrepancy The event type ID is not known.	**M002**
	Unknown event type id [*eventType*].
	cep10108e: Event type is not found. EngineId=*cepEngineName* ,eventType=*eventTypeId*
Format error A format other than CSV or XML is specified for the event data format.	**M003**
	Unknown event format [*eventFormat*].
	cep10114e: Unknown Event format. EngineId=*cepEngineName*, format=*eventFormat*
Format error Send message format error	**M004**
	Data format is Abnormal.
	cep10303e: Event data cannot read. Event is aborted. EngineId=*cepEngineName*, ClientIP=*eventSenderIpAddress*
Format error The character set is not known.	**M005**
	Charset is Abnormal [*characterSetName*].
	cep10304e: Charset Name is abnormal. EngineId=*cepEngineName*, CharsetName=*characterSetName*
Processing result error An error occurred when decoding event data.	**M006**
	Decoding event data failed.
	cep10309e: Decoding event data failed. EngineId=*cepEngineName*
Settings content discrepancy The event data type does not match the deployed event type definition.	**M007**
	It differs from the registered format [*eventFormat*].
	cep10116e: It differs from the registered format. EngineId=*cepEngineName*, format=*eventFormat*
Format error The event data size exceeds 32,000,000 bytes.	**M008**
	Bad Event data size [*eventDataSize*].
	cep10310e: Event Data Size is over. EngineId=*cepEngineName*, Size=*eventDataSize*
CEP server not running	**S001**
	Server is not Running.
	cep10300w: FrontServer is not running. Event is aborted. EngineId=*cepEngineName*
Busy status The CEP Server is temporarily or permanently overloaded.	**S002**
	Server is busy now.
	cep10301w: FrontServer is busy. Event is aborted. EngineId=*cepEngineName* or cep10302e: FrontServer is continuously busy. Event is aborted. EngineId=*cepEngineName*
Processing result error Log output processing failed.	**S003**
	Logging failed.
	cep10401e: The Log was not able to be output. EngineId=*cepEngineName*, EVENT=*receivedEvent*, ERRORINFO=*internalInformation*

Processing result	Response code and message (*1)
	Note: Newlines in received events are converted to &#012; before output. If an event exceeds the length that can be output to the syslog, only partial event information is output to the syslog.

*1: The contents of each row are as follows:

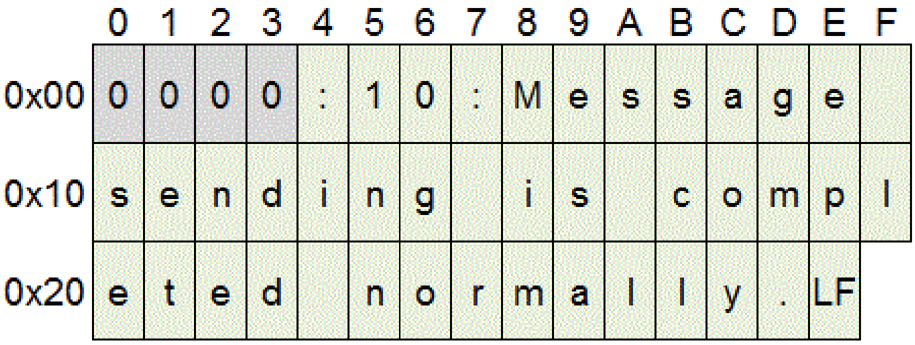
Top row: Response code

Middle row: Error message (variable information shown in *italic*)

Bottom row: Message output to syslog and engine log (variable information shown in *italic*)

Example

If 10 event data are sent and then a send complete notification is sent, the received message byte array is as follows when message send ends normally:



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	0	0	0	0	:	1	0	:	M	e	s	s	a	g	e	
0x10	s	e	n	d	i	n	g		i	s		c	o	m	p	l
0x20	e	t	e	d		n	o	r	m	a	l	l	y	.	LF	

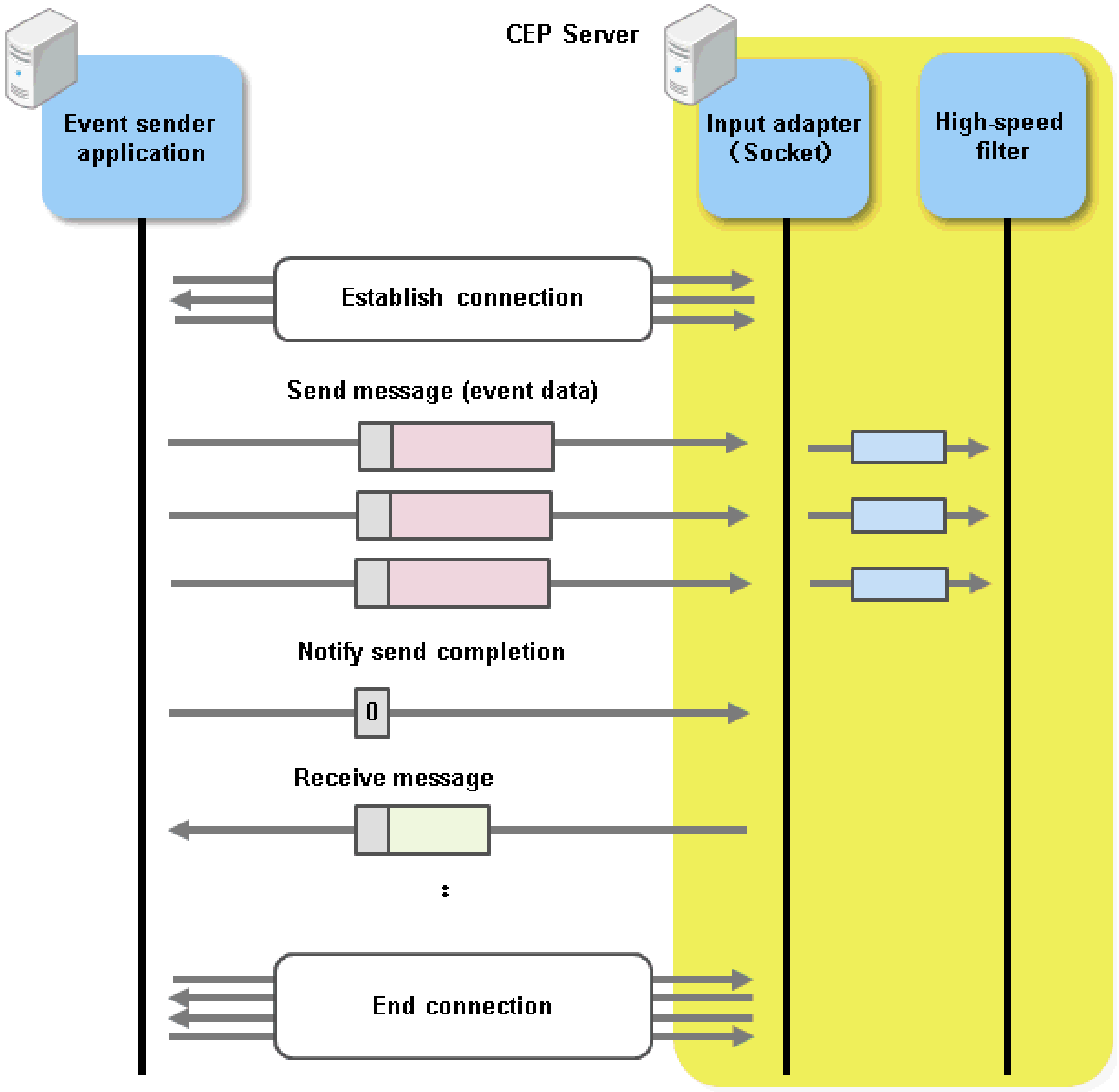
3.3.3.4 Socket Communication Processing Procedures

This section explains the procedures when a socket adapter is used, from connection using socket communication (TCP/IP) to disconnection.

The event sender application communicates in accordance with the procedures (processing flow) shown below.

Processing procedure:

1. Connect to the CEP Server socket adapter. (**Establish connection**)
2. Send event data. (**Send message**)
3. Send next event data (if required). (Repeat step 2)
4. Send a notification. (**Send complete notification**)
5. Receive response message from the CEP Server. (**Receive message**)
6. Send next event data (if required). (Go back to step 2)
7. If all event data sends are completed, close the connection. (**End connection**)



See

- Refer to "3.3.3.1 End Point" for information on the input adapter end point at the connection destination when establishing a connection.
- Refer to "3.3.3.2 Send Message" for information on message sending and the send complete notification.
- Refer to "3.3.3.3 Response Message" for information on message reception.

3.3.3.5 Notes

This section explains notes related to the use of the socket adapter.

Sending messages continuously

If an error is detected at the socket adapter, a response message is posted before a send complete notification is received and the connection is closed.

In this case, the event sender application cannot continue to send messages (an error occurs for data sending). Therefore, have the event sender application interrupt message sending, then receive a response message from the socket adapter.

Sending messages that do not match the size information

If event data is sent that is less than the data size set in the "size information", the CEP Server will wait to receive the correct amount of data, and does not post a response message until the appropriate amount of data is sent.

In the event sender application, ensure that the "size information" matches the event data size.

Number of simultaneous connections

There is no maximum number of simultaneous socket adapter connections. Connections can be made up to the maximum number of operating system file descriptors.

However, to ensure efficient resource usage, close connections when the event sender application is not continuing to send data.

Point

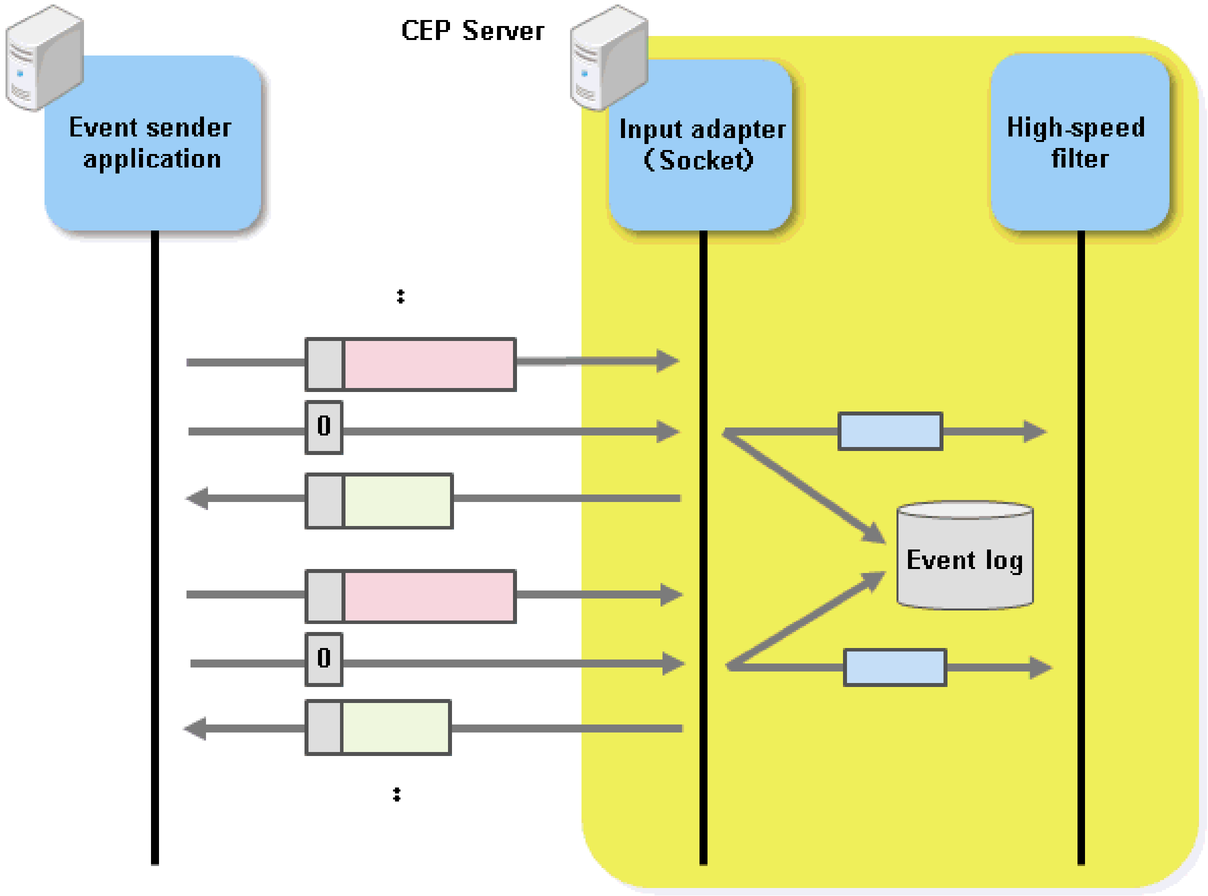
The CEP Server side does not close connections unless an error is detected at the socket adapter.

Using logging at the input adapter

The socket adapter outputs logs at the time of a send complete notification.

Therefore, have the event sender application initiate a send complete notification for each individual message and receive a response message, as shown below.

If a send complete notification is not sent nor a response message received for each message, the send success count for response messages might be inaccurate.



Performance considerations

When small packets are sent via TCP/IP communication consecutively, communication performance may deteriorate due to two operations: the Nagle algorithm, which collects send messages until it receives ACK from the receiver and then attempts to send them, and the TCP delayed ACK, which attempts to return multiple ACK at one time.

To prevent consecutive sending of small packets, use a buffered stream provided by the programming language (such as java.io.BufferedOutputStream in Java) in the event sender application.

Even if a buffered stream is used, communication performance may deteriorate for the same reason if a send complete notification is sent for each message and a response message is received. In such cases, consider setting the TCP_NODELAY option, which disables the Nagle algorithm, in the event sender application. In Java, you can set this using the setTcpNoDelay method of the java.net.Socket class.

3.4 Error Processing

If the event sender application receives a response message indicating an error, take action according to the type of error.

Note that developers themselves must resolve error processing relating to ordinary socket communication (TCP/IP).

CEP Server not running

At the CEP Server, execute cepdispeng (with the -a option specified) to check if the CEP engine has started. If a socket adapter is being used, check the command results to see if the CEP engine listening on the specified TCP port has started.

If not started, use cepstarteng to start the CEP Server.

Format error

Check if the data format of the message sent from the event sender application conforms to the contents of "3.2 About Event Data". Also check if the send message specification method is compliant with the input adapter type.

If there is an error, reconfigure the event sender application.

Settings content discrepancy (event type ID does not match)

At the CEP Server, execute cepdispeng (with the `-i` option specified) to check if the event type ID of the send message matches the event type definition (development asset ID) deployed to the CEP engine.

If there is an error, correct so that they match.

Settings content discrepancy (event data format does not match)

At the CEP Server, execute cepgetrsc (with `eventtype` and the `-n` option specified) to check if the event data format of the send message (CSV or XML) matches the event data format in the deployed event type definition.

If there is an error, correct so that they match.

Processing result error (decoding error)

Check if the encoding of the event data sent from the event sender application is the same as the specified character set.

If different, specify the same character set.

Processing result error (log output failure)

Refer to the syslogs or the engine logs at the CEP Server and check the detailed error information.

Refer to *Messages* for information on the appropriate action.

Busy status

Refer to the syslogs or the engine logs at the CEP Server and check if the input adapter overload is temporary or permanent.

Refer to *Messages* for information on the appropriate action.

See

Refer to Chapter 8, "Command Reference" in the *User's Guide* for information on the cepdispeng, cepstarteng, and cepgetrsc commands.

3.5 Sample Programs

This section provides event sender application sample programs for each of the input adapters listed below.

Refer to these samples when developing event sender applications.

- SOAP Adapter
- HTTP Adapter
- Socket Adapter

3.5.1 SOAP Adapter

Sample program source code:

```
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class SoapClient {
    URL soapAdapterUrl = null;
    String url = "http://%HOSTNAME%/%ENGINE%FrontServerService/SoapReceiverService";
    HttpURLConnection con = null;

    public static void main(String[] args) {
        String hostName = "";
        String engineName = "";
        String dataType = "";
        String charSet = "";
        String eventTypeId = "";
        String data = "";
        long   lWait = 10;

        try {
            int loop = 0;
            if (args.length != 8) {
                System.out.println("param is Abnormal");
                return;
            }

            hostName    = String.valueOf(args[0]);
            engineName  = String.valueOf(args[1]);
            dataType    = String.valueOf(args[2]);
            charSet     = String.valueOf(args[3]);
            eventTypeId = String.valueOf(args[4]);
            data        = String.valueOf(args[5]);
            lWait       = Long.valueOf(args[6]);
            loop        = Integer.valueOf(args[7]);

            SoapClient sc = new SoapClient(hostName, engineName);
            sc.sendMessage(data, dataType, eventTypeId,charSet, loop, lWait);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public SoapClient(String hostName, String engineName) throws IOException {

        try {
            String wkUrl = url.replaceAll("%ENGINE%", engineName);
            wkUrl = wkUrl.replaceAll("%HOSTNAME%", hostName);

            soapAdapterUrl = new URL(wkUrl);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    private HttpURLConnection open(String charSet) throws IOException {

        HttpURLConnection con = null;
        con = (HttpURLConnection) soapAdapterUrl.openConnection();

        con.setRequestMethod("POST");
        con.setRequestProperty("content-type", "text/xml;charset=" + charSet);
        con.setDoOutput(true);
        con.connect();

        return con;
    }

    private void sendMessage(String baseData, String dataType, String eventTypeId,
        String charSet, int loop, long lWait) throws Exception {

        BufferedOutputStream bos = null;
        InputStreamReader ir1 = null;
        BufferedReader br1 = null;

        try {
            for (int i = 0; i < loop; i++) {
                con = this.open(charSet);
                bos = new BufferedOutputStream(con.getOutputStream());

                String data = baseData.replaceAll("%COUNTER%", String.valueOf(i));
                String MSG
                    = "<S:Envelope xmlns:S=\"http://schemas.xmlsoap.org/soap/envelope/\">"
                    + "<S:Header />"
                    + "<S:Body>"
                    + "<a:notify xmlns:a=\"http://adapter.front.cep.cspf.fujitsu.com/\">"
                    + "<type>" + dataType + "</type>"
                    + "<eventTypeId>" + eventTypeId + "</eventTypeId>"
                    + "<data>" + data + "</data>"
                    + "</a:notify>"
                    + "</S:Body>"
                    + "</S:Envelope>";
                bos.write(MSG.getBytes(charSet));

                bos.flush();
                bos.close();

```

```
100                     ir1 = new InputStreamReader(con.getInputStream());
101                     br1 = new BufferedReader(ir1);
102 
103                     String line;
104                     while ((line = br1.readLine()) != null) {
105                         System.out.println(line);
106                     }
107 
108                     br1.close();
109                     ir1.close();
110 
111                     con.disconnect();
112                     Thread.sleep(lWait);
113                 }
114 
115             } catch (MalformedURLException e) {
116                 e.printStackTrace();
117                 throw e;
118             } catch (IOException e) {
119                 e.printStackTrace();
120                 throw e;
121             } catch (InterruptedException e) {
122                 e.printStackTrace();
123                 throw e;
124             } finally {
125                 try {
126                     if (br1 != null) {
127                         br1.close();
128                     }
129                     if (ir1 != null) {
130                         ir1.close();
131                     }
132                 } catch (IOException e) {
133                     e.printStackTrace();
134                     return;
135                 }
136             }
137 
138             return;
139         }
140     }
```

Detailed explanation of source code:

- **main method**
    - Line numbers 30 to 37

      The following data is obtained from the arguments at runtime:

Argument	Variable	Use
1	*hostName*	CEP Server host name
2	*engineName*	CEP engine name
3	*dataType*	Event data format
4	*charSet*	Character set
5	*eventTypeId*	Event type ID
6	*data*	Event data `%COUNTER%` included in event data needs to be substituted with loop count so that different event data shall be sent at each loop.

Argument	Variable	Use
7	*lWait*	Data transmission wait time
8	*loop*	Data transmission count

  - Line number 39

    Creates a SoapClient object.

  - Line number 40

    Calls a method for sending event data.

    Event data, character set, data transmission count, and data transmission wait time are passed as arguments.

- **SoapClient constructor**

  - Line numbers 49 and 50

    Creates an end point address from the arguments.

  - Line number 52

    Creates a URL object.

- **open method**

  - Line number 61

    Creates a connection destination "HttpURLConnection" from the URL object.

  - Line number 63

    Specifies POST as the HTTP method to send SOAP messages using the POST method.

  - Line number 64

    Specifies "content-type" as "text/xml". "content-type" must be "text/xml". (SOAP 1.1 specification)

    Sets the event data character set.

  - Line number 65

    Sets a flag to be output.

  - Line number 66

    Connects to the data recipient.

- **sendMessage method**

  - Line numbers 79 to 112

    Loops the number of *loop* times specified in the argument.

  - Line number 80

    Executes the open method, and connects to the data recipient.

  - Line number 81

    Retrieves BufferedOutputStream.

  - Line number 83

    Replaces %COUNTER% contained in the event data with the loop count.

    This sends different event data every time.

    This example assumes that the following event data is used:

```
"STR0001","CPN0001",%COUNTER%,"30"
```

  - Line number 84

    Creates a SOAP message.

- Line number 94

  Writes the SOAP message using the specified character set.

- Line number 96

  Flushes BufferedOutputStream.

- Line number 97

  Closes BufferedOutputStream.

- Line number 99 and 100

  Retrieves InputStream from HttpURLConnection, and creates BufferedReader.

- Line number 104

  Writes sent results from BufferedReader to the standard output.

- Line number 107 and 108

  Closes InputStream and BufferedReader.

- Line number 110

  Disconnects HttpURLConnection.

- Line number 111

  Waits until the next event data is sent.

3.5.1.1 Example of Sample Execution (Sends a CSV data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Command execution result

```
# java -cp ./ SoapClient localhost CepEngine CSV UTF-8 CSVEvent SOAP,CSV,%COUNTER% 1 3 <ENTER>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
```

Note that in the example above, newlines have been added (lines 2 to 4, 6 to 8, and 10 to 12) for readability only. The actual output does not have newlines.

Engine log output result

```
2012-07-29 13:01:20,693 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :0: String
        ID      :SOAP: String

2012-07-29 13:01:20,730 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :1: String
        ID      :SOAP: String
```

```
2012-07-29 13:01:20,768 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :2: String
        ID      :SOAP: String
```

3.5.1.2 Example of Sample Execution (Sends an XML data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Note

In case XML data is sent through SOAP adapter, enclose the data with CDATA as described in the command execution result below.

Command execution result

```
# java -cp ./ SoapClient localhost CepEngine XML UTF-8 XMLEvent '<![CDATA[<?xml version="1.0"\
 encoding="UTF-8"?><XMLEvent><ID>SOAP</ID> <operation>XML</operation>\
 <count>%COUNTER%</count></XMLEvent>]]>' 1 3 <ENTER>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
<?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/env
elope/"><S:Body><ns2:notifyResponse xmlns:ns2="http://adapter.front.cep.cspf.fujitsu.com/"><re
turn>Code=0 Message=Sending message completed normally.</return></ns2:notifyResponse></S:Body>
</S:Envelope>
```

Note that in the command line example above, backslash ("\") and newlines have been added for readability only. The actual command line does not have backslash or newlines.

Note that in the output example above, newlines have been added (lines 4 to 6, 8 to 10, and 12 to 14) for readability only. The actual output does not have newlines.

Engine log output result

```
2012-07-29 13:02:21,860 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :0: String
        ID      :SOAP: String

2012-07-29 13:02:21,900 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :1: String
        ID      :SOAP: String

2012-07-29 13:02:21,935 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :2: String
        ID      :SOAP: String
```

3.5.2 HTTP Adapter

Sample program source code:

```
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class HttpClient {
    URL httpAdapterUrl = null;
    String url = "http://%HOSTNAME%/%ENGINE%FrontServerService/HttpReceiver";
    HttpURLConnection con = null;

    public static void main(String[] args)  {
        String hostName = "";
        String engineName = "";
        String dataType = "";
        String charSet = "";
        String eventTypeId = "";
        String data = "";
        long   lWait = 10;

        try {
            int loop = 0;
            if (args.length != 8) {
                System.out.println("param is Abnormal");
                return;
            }

            hostName    = String.valueOf(args[0]);
            engineName  = String.valueOf(args[1]);
            dataType    = String.valueOf(args[2]);
            charSet     = String.valueOf(args[3]);
            eventTypeId = String.valueOf(args[4]);
            data        = String.valueOf(args[5]);
            lWait       = Long.valueOf(args[6]);
            loop        = Integer.valueOf(args[7]);

            HttpClient hc = new HttpClient(hostName, engineName);
            hc.sendMessage(data, charSet, dataType, eventTypeId, loop, lWait);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public HttpClient(String hostName, String engineName) throws IOException {

        try {
            String wkUrl = url.replaceAll("%ENGINE%", engineName);
            wkUrl = wkUrl.replaceAll("%HOSTNAME%", hostName);

            httpAdapterUrl = new URL(wkUrl);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    private HttpURLConnection open(String charSet, String dataType, String eventTypeId)
            throws Exception {
```

```

        HttpURLConnection con = null;
        con = (HttpURLConnection) httpAdapterUrl.openConnection();

        con.setRequestMethod("POST");
        con.setRequestProperty("TYPE", dataType);
        con.setRequestProperty("EVENT-TYPE-ID", eventTypeId);

        if ("csv".equalsIgnoreCase(dataType)) {
            con.setRequestProperty("content-type", "text/plain; charset=" + charSet);
        } else if ("xml".equalsIgnoreCase(dataType)) {
            con.setRequestProperty("content-type", "text/xml; charset=" + charSet);
        } else {
            System.out.println("datatype is Abnormal");
            throw new Exception();
        }

        con.setDoOutput(true);
        con.connect();

        return con;
    }

    private void sendMessage(String data, String charSet, String dataType,
        String eventTypeId,int loop, long lWait) throws Exception {

        BufferedOutputStream bos = null;
        InputStreamReader ir1 = null;
        BufferedReader br1 = null;

        try {
            for (int i = 0; i < loop; i++) {
                con = this.open(charSet, dataType, eventTypeId);
                bos = new BufferedOutputStream(con.getOutputStream());

                String MSG = data.replaceAll("%COUNTER%", String.valueOf(i));
                bos.write(MSG.getBytes(charSet));

                bos.flush();
                bos.close();

                ir1 = new InputStreamReader(con.getInputStream());
                br1 = new BufferedReader(ir1);

                String line;
                while ((line = br1.readLine()) != null) {
                    System.out.println(line);
                }

                br1.close();
                ir1.close();

                con.disconnect();
                Thread.sleep(lWait);
            }

        } catch (MalformedURLException e) {
            e.printStackTrace();
            throw e;
        } catch (IOException e) {
            e.printStackTrace();
            throw e;
        } catch (InterruptedException e) {
```

```
123              e.printStackTrace();
124              throw e;
125          } finally {
126              try {
127                  if (br1 != null) {
128                      br1.close();
129                  }
130                  if (ir1 != null) {
131                      ir1.close();
132                  }
133              } catch (IOException e) {
134                  e.printStackTrace();
135                  return;
136              }
137          }
138
139          return;
140      }
141  }
```

- Detailed explanation of source code:

- **main method**

  - Line numbers 30 to 37

    The following data is obtained from the arguments at runtime:

Argument	Variable	Use
1	*hostName*	CEP Server host name
2	*engineName*	CEP engine name
3	*dataType*	Event data format
4	*charSet*	Character set
5	*eventTypeId*	Event type ID
6	*data*	Event data `%COUNTER%` included in event data needs to be substituted with loop count so that different event data shall be sent at each loop.
7	*lWait*	Data transmission wait time
8	*loop*	Data transmission count

  - Line number 39

    Creates an HttpClient object.

  - Line number 40

    Calls a method for sending event data.

    Event data, character set, data transmission count, and data transmission wait time are passed as arguments.

- **HttpClient constructor**

  - Line numbers 49 and 50

    Creates an end point address from the arguments.

  - Line number 52

    Creates a URL object.

- **open method**

  - Line number 62

    Creates a connection destination "HttpURLConnection" from the URL object.

  - Line number 64

    Specifies POST as the HTTP method to send SOAP messages using the POST method.

  - Line number 65

    Sets "TYPE" in request header.

  - Line number 66

    Sets "EVENT-TYPE-ID" in request header.

  - Line numbers 68 to 75

    Sets "content-type". Sets "text/plain" if the event data format is CSV and "text/xml" if it is XML. The event data character set should also be set.

  - Line number 77

    Sets a flag to be output.

  - Line number 78

    Connects to the data recipient.

- **sendMessage method**

  - Line numbers 91 to 114

    Loops the number of *loop* times specified in the argument.

  - Line number 92

    Executes the open method, and connects to the data recipient.

  - Line number 93

    Retrieves BufferedOutputStream.

  - Line number 95

    Replaces %COUNTER% contained in the event data with the loop count.

    This sends different event data every time.

    This example assumes that the following event data is used:

    ```
    "STR0001","CPN0001",%COUNTER%,"30"
    ```

  - Line number 96

    Writes the data using the specified character set.

  - Line number 98

    Flushes BufferedOutputStream.

  - Line number 99

    Closes BufferedOutputStream.

  - Line numbers 101 and 102

    Retrieves InputStream from HttpURLConnection, and creates BufferedReader.

  - Line number 106

    Writes sent results from BufferedReader to the standard output.

- Line numbers 109 and 110

  Closes InputStream and BufferedReader.

- Line number 112

  Disconnects HttpURLConnection.

- Line number 113

  Waits until the next event data is sent.

3.5.2.1 Example of Sample Execution (Sends a CSV data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Command execution result

```
# java -cp ./ HttpClient localhost CepEngine CSV UTF-8 CSVEvent HTTP,CSV,%COUNTER% 1 3 <ENTER>
Code=0 Message=Sending message completed normally.
Code=0 Message=Sending message completed normally.
Code=0 Message=Sending message completed normally.
```

Engine log output result

```
2012-07-29 13:05:02,954 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :0: String
        ID      :HTTP: String

2012-07-29 13:05:03,027 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :1: String
        ID      :HTTP: String

2012-07-29 13:05:03,108 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :2: String
        ID      :HTTP: String
```

3.5.2.2 Example of Sample Execution(Sends an XML data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Command execution result

```
# java -cp ./ HttpClient localhost CepEngine XML UTF-8 XMLEvent '<?xml version="1.0"\
 encoding="UTF-8"?><XMLEvent><ID>HTTP</ID> <operation>XML</operation>\
 <count>%COUNTER%</count></XMLEvent>' 1 3 <ENTER>
Code=0 Message=Sending message completed normally.
Code=0 Message=Sending message completed normally.
Code=0 Message=Sending message completed normally.
```

Note that in the command line example above, backslash ("\") and newlines have been added for readability only. The actual command line does not have backslash or newlines.

Engine log output result

```
2012-07-29 13:07:32,670 [DEBUG] abc--0:length=1
abc--0[0]
```

```
        operation       :XML: String
        count   :0: String
        ID      :HTTP: String

2012-07-29 13:07:32,685 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :1: String
        ID      :HTTP: String

2012-07-29 13:07:32,698 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :2: String
        ID      :HTTP: String
```

3.5.3 Socket Adapter

Sample program source code:

```java
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;

public class SocketClient {
    Socket s = new Socket();

    public static void main(String[] args) {
        String hostName = "";
        int    port = 0;
        String dataType = "";
        String charSet = "";
        String eventTypeId = "";
        String data = "";
        long   lWait = 10;
        int    loop = 0;
        int    dataCount = 1;

        if (args.length != 9) {
            System.out.println("param is Abnormal");
            return;
        }

        try {
            hostName    = String.valueOf(args[0]);
            port        = Integer.valueOf(args[1]);
            dataType    = String.valueOf(args[2]);
            charSet     = String.valueOf(args[3]);
            eventTypeId = String.valueOf(args[4]);
            data        = String.valueOf(args[5]);
            lWait       = Long.valueOf(args[6]);
            loop        = Integer.valueOf(args[7]);
            dataCount   = Integer.valueOf(args[8]);

            SocketClient c = new SocketClient(hostName, port);
            c.sendMessage(dataCount, dataType, charSet, eventTypeId, data, loop, lWait);

            System.out.println(c.readResponse());

        } catch (InterruptedException e) {
            e.printStackTrace();
            return;
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
    }

    public void sendMessage(int dataCount, String dataType, String charSet,
        String eventTypeId,String data, int loop, long wait)
        throws IOException, InterruptedException {

        DataOutputStream dos = new DataOutputStream(
                new BufferedOutputStream(s.getOutputStream()));

        for (int k = 0; k < loop; k++) {

            for (int i = 0; i < dataCount; i++) {
                int count = ( k * dataCount ) + i;
                String msg = data.replaceAll("%COUNTER%", String.valueOf(count));

                int length = dataType.getBytes().length + 4 + eventTypeId.getBytes().length
                 + 4 + charSet.getBytes().length + 4 + msg.getBytes().length;
                dos.writeInt(length);

                dos.write(dataType.getBytes());

                dos.writeInt(eventTypeId.getBytes().length);
                dos.write(eventTypeId.getBytes());

                dos.writeInt(charSet.getBytes().length);
                dos.write(charSet.getBytes());

                dos.writeInt(msg.getBytes().length);
                dos.write(msg.getBytes(charSet));

                dos.flush();
            }

            Thread.sleep(wait);
        }
        dos.writeInt(0);
        dos.flush();
    }

    public SocketClient(String host, int port) throws IOException {
        InetSocketAddress address = new InetSocketAddress(InetAddress.getByName(host),
            Integer.valueOf(port));
        s.setSendBufferSize(1000000000);
        s.connect(address);
    }

    public String readResponse() throws IOException {
        BufferedReader br = new BufferedReader( new InputStreamReader(s.getInputStream()));

        String ret =br.readLine();
        System.out.println("RESPONSE:" + ret);
        return "";
    }
}
```

- Detailed explanation of source code:
- **SocketClient class**
    - Line number 11

      Creates a Socket object.
- **main method**
    - Line numbers 30 to 38

      The following data is obtained from the arguments at runtime:

Argument	Variable	Use
1	*hostName*	CEP Server host name
2	*port*	Socket adapter port
3	*dataType*	Event data format
4	*charSet*	Character set
5	*eventTypeId*	Event type ID
6	*data*	Event data `%COUNTER%` included in event data needs to be substituted with loop count so that different event data shall be sent at each loop.
7	*lWait*	Data transmission wait time
8	*loop*	Data transmission count
9	*dataCount*	Number of event data to send at a time

- Line number 40

  Creates a SocketClient object.
- Line number 41

  Calls a method for sending event data.

  The number of event data to send at a time, event data format, character set, event type ID, event data, data transmission count, and data transmission wait time are passed as arguments.
- Line number 43

  Outputs a response from the CEP Server.

- **SocketClient constructor**
    - Line number 91

      Creates InetSocketAddress from the arguments.
    - Line number 93

      Sets the underlying size to be set in a network input/output buffer to be used.
    - Line 94

      Connects the socket to the CEP Server.
- **sendMessage method**
    - Line number 57

      Creates DataOutPutStream from the socket.
    - Line numbers 60 to 85

      Loop the number of *loop* times specified in the argument.

- Line numbers 62 to 82

  Loop of the number of event data to send at a time specified in the argument.

- Line number 64

  Replaces %COUNTER% contained in the event data with the loop count.

  This sends different event data every time.

  This example assumes that the following event data is used:

  ```
  "STR0001","CPN0001",%COUNTER%,"30"
  ```

- Line number 66

  Requests the total size of event data (unique format).

- Line number 68

  Writes the total size of event data (unique format).

- Line number 70

  Writes the byte array of event data format.

- Line number 72

  Writes the size of event type ID.

- Line number 73

  Writes the byte array of event type ID.

- Line number 75

  Writes the size of character set.

- Line number 76

  Writes the byte array of character set.

- Line number 78

  Writes the size of event data.

- Line number 79

  Writes the byte array of event data.

- Line number 81

  Flushes DataOutputStream.

- Line number 84

  Waits until the next event data is sent.

- Line number 92

  Receives a send complete notification (0), and notifies the CEP Server that the event has been sent.

- Line number 93

  Flushes DataOutputStream.

- **readResponse method**

  - Line number 98

    Creates BufferedReader from the socket.

  - Line number 100

    Reads a response message (one line).

- Line number 101

  Outputs the contents of the response message to the standard output.

3.5.3.1 Example of Sample Execution (Sends a CSV data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Command execution result

```
# java -cp ./ SocketClient localhost 8001 CSV UTF-8 CSVEvent SOCKET,CSV,%COUNTER% 1 2 2 <ENTER>
RESPONSE:0000:6:Sending message completed normally.
```

Engine log output result

```
2012-07-29 13:27:49,410 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :0: String
        ID      :SOCKET: String

2012-07-29 13:27:49,422 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :1: String
        ID      :SOCKET: String

2012-07-29 13:27:49,427 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :2: String
        ID      :SOCKET: String

2012-07-29 13:27:49,428 [DEBUG] abc:length=1
abc[0]
        operation       :CSV: String
        count   :3: String
        ID      :SOCKET: String
```

3.5.3.2 Example of Sample Execution(Sends an XML data)

An example of sample execution is shown below.

In this example, debug information is output to the engine log by using DebugLogListener.

Command execution result

```
# java -cp ./ SocketClient localhost 8001 XML UTF-8 XMLEvent '<?xml version="1.0"\
 encoding="UTF-8"?><XMLEvent><ID>SOCKET</ID> <operation>XML</operation>\
 <count>%COUNTER%</count></XMLEvent>' 1 2 2 <ENTER>
RESPONSE:0000:4:Sending message completed normally.
```

Note that in the command line example above, backslash ("\") and newlines have been added for readability only. The actual command line does not have backslash or newlines.

Engine log output result

```
2012-07-29 13:30:56,861 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :0: String
        ID      :SOCKET: String
```

```
2012-07-29 13:30:56,862 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :1: String
        ID      :SOCKET: String

2012-07-29 13:30:56,865 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :2: String
        ID      :SOCKET: String

2012-07-29 13:30:56,865 [DEBUG] abc--0:length=1
abc--0[0]
        operation       :XML: String
        count   :3: String
        ID      :SOCKET: String
```

Chapter 4 Custom Listener Reference

The custom listener passes the results of complex event processing to a user-developed Java class. This chapter provides an overview of user-developed Java classes and contains information required for their development.

4.1 Overview of User-Developed Java Classes

A user-developed Java class must implement the following Java interface:

```
com.fujitsu.cspf.cep.CustomListener
```

The following two methods must be implemented using this interface:

Type of return value	Method name	Argument	Explanation
void	setArgs	String[] args	The args parameter specified using the `@CustomListener` annotation is passed.
		String statementName	The name (specified using `@Name`) of the complex event processing statement to which the `@CustomListener` annotation was attached is passed.
void	update	Map[] newEvents	The processing result (output event) of the complex event processing statement to which the `@CustomListener` annotation was attached is passed as an array of the java.util.Map object. The output event passed as a java.util.Map object contains a property name and value pair. If the rules output events periodically, the update method may be called even if there is no output event.

The following processes are performed for the user-developed Java class each time there is an output from a complex event processing statement:

1. An instance of the user-developed Java class is generated.
2. The setArgs method is called.
3. The update method is called.

In addition, an instance of a user-developed Java class is generated when the CEP engine is started.

If an exception occurs during custom listener processing, the CEP engine catches the exception and outputs it to the engine log and the system log. Processing of other events continues.

Note

A user-developed Java class runs on the same Java VM as the CEP engine. Take the following points into account when designing a user-developed Java class:

- Design a user-developed Java class so there is no bottleneck in processing time

  Design a user-developed Java class so that it takes only a short time to generate an instance of the user-developed Java class and to call the setArgs and update methods. If these processes take time, events awaiting processing may accumulate in the CEP engine and adversely affect the processing performance of the entire CEP engine. Particularly if a large volume of events is to be output, the impact will be greater.

- Create a thread-safe design

  Processing of a user-developed Java class is called from multiple threads, so create a thread-safe design. For example, if using class variables, you must consider the fact that the user-developed Java class will be called from multiple threads.

However, an instance of a user-developed Java class is generated each time an output event occurs, and one instance runs on only one thread. Therefore, if processing involves merely operating the instance variables, there is no need to consider multiple threads.

- Throw errors that must be monitored as exceptions

  To monitor errors that occur in a user-developed Java class, throw details of the error as an exception outside the user-developed Java class. An exception that is thrown is caught by the CEP engine and output to the engine log and the system log.

Example

Sample source code of a user-developed Java class is stored in the following directory:

```
/opt/FJSVcep/sample/CustomListener
```

4.2 Developing a User-Developed Java Class

This section explains how to implement and deploy a user-developed Java class to be called via the custom listener of the CEP engine.

4.2.1 CustomListener Interface

The CustomListener interface is contained in /opt/FJSVcep/cep/lib/CepServerCustom.jar on the CEP Server. Copy CepServerCustom.jar to the Java development environment you are using, set the class path, and develop the Java class.

4.2.2 Custom Log

Logs can be output from a user-developed Java class to a log file (custom log) for the custom listener. The output destination of the custom log is as follows:

```
/var/opt/FJSVcep/cep/cep/logs/EngineLog/cepEngineName/custom.log
```

Output

Use the Apache Log4j class (org.apache.log4j.Logger) for implementation.

Obtain a log output instance using the following method (The argument must be "custom"):

```
Logger myLogger = Logger.getLogger("custom");
```

Information

The Apache Log4j jar file is located in /opt/FJSVcep/log4j/lib/log4j-1.2.16.jar.

Use the fatal, error, warn, and info methods of the Logger class to output logs.

```
myLogger.error("xxxxxx");
```

Note

You cannot use the trace or debug methods of the org.apache.log4j.Logger class.

4.2.3 Compilation

Compile the source code you created, and generate a class file or jar file.

If using javac of JDK7 or later, specify the "-target 1.6" option when compiling the source code.

4.2.4 Deployment

Store the created class file in the following directory:

```
/etc/opt/FJSVcep/config/custom/engineName/classes
```

Store the created jar file in the following directory:

```
/etc/opt/FJSVcep/config/custom/engineName
```

Note

- Grant access permissions so that the engine execution user can read the created class file and jar file.
- For storing a class file, create a directory corresponding to the class package name in the classes directory. Grant access permissions also for the created directory so that the engine execution user can read it.
- The class file and the jar file stored during the CEP engine is running are not enabled until the CEP engine is restarted.

Example

Example of storing a class file

Create a directory named com/example as shown below for storing the com.example.Example class.

```
# cd /etc/opt/FJSVcep/config/custom/engineName/classes <ENTER>
# mkdir -p com/example <ENTER>
# chmod 555 com <ENTER>
# chmod 555 com/example <ENTER>
# cp pathOfClassFileToBeStored com/example/ <ENTER>
# chmod 444 com/example/classFileName <ENTER>
```