# HP OpenView Event Correlation Services Designer's Reference 

HP-UX, Solaris, Windows NT ${ }^{\circledR}$, Windows ${ }^{\circledR} 2000$ andWindows ${ }^{\circledR}$ XP


Manufacturing Part Number: J 1095-90314
J anuary 2003
© Copyright 2001 Hewlett-Packard Company.

## Legal Notices

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, theimplied warranties of merchantability and fitness for a particular purpose Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another Ianguage without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.22719 for other agencies.

## HEWLETT-PACKARD COMPANY

3404 E. Harmony Road
Fort Collins, CO 80528 U.S.A.
Use of this manual and flexible disk(s), tape cartridge(s), or CD-ROM (s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright Notices. ©Copyright 1983-2001 Hewlett-Packard Company, all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

Contains software from AirMedia, Inc.
© Copyright 1996 AirMedia, Inc.

## Trademark Notices

J ava ${ }^{T M}$ is a U.S. trademark of Sun Microsystems, Inc.
Microsoft $\circledR^{\circledR}$ is a U.S. registered trademark of Microsoft Corporation.
Windows NT $\circledR^{\circledR}$ is a U.S. registered trademark of Microsoft Corporation.
Windows ${ }^{\circledR} 2000$ is a U.S. registered trademark of Microsoft Corporation.
Windows $\circledR^{\circledR}$ and MS Windows ${ }^{\circledR}$ are U.S. registered trademarks of Microsoft Corporation.

Netscape ${ }^{T M}$ and Netscape Navigator ${ }^{T M}$ are U.S. trademarks of Netscape Communications Corporation.

Oracle® is a registered U.S. trademark of Oracle Corporation, Redwood City, California.

Oracle7 ${ }^{\text {TM }}$ is a trademark of Oracle Corporation, Redwood City, California.

OSF/M otif $\circledR^{8}$ and Open Software Foundation $®$ are trademarks of Open Software Foundation in the U.S. and other countries.

Pentium ${ }^{\circledR}$ is a U.S. registered trademark of Intel Corporation.
UNIX ® is a registered trademark of The Open Group.
Perl is a trademark of O'Reilly \& Associates, Inc.
All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

## Contents

1. Introduction
Purpose. ..... 20
Audience ..... 21
2. Circuits and Compound Nodes ECS Correlation Circuits ..... 25
Circuit Features .....  26
Streams ..... 27
Stream Policy. ..... 27
Event Flow Within a Circuit ..... 29
Circuit Design and Stream Policy ..... 30
Circuit Sharing Between Streams ..... 30
Multiple Circuits ..... 32
The ECS Designer ..... 33
Compound Nodes ..... 34
Properties of Circuits and Compound Nodes ..... 34
External Event Filtering ..... 35
Global Definitions. ..... 37
Attributes: Passing Information out of a Compound Node ..... 40
Parameters: Passing Information into a Compound Node. ..... 41
Circuit Policy ..... 42
3. Circuit Nodes
Common Features of Nodes ..... 46
Node Configuration ..... 46
Name and Status ..... 46
Node Parameters ..... 49
Input and Output Ports of Nodes ..... 53
Node Attributes ..... 55
Transit Delays and Simulation Statistics. ..... 56
Annotate Node ..... 58
Annotate Node Processing ..... 59

## Contents

Annotate Node Parameters ..... 61
Annotate Node Ports ..... 62
Annotate Node Notes ..... 63
Clock Node ..... 64
Clock Node Processing ..... 65
Clock Node Parameters ..... 66
Clock Node Ports ..... 70
Clock Node Notes ..... 70
Combine Node. ..... 71
Combine Node Description ..... 73
Combine Node Parameters ..... 75
Combine Node Ports ..... 78
Combine Node Notes ..... 78
Compound Node ..... 80
Count Node ..... 81
Count Node Processing ..... 82
Count Node Parameters ..... 84
Count Node Attributes ..... 84
Count Node Ports ..... 85
Count Node Notes ..... 85
Create Node ..... 87
Create Node Processing ..... 87
Create Node Parameters ..... 89
Create Node Ports ..... 91
Create Node Notes ..... 91
Delay Node ..... 93
Delay Node Processing ..... 93
Delay Node Parameters ..... 95
Delay Node Ports ..... 96
Delay Node Notes ..... 96
Extract Node. ..... 97
Extract Node Processing. ..... 97

## Contents

Extract Node Parameters ..... 100
Extract Node Ports ..... 101
Extract Node Notes ..... 101
Filter Node ..... 102
Filter Node Processing ..... 102
Filter Node Parameters ..... 103
Filter Node Ports ..... 104
Modify Node ..... 105
Modify Node Processing ..... 106
Modify Node Parameters ..... 107
Modify Node Ports ..... 107
Modify Node Notes ..... 107
Rate Node. ..... 109
Rate Node Processing. ..... 109
Rate Node Parameters ..... 111
Rate Node Attributes ..... 114
Rate Node Ports ..... 115
Rate Node Notes ..... 115
Rearrange Node ..... 116
Rearrange Processing. ..... 116
Rearrange Node Parameters ..... 118
Rearrange Node Ports ..... 120
Rearrange Node Notes ..... 121
Sink Node ..... 122
Sink Node Processing ..... 122
Sink Node Ports ..... 124
Sink Node Notes ..... 124
Source Node ..... 125
Source Node Processing ..... 125
Source Node Ports ..... 127
Table Node ..... 128
Table Node Processing ..... 129

## Contents

Table Node Parameters. ..... 133
Table Node Ports ..... 135
Table Node Attributes ..... 135
Table Node Examples ..... 136
Table Node Notes ..... 137
Unless Node ..... 139
Unless Node Processing ..... 141
Unless Node Parameters ..... 143
Unless Node Ports ..... 144
Unless Node Examples ..... 145
Unless Node Notes ..... 145
4. Events in ECS
Introduction ..... 149
Event Flow ..... 150
Manipulating Events ..... 150
Event Header Attributes ..... 152
Primitive Events. ..... 154
Structure of Primitive Events ..... 154
Working with Event Attributes ..... 155
Temporary Events ..... 157
Structure of Temporary Events ..... 157
Composite Events ..... 159
Working with Attributes and Components ..... 159
5. Timing Considerations
Time Synchronization ..... 167
Times and Durations ..... 168
Current Time ..... 169
Event Time Attributes ..... 170
Creation Time ..... 170
Arrival Time ..... 170

## Contents

Transit Delays ..... 171
Simulating Time in the ECS Designer ..... 173
Engine Input Event Logs ..... 176
Order of Processing ..... 177
Engine Time Management ..... 179
Input Phase. ..... 179
Cleanup Phase ..... 179
Input Phase Overload Processing ..... 180
6. Data Store and Fact Store
Data Store and Fact Store Contexts ..... 183
Overview ..... 183
Contexts and Multiple Circuits ..... 183
Event Creation ..... 184
Store Versions and Transit Delays ..... 185
Data Store ..... 186
File Formats ..... 187
Data Types ..... 188
Using the Current Context to Access the Data Store ..... 190
Using Event Contexts to Access the Data Store ..... 191
Fact Store ..... 193
File Formats ..... 194
Data Types ..... 195
Using the Current Context to Access the Fact Store ..... 197
Using E vent Contexts to Access the Fact Store ..... 199
Multiple Data and Fact Stores ..... 200
Simulating Event Contexts in the ECS Designer ..... 201
Overview ..... 201
ADD Commands ..... 201
DELETE Commands ..... 202
Dumping Stores ..... 202

## Contents

7. Identifiers, Comments, and Reserved Words
Identifiers ..... 205
Comments in ECDL ..... 206
Reserved Words and Symbols ..... 207
8. Data Types
Integer Data Type ..... 213
Real Data Type ..... 214
Boolean Data Type ..... 215
Duration Data Type ..... 216
Time Data Type ..... 217
Oid Data Type ..... 218
String Data Type ..... 219
Token Data Type ..... 221
Tuple Data Type. ..... 222
List Data Type ..... 223
Dictionary Data Type. ..... 225
Event Data Type. ..... 226
Void Data Type ..... 227
Function Data Type ..... 228
Opaque Data Type ..... 229
Dynamic and User Defined Types ..... 230
Dynamic types ..... 230
Union types ..... 231
User-Defined types ..... 231
9. Operators and Built-in Functions
Operators ..... 235

+ (add, string concatenate, positive) ..... 237
- (subtract, negative) ..... 238
* (multiply) ..... 240
/ (real divide) ..... 241


## Contents

^(exponentiate) ..... 242
= and !=(equality operators) ..... 243
$\gg=\ll=$ (relational operators). ..... 244
:: (list constructor) ..... 246
Built-in Functions ..... 247
alter ..... 252
and. ..... 253
append ..... 254
bitand ..... 255
bitinv ..... 256
bitleft ..... 257
bitor ..... 258
bitright ..... 259
bitxor ..... 260
chr ..... 261
datastore ..... 262
div ..... 263
explode ..... 264
explode time ..... 265
fact_exists ..... 267
fact_find_left ..... 268
fact_find_right ..... 269
feed ..... 270
feedall ..... 271
feedothers ..... 272
flush. ..... 273
fold ..... 274
foldr ..... 275
gen_ccall ..... 276
gen_perlcall ..... 276
implies ..... 276
implode_time ..... 278

## Contents

integerOf ..... 280
inter ..... 280
interc ..... 281
I p.fromOctet ..... 282
Ip.toOctet ..... 283
join ..... 284
length ..... 285
Match.make ..... 286
Pattern-matching ..... 287
Defining Match Expressions ..... 288
Tags. ..... 290
Assignment Rules ..... 290
Sub-Patterns Assignment ..... 291
Examples of Pattern-matching Conditions ..... 292
Match.test ..... 293
Match.testVars ..... 294
mod. ..... 296
not ..... 296
nth ..... 297
oid.append ..... 298
oid.join ..... 299
oid.last ..... 300
oid.split ..... 301
oid.split_at ..... 302
oid.split_nnm. ..... 303
or ..... 304
ord ..... 304
perl_interp ..... 305
reverse ..... 306
round ..... 306
split ..... 307
stringOf ..... 308

## Contents

System.audit_log ..... 309
System.circuit_dump ..... 310
System.trace ..... 311
System.set_trace_mask ..... 312
Timenow ..... 314
truncate. ..... 315
union ..... 316
unionc ..... 317
10. Writing ECDL Expressions
Binding and Comparisons ..... 321
Named Values ..... 321
Binding Patterns ..... 321
"Let" Expressions in Node Parameters ..... 324
Comments in ECDL ..... 325
Data Type Handling ..... 326
Tuple Manipulation ..... 326
List Manipulation ..... 327
Searching Lists ..... 328
Deriving Lists from Lists ..... 330
Dictionary and Event Manipulation ..... 332
Flow-control Expressions ..... 334
Making Decisions: If and Choose Expressions ..... 334
Case. ..... 336
Typecase ..... 337
Sequence ..... 339
Exceptions ..... 341
Handling Exceptions ..... 341
Standard Exceptions ..... 342
Raising Exceptions ..... 343
Functions and Language Layout ..... 344
Calling Function Syntax ..... 344

## Contents

Prefix and Infix Functions ..... 345
User Defined Functions ..... 345
Advanced Function Writing Features ..... 347
Fixity Declarations, Associativity and Precedence ..... 351
Modules ..... 355
Module Files ..... 356
M odules and Name Search ..... 356
Invoking Perl Functions from ECDL ..... 359
Perl to ECDL Mapping ..... 360
Perl to ECDL Data Type Mapping ..... 362
Invoking 'C' F unctions ..... 363
Writing functions in C ..... 363
An Introduction to ECDL for C Programmers ..... 370
11. Event Log File Format
Log File Syntax. ..... 379
The Event Syntax Line. ..... 380
Logs Generated by the ECS Engine. ..... 381
Annotation Logs ..... 382
The Annotation Event Format ..... 383
Special Characters ..... 385
12. Audit Logging
Verifying the Operation of a Circuit ..... 389
The audit_log Function ..... 390
Using Unique IDs to Lookup Events Details ..... 391
Designing an Audit ..... 392
Static and Dynamic Evaluation ..... 392
13. Files and Directories
Files Used or Generated by the ECS Designer ..... 397

## Contents

A. Event Correlation in OVO
OVO Message Attributes. ..... 403
Using Event Attributes ..... 408
Logging E vents in OVO ..... 409
Event Log Format ..... 409
Troubleshooting OVO-specific Problems ..... 412
Glossary

Contents

## Contact Information

Contacts Please visit our HP OpenView web site at:http://openview.hp.com/There you will find contact information as well as details about theproducts and services HP OpenView has to offer.
Support The "hp OpenView support" area of the HP OpenView web site includes:

- Downloadable documentation
- Troubleshooting information
- Patches and updates
- Problem reporting
- Training Information
- Support program information


## 1 Introduction

## Purpose

This reference contains the information and concepts you need to use HP OpenView Event Correlation Services (ECS) Designer:

- the ECS circuit paradigm, based on event flow through interconnected correlation nodes
- the importance of time in event correlation
- the mechanism for parameterizing ECS circuits to make them data-driven
- the Event Correlation Description Language (ECDL), including its language constructs, functions, and data types


## Audience

The reader should be experienced in building network management applications on UNIX or Windows NT platforms, and be familiar with:

- network management principles
- the problems that typically occur within their managed network, and the events generated as a consequence
- programming in high-level languages, particularly functional languages (only advanced users of the ECS Designer)

Introduction
Audience

## 2 Circuits and Compound Nodes

An ECS circuit is built using the ECS Designer. See the HP OpenView Event Correlation Services Designer's Guidefor detailed information on how to use this tool. This chapter defines the basic concept of a circuit and describes how a circuit is connected to its external environment. It also describes how circuits are modularized and broken down into Compound nodes.

The information in this chapter provides a foundation for later chapters in this book. The first section:

- "ECS Correlation Circuits" on page 25
describes what a circuit is and modularization using Compound nodes. The second section discusses circuit properties and the differences between a top-level circuit and a Compound node:
- "Compound Nodes" on page 34

NOTE
In HP OpenView Operations an ECS circuit is equivalent to a template.

## ECS Correlation Circuits

Conceptually, event correlation in ECS is done by processing a stream of events through a correlation circuit. Like electrons in an electronic circuit, events flow through a correlation circuit from inputs to outputs. The components in a correlation circuit are called correlation nodes, (or usually just nodes) and events flow from one node to the next along connections.

Correlation nodes are standardized building-blocks, like integrated circuit chips, that are parameterized by supplying values and expressions, to perform specific operations on the events flowing through them. For example, some nodes are used to filter events, others to store events, and others to create or modify events. By connecting such nodes together and parameterizing them with ECDL expressions, an ECS circuit is constructed.

ECDL is the Event Correlation Description Language used to describe the values and expressions that constrain a general purpose node to a particular correlation task. Although ECDL is a full-featured language, most node parameters are just literal values or simple expressions. If the problem is complex, however, the full power of a functional programming language is available.

The ECS Designer is a graphical tool that makes it easy to build circuits and simulate their operation. The ECS Designer must be used to create or modify a circuit; there is no other way in which a circuit can be developed, edited, or compiled.
ECS circuits can implement a wide variety of correlation logic to:

- Reduce the number of events to be managed by suppressing:
- repetitious events,
- unwanted event types,
- paired events (such as power-down followed closely by power-up)
- Provide the root cause rather than the symptoms of a network management incident by:
- recognizing patterns of events
- Increase the information content of events by:
- consolidating information from a number of events
- adding simple data and facts to events
- annotating events with information derived from an external system such as a database
- Translate events from one format to another.

For example, generating the appropriate SNMP trap in response to a pattern of ASCII messages).

ECS can reduce network loads by eliminating unwanted messages, and lead to quicker problem identification and resolution by increasing the information content of remaining messages.

## Circuit Features

The circuits built with ECS Designer have a number of important features:

Input Port. A circuit must have at least one input port through which events can enter the circuit. Each input port can be configured to accept certain events only.
Output Port. A circuit must have one or more output ports from which events are output from the circuit.

Nodes. A circuit is comprised of at least one correlation node. Like circuits, nodes have one or more input and output ports through which events flow. Nodes provide pre-packaged logical operations that affect the stream of events flowing through them.

Connections. N odes are connected together by joining the output port of one node to the input port of another. An output port can be connected to many input ports, and an input port can be connected to many output ports.
Node Parameters. To configure a node you enter expressions in the node's parameters. Expressions can read and set event attributes, compare values, lookup information, and perform general purpose computations.

Global Definitions. Expressions in node parameters can make use of global definitions to simplify expressions, provide consistency, and improve readability.

Fact and Data Stores. Simple, relatively static information can be loaded into a Fact or Data Store and accessed from expressions in node parameters.

Annotation. Complex or dynamic information outside ECS can be accessed through the Annotate node.

Compound Nodes. Any circuit or partial circuit can be encapsulated as a Compound node. Once encapsulated, the Compound node can be used like any other node. The properties of circuits and Compound nodes are discussed in detail in "Compound Nodes" on page 34.

## Streams

Multiple independent event streams are supported within the E CS Engine, where each event that enters the engine simultaneously flows into every stream. E ach stream is correlated independently, which means that outputting or discarding an event in one event stream does not affect this event in any other event stream. Applications register for a specific event stream.

In a standalone ECS correlation engine environment and in NNM, applications specify their desired stream as a parameter when registering for event output.

The environment in which the engine is running can restrict the means by which events can be output. For more details, see the HP OpenView ECS Administrator's Guide.

## Stream Policy

Each stream has a configurable policy for handling events that are not accepted by any circuit. The stream policy determines how events are processed:

- A stream with an output policy outputs an event unless at least one circuit enabled on the stream discards the event.
- A stream with a discard policy outputs an event when at least one circuit enabled on the stream outputs it.

The default stream policy is "Output" which outputs events that are not accepted by any circuit.

These rules are summarized in Table 2-1.
Table 2-1 Summary of Stream Policy Rules
$\left.\begin{array}{|l|l|l|l|}\hline \text { Policy } & \begin{array}{l}\text { Allowed Circuit } \\ \text { Policies }\end{array} & \begin{array}{l}\text { An Event is } \\ \text { Discarded When... }\end{array} & \begin{array}{l}\text { An Event is } \\ \text { Output When... }\end{array} \\ \hline \text { Output } & \begin{array}{l}\text { - Output } \\ -\end{array} \text { Unspecified }\end{array} \quad \begin{array}{l}\text { At least one circuit in the stream } \\ \text { discards the event. }\end{array} ~ \begin{array}{l}\text { The event is not discarded by } \\ \text { any circuit in the stream. }\end{array}\right]$

For example, a circuit designed to create a special warning event when a security violation is detected would probably be given a Discard circuit policy, meaning that it could only be enabled on a stream with a Discard policy. When this stream is initially created it is opaque-that is, it will not output any events until at least one circuit is enabled. When the security violation circuit is enabled, the only events output from the stream are those explicitly output by the circuit, in this case when the circuit detects a security violation. Only then is the special warning event created and emitted.

Compare this with a circuit designed to suppress events during scheduled maintenance. In this case, the circuit would have a policy of Output and could only be enabled on an Output stream. When the output stream is enabled it is transparent-that is, it passes all events from its input to its output. When the scheduled maintenance circuit is enabled, the only events suppressed are those that the circuit recognizes are being generated by equipment under scheduled maintenance; all other events are passed through.

In practise, output stream policies are more common than discard stream policies. In NNM, for example, the default stream has an output policy, and this policy cannot be changed.

## Event FIow Within a Circuit

To understand how a circuit affects the correlation of an event, it is important to understand the difference between copying an event and creating a new reference to the event.

When a primitive event enters the ECS engine, there is just a single instance of the event which can never be modified; it is either output unchanged, or discarded completely. This ensures that an event referred to by a node will not change unexpectedly.
As the original primitive event cannot be modified, the engine uses references to the same event throughout the circuit. New references to events are created by each source node that accepts the event, and when the output from one node is connected via junctions into the inputs for several nodes.

References to events are discarded under the following circumstances:

- When an event exits a node from an unconnected port, the reference to the event is discarded.
- When an event enters a node that does not output the event, the reference to the event is (eventually) discarded. For instance, events entering the inhibitor port of an unless node are discarded when the Unless node has finished with the event. See Chapter 3, "Circuit Nodes," on page 43 for details on specific nodes.
It is normal for references to an event to be discarded many times by a circuit. However, an event is only considered to be discarded by a circuit when every reference to the event in the circuit has been discarded, and the event never reached an output port in the circuit.
To change an event, a copy of the event is made and the changes are made to that. This means that all existing references to the original event will be unaffected by the new changed event, and the circuit's decision whether to output the new changed event has no effect on the decision whether to output the original event.
New primitive events are created or modified under the following circumstances:
- The Extract node creates a new event by copying events from a Table node, and then modifies the creation/arrival time of the new event.
- The Modify node creates a copy of the input event and modifies that. The reference to the input event is then discarded.
- The Create node creates a new event. The reference to the input event is then discarded.

If several references to the same event reach output node(s), the second and subsequent occurrences are ignored. In other words, the original event will never be output more than once, even if it reaches more than one output node in one or more circuits.

## Circuit Design and Stream Policy

The stream policy affects the way that most circuits are designed. This is because a circuit may accept "extra" events that it does not wish to correlate. For instance, a Source node may accept a broader category of events than the circuit wants, or a circuit may use one event to assist in the correlation of another event without wanting to affect the output decisions for the first event.

The circuit designer does not want to affect the output decisions for these "extra" events. In other words, as the circuit becomes enabled or disabled, the engine's event flow for these extra events should be unchanged.

If the stream policy is "output", the circuit designer must output these extra events otherwise they would be discarded from the event flow. Similarly, if the stream policy is "discard", the circuit designer must discard these events otherwise they would be output from the event flow.

To summarize, for a circuit to preserve the original output decision for an event, the circuit should handle the event the same way that the stream policy would.

For this reason, circuit designers and the ECS administrator should agree on the appropriate stream policy for the ECS engine.

## Circuit Sharing Between Streams

A circuit can be enabled on more than one stream. There are two ways to do this:

- Load the circuit once and enable it multiple times so that the same instance of the circuit is used in multiple streams.
- Load the circuit using a different symbolic name in each stream so that there is a different instance of the circuit in each stream.
Both methods are valid approaches, depending on what you are trying to achieve.

Single Instances Shared Across Multiple Streams E ach stream can contain multiple circuits, and the same circuit can be enabled in multiple streams. If the same circuit is enabled in several streams, only one instance of the circuit is created, and outputting an event from the circuit is recognized by every stream to which the circuit belongs. In other words, the same instance of a circuit contributes to multiple streams. This can be far more efficient for the ECS engine than loading and enabling a new instance of the same circuit for each stream.
Events only enter the circuit once, regardless of how many streams the circuit is enabled in. The input of a circuit will not be disabled until it has been disabled in every stream, but the output of a circuit can be enabled and disabled on a per-stream basis. When a circuit's output is disabled in a specific stream, the events within the circuit will be flushed into that stream (subject to stream policy), but the circuit itself and the other streams will be unaffected.

There are a number of reasons why it is more efficient to have the same instance of a circuit in multiple streams rather than a different instance in each stream:

- Create, Modify, and Extract nodes produce new events, meaning that multiple instances of a circuit would produce different events from each stream.
- Annotation and topology requests must be done multiple times if different instances of a circuit are used in each stream.
- Performance suffers if the same things is done multiple times.
- Timing in general and multiple calls to topol ogy could potentially produce different results from the same circuit on different streams.

Separate Instances for E ach Stream The same circuit can be loaded multipletimes using different circuit identifiers. This means different instances of the same circuit can be loaded in different streams. This can be useful when the circuit is used with a different data store or fact store. For example, when testing a new circuit you might want to test the new circuit in a separate "test" stream using test instances of existing production circuits. This would ensure that the testing would not make any changes to production data stores or fact stores.

## Multiple Circuits

So far we have examined events within a single circuit, now let's examine what happens when the same event enters multiple circuits. In particular, consider what should happen when some circuits output the event and some circuits discard the event.

The stream's policy can be thought of as the default behavior for events before any circuits are loaded. Therefore, the purpose of circuits is to change this behavior for an event.
For instance, if the stream's policy is to output events, circuits will change the event flow by discarding events and by creating new events. Circuits will preserve the event flow by outputting events.

Similarly, if the stream's policy is to discard events, circuits will change the event flow by outputting events and by creating new events. Circuits will preserve the event flow by discarding events.

When several circuits make different decisions for the same event, the engine will uphold a decision to change the event flow over a decision to preserve the event flow. Therefore, if the same event enters multiple circuits and is output by some circuits and discarded by others, the event will actually be discarded under an output policy, and output under a discard policy.

Remember, with an output policy, the engine will output an event unless it is discarded by any circuit. With a discard policy, the engine will discard the event unless it is output by any circuit.
When a new event is created by a Create node, a M odify node or an Extract node, this new event will never be affected by another circuit. This is because circuits only see the original event flow entering the engine; they cannot see events created by another circuit.
When an event enters the ECS engine, it simultaneously enters every circuit that accepts it. With a discard policy, the event is output as soon as any circuit outputs it as the remaining circuits will not affect the output decision. However, with an output policy, the event cannot be output until every circuit that accepted it has output it. This means that under a discard policy an event has the transit delay of the circuit with the smallest transit delay, but under an output policy an event has the transit delay of the circuit with the largest transit delay.

## The ECS Designer

The ECS Designer allows you to load, build and simulate one circuit at a time, so it is important to understand what happens when multiple circuits operate on the same engine.

Although circuits are designed for either stream policy, the simulator within the ECS Designer always runs with a Discard policy. This means that events that are not accepted by any input port in the simulated circuit will not be seen in the output event logs.

This allows a circuit designer to focus on the events that are affected by the circuit.

## Compound Nodes

Basically, a Compound node is a self-contained block of ECS circuitry. It provides a means to encapsulate selected functionality and encourages:

- reuse of commonly used circuitry
- standardization of common correlation functions
- dividing a large problem into smaller problems

All circuits, whether they are enclosed in a Compound node or not, possess a set of properties that determine how that circuit interacts with its environment. Some of these properties apply when the circuit is used as a Compound node, others apply when it is used as a top-level circuit. The following sections describe all the circuit properties in detail.

## Properties of Circuits and Compound Nodes

Table 2-1 summarizes the properties of top-level circuits and Compound nodes. Note that a circuit always possesses all the properties, but that only the appropriate set of properties is active at any given time. So, some of these properties apply only when the circuit is used as a Compound node and others apply only when it is used as a top-level circuit connected to the outside world. The only common property is the event flow circuit itself.

These differences lead to some subtle behavior differences between a circuit in a compound node and a top-level circuit:

- When a circuit is treated as a Compound node its global definitions are active but not visible in the ECS Designer. Only global definitions in the top-level circuit are visible in the ECS Designer.
- Circuit input ports become node input ports when a circuit is packaged as a Compound node. Configuration details that are specific to a circuit's interface with the outside world are ignored when the circuit is used as a Compound node.
- If a Compound node is to be configurable, attributes and parameters must be exported from the circuit. Conversely, when the Compound node is used as a top-level circuit its exported attributes and parameters are ignored.


## Table 2-2 Differences Between Circuits and Compound Nodes

| Property | Top-level Circuit | Compound Node |
| :--- | :--- | :--- |
| Event Flow Circuit | YES | YES |
| External Input port <br> filtering | YES | Ignored |
| Global Definitions | YES | Uses globals from the <br> enclosing circuit. Globals <br> defined in the compound are <br> not visible. |
| Attributes | Ignored, not <br> appropriate | YES |
| Parameters | Ignored, not <br> appropriate | YES |
| Circuit Policy | YES | YES |

These properties are discussed in detail in the remainder of this chapter.

## External Event Filtering

When a top-level ECS circuit is connected to an event stream, it may be subjected to many events in which the circuit has no interest. The events in which a circuit is interested are defined by the external input port filters.

Events that do not pass into any of the circuits running on the engine are handled by the engine's policy.

An ECS circuit can have many input ports. Each input port can be configured to filter events based on the event's encoding type, event syntax, event type, the transit delay, and/or an ECDL Filter Condition expression. The conditions are specified in the ECS Designer on the External Tab. See the HP OpenView Event Correlation Services Designer's Guidefor details.

## Circuits and Compound Nodes

## Compound Nodes

## Input port event processing

Events can enter more than one input port, but only under the following conditions:

- the event satisfies the conditions for that port, and
- there is no other port with a more restrictive condition that this event satisfies.

In other words, an event enters the most restrictive port that it can, to the exclusion of less restrictive ports. If there are several equally restrictive ports then the event enters all the equally restrictive input ports.

An ECS circuit can have up to 50 input ports. Each event is tested against each input port's conditions, depending on the restrictions specified for each of the input ports. The restrictions that can be specified are the Encoding Type, Event Syntax, Event Type, Transit Delay range, and Filter Condition. Only when an event matches more than one port's input specifications at the same restriction level, will it enter more than one port.

When an event enters more than one input it is processed by both circuit paths. However, the event is not duplicated; there is still only one event and the ECS Engine will ultimately transmit only one instance of the event from the output, unless the event is suppressed.

For example, a circuit with five inputs might have the following External Tab settings:

| External Port <br> Name | Encoding <br> Type | Event Syntax | Event <br> Type | Min <br> TD | Max <br> TD | Filter <br> Condition |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| input1 | "mdl" | "SimpleEvent" | 1 |  |  | true |
| input2 | "mdl" | "AMS" |  | 0 s | 10 s | true |
| input3 | "mdl" | "AMS" |  | 10 s | 15 s | true |
| input4 | "mdl" | "AMS" |  |  |  | true |
| input5 | "mdl" | "AMS" |  |  |  | true |

Events with an Event Syntax of "SimpleEvent" and an Event Type of 1 will enter input1 only. Events with an Event Syntax of "AMS" enter the other input ports as follows:

- If the event has a TD (Transit Delay) of 10 seconds or less then it will enter input2.
- If the event has a TD of 10 seconds to 15 seconds then it will enter input3.

Events with a TD of exactly 10 seconds will enter both input2 and input3.

- Otherwise, if the event has a transit delay of more than 15 seconds then it will enter both input4 and input5, because both these inputs specify the same conditions.


## Global Definitions

Global definitions are named constant values. A Global Definition is available to all the nodes within the circuit, including any Compound nodes inside it.

Global definitions are specified in the Global Definitions dialog box displayed by selecting Circuit: Global Definitions from the ECS Designer menu.

Use global definitions to improve the readability of expressions by substituting short meaningful names for complex values, and to factor out values and functions that are used in many places in a circuit.

Global values are defined with the val keyword; global functions are defined with the fun keyword; and global type definitions are defined with the type keyword.

Global values Global values have the form:
val constantName = value
Each global value is introduced with the val keyword, followed by the name, an equals sign (=), and the value.

For example, to declare a value to test the generic-trap field of SNMP traps, you could define the following Global Definition:

```
val enterpriseSpecific = 6
```

Circuits and Compound Nodes

## Compound Nodes

## Function definitions

## Type definitions

This binds the integer value 6 to the global name enterpriseSpecific. Now you can test a condition by writing:

```
input_event "generic-trap" = enterpriseSpecific
```

Function definitions have the form:

```
fun <funcname> <arg1> [<arg2>...<argn>] [: returnType] =
expression
```

where at least one argument must be supplied. arg1 to argn can be simple argument names as in:

```
fun intplus x y = x+y
```

or can optionally specify argument type constraints (note the use of parentheses):

```
fun intplus (x:integer) (y:integer) = x+y
```

A return type can al so be specified as follows:

```
fun intplus (x:integer) (y:integer) : integer = x+y
```

The following expression would then result in a value of 7 :
intplus 34
See Chapter 10, "Writing ECDL Expressions," on page 319 for information about more complex expression writing.

Global type definitions have the form:

```
type typeName = type
```

For example, suppose you find that you are frequently comparing the agent-addr attributes of two events for equality, perhaps in an expression like:

```
input_event "agent-addr" = inhibitor_event "agent-addr"
```

You could define a type anyEvent and a function equalSource to save typing this expression repeatedly:

```
type anyEvent = any dict, -- Events have a 'Dictionary' data type
fun equalSource (ev1:anyEvent) (ev2:anyEvent) : boolean =
ev1 "agent-addr" = ev2 "agent-addr"
```


## Library node external references

Now, in the Condition parameter of an Unless node, you could write the expression:

```
```

not (equalSource input_event inhibitor_event)

```
```

```
```

not (equalSource input_event inhibitor_event)

```
```

Global definitions are stored with the ECS circuit file. If you convert part of a circuit to a Compound node and save it in the Library, the global definitions are not saved with the Compound node. If you then use that Compound node in a different circuit it will still refer to the old global definitions from the original circuit, and the circuit will not verify.

However, when a Library node is imported, any global definitions associated with it are retained. This means that you can save a Compound node to the library, open it as a top-level circuit, add the required global definitions and then saveit. From now on the Compound has its own global definitions that override any definitions of the same name in circuits that the Compound is imported into.

If you use Node: Make local on a Compound node, it loses its global definitions. If a reference is not resolved then an error will be reported when the circuit is verified.

Scope
Global definitions in Compound nodes form a series of enclosing scopes: the outermost scope is the Global Definitions that bel ong to a top-level circuit. The scope of "gl obal definitions" in nested Compounds follows the nesting of the Compound nodes themselves. Definitions in inner scopes override definitions in outer scopes.

## NOTE

The ECS Designer can be used to edit the Global Definitions of the current top-level circuit only. For example, if you have drilled down into a library Compound node and you select File:Global Definitions from the menu, the top-level circuit Global Definitions are displayed, rather than the Global Definitions for the current library Compound node. To display the Global Definitions that belong to the library Compound node you must load the Compound node as a top-level circuit by selecting circuit: Open from the menu, navigating to the library directory, and opening the appropriate .ecs file.

Circuits and Compound Nodes
Compound Nodes

## Attributes: Passing Information out of a Compound Node

The Count, Rate, and Table primitive nodes have preconfigured attributes that can be referenced from other nodes. Compound nodes too can be configured to have one or more attributes. In other words, attributes are the means by which information is passed out of a Compound node, into the enclosing circuit.
Compound node attributes are created by exposing the attributes of primitive nodes contained inside them. For example, a Compound node could define an attribute called Power_fail_count that makes the value of a Count nodes's count attribute visible outside the compound.

Figure 2-1 Compound Node Attributes


To expose an attribute within a Compound node:

1. With the Compound node open in the ECS Designer, define a unique name for the attribute by clicking the Attributes Tab and adding the appropriate details. In the Attributes Tab of the ECS Designer you enter the details separately as:
```
Exported Attribute Name: Power_fail_count
    Internal Node Name: Power_fail
    Node Attribute Name: count
```


## Compound Node Parameters



There are three steps to parameterizing a Compound node:

1. With the Compound node open in the ECS Designer, define a unique name for the parameter by clicking the Parameter Tab and adding the appropriate details.
2. Within the Compound node's circuit, use the newly defined parameter in ECDL expressions, as you would a Global Definition.
3. Place the Compound node in its enclosing circuit and configure the parameter value just as you would for any other node.

Circuits and Compound Nodes

## Compound Nodes

See the HP OpenView Event Correlation Services Designer's Guide for details.

## Circuit Policy

Compounds imported from the circuit library may have an associated intended circuit policy. If a compound is imported into a circuit with a conflicting policy (e.g. a compound written for an output engine configuration is imported into a circuit written for a discard configuration), a warning message is produced, but the import of the compound is still allowed.

## 3 Circuit Nodes

Circuit nodes are the building blocks with which circuits are constructed. This chapter describes the fifteen primitive nodes in detail. The first section:

- "Common Features of Nodes" on page 46
describes those aspects common to all nodes and describes how to customize node behavior through the node parameters. Following this is a complete description of each node that includes:
- a summary and diagram of the node
- a diagram and detailed description of the processing done by the node
- a description of the node's parameters
- a description of the node's attributes (if any)
- a description of the node's ports
- notes and design hints


## Primitive nodes The primitive nodes described in this chapter are:

- "Annotate Node" on page 58
- "Clock Node" on page 64
- "Combine Node" on page 71
- "Count Node" on page 81
- "Create Node" on page 87
- "Delay Node" on page 93
- "Extract Node" on page 97
- "Filter Node" on page 102
- "M odify N ode" on page 105
- "Rate N ode" on page 109
- "Rearrange N ode" on page 116
- "Sink Node" on page 122
- "Source Node" on page 125
- "Table N ode" on page 128

The Compound node lets you encapsulate a circuit to create your own node. It is described separately in Chapter 2, "Circuits and Compound N odes," on page 23.

## Common Features of Nodes

This section describes features common to all primitive nodes and should be read in conjunction with the node descriptions in the remainder of this chapter.

## Node Configuration

Nodes are manipulated within an ECS circuit using the HP OpenView Communications ECS Designer in Build mode. To configure a node you select the node and then select Node: Configure, or right-click to display the popup menu and select Configure. The Configure dialog box is displayed. Select the Parameter or Port tab, to display the appropriate settings.

E ach node is assigned a default name when it is placed on the canvas. However you can override it with a descriptive name of your own. In addition to a name, most nodes have one or more parameters. Generally, the ECS Designer supplies a default value for each parameter, but these values are frequently useful only as a guide to the form and type of the parameter value. Most of the work in designing a circuit is in writing the ECDL expressions that comprise the node parameters.

As you design the correlation circuit the status of each node is indicated by its color. You can instantly recognize an incomplete or incorrect node configuration by its color.


#### Abstract

Attributes

Node names The expressions you write for node parameters can refer to the attributes of other nodes. You cannot see the attributes of a node in Build mode. But, if your circuit verifies and runs a simulation successfully, you can view the current values of a node's attributes. These values are displayed in the Attributes page of the Node Status dialog, reached by selecting Simulate: Node and then selecting the Status Attributes tab.

\section*{Name and Status}

When you first place a node on the canvas, the ECS Designer gives it a default name constructed from the node type, a number, and underscores in the general pattern:


```
nodeType_n_
```

where $n$ is the next available number. For example, the default name of the first Count node that you place on the canvas is count_1_. You can change this name in the Configure dialog reached by selecting Node : Configure.

The trailing underscore causes the node name to be suppressed on the screen. To make the node name visible, delete the trailing underscore.


CAUTION If you change the name of a node that has attributes that are referenced by other nodes, you must change all references to the node throughout the ECS circuit. If you change a node name and forget to update a reference then the ECS circuit will fail to verify.

Status indications The fill, border, and background colors of a node vary to indicate its status. There are two major groups of variations: those you see in Build mode, and those you see in Simulate mode.

Building and The following variations in node appearance are displayed in both Build simulating circuits and Simulate modes.

| Node <br> Appearance | Color and <br> Border | Description |
| :--- | :--- | :--- |
|  | Light fill, black <br> border | In Build mode this indicates a correctly <br> parameterized and connected node. In <br> Simulate mode it indicates the node is <br> inactive and no breakpoint is set. |
|  | Gray surround <br> with resizing <br> handles | Selected |

Circuit Nodes
Common Features of Nodes

| Node <br> Appearance | Color and <br> Border | Description |
| :--- | :--- | :--- |
|  | Dark gray drop <br> shadow | Library node (Compound node) |

Building circuits These node variations appear in Build mode only. only

| Node <br> Appearance | Color and <br> Border | Description |
| :--- | :--- | :--- |
|  | Pale blue fill | Parameters missing or active ports not <br> connected |
|  | Red fill | Verification failed |
|  |  |  |

Simulating circuits These node variations appear in Simulate mode only. only

| Node <br> Appearance | Color and <br> Border | Description |
| :--- | :--- | :--- |
|  | Red outline | Breakpoint set |
|  | Green fill | Processing event when tracing or <br> stepping |
|  |  |  |


| Node <br> Appearance | Color and <br> Border | Description |
| :--- | :--- | :--- |
| $\square$ | Purple frame | Location of current event when tracing or <br> stepping |
|  |  |  |
|  |  |  |

## Node Parameters

Many nodes have parameters. Parameters are used to customize node behavior. To change a node's behavior, you select the Parameters Tab of the node Configure dialog, reached by selecting Node: Configure in Build mode. The expressions you enter in this dialog box control aspects of the node's behavior.

## Expressions

All parameters take arbitrarily complex expressions. You can type a simple literal value such as the value 500 in the Max Events parameter in Figure 3-1. On the other hand, this example also shows the save Until parameter fetches a value called TableTimeout from the Data Store. Save Until is assigned whatever value TableTimeout has at the time the expression is evaluated.

Expressions are written in a language called ECDL (Event Correlation Description Language). See Chapter 10, "Writing ECDL Expressions," on page 319 for details.

Expressions in node parameters can refer to:

- other node's attributes using dot notation (nodeName. attribute)
- the Data and Fact stores
- surrounding values such as global definitions or the parameters of a compound node containing the node


## Common Features of Nodes

Figure 3-1 Configure Dialog-Parameters Screen


Event names To read the value of an event attribute from an ECDL expression you need to use the appropriate event name. Event names are predefined expressions that provide access to the dictionary structure that represents an event. The predefined event names are listed in Table 3-1.

Table 3-1
Event Names

| Name | Node(s) this event name is defined for |
| :--- | :--- |
| input_event | Annotate Create Extract Filter Modify Unless <br> Also used in the Filter Condition on the External Tab. |
| output_event | Combine |
| created_event | Create |
| current_event | Table |
| retained_event | Table |

## Table 3-1 Event Names

| Name | Node(s) this event name is defined for |
| :--- | :--- |
| inhibitor_event | Unless |

To retrieve an attribute value from a primitive event, you simply specify the event attribute you want. For example,

```
input_event create_time
```

retrieves the value of the create_time attribute from the event header. Generally header attributes are defined by tokens, whereas body attributes are defined as strings. For example, to retrieve the "equipmentId" body attribute we place double-quotes around the string, as in:

```
input_event "equipmentId"
```

When retrieving attribute values from a composite event you need to specify the "path" to the event you are interested in. For example,

```
input_event 2 1 create_time
```

retrieves the create_t ime event header attribute from the first event inside the second event inside the composite event referred to by input_event. SeeChapter 10, "Writing ECDL Expressions," on page 319 for details.

## Common Features of Nodes

## Static and dynamic evaluation

Some node parameters are statically evaluated, others are evaluated dynamically. When an expression refers to a Data Store or a Fact Store, you must be aware of when the expression is evaluated:

- Statically evaluated expressions are evaluated once when the circuit is loaded. Data or fact store values are retrieved at this point and any subsequent updates to the stores have no effect on the parameter value.
- Dynamically evaluated expressions are evaluated each time the value is required. Updated Data or Fact Store values are reflected in the evaluated parameter values.

In the example shown in Figure 3-2 on page 53, the Retain Condition parameter is dynamically evaluated for each input event arriving at the Table node.

In the description of each node, the Evaluation column in the table of parameters for each node indicates whether the parameter is evaluated statically or dynamically. Generally, parameters that define a condition are evaluated dynamically, and all others are evaluated statically.

## CAUTION

The Data Store and Fact Store can be updated while the ECS Engine is running. However, statically evaluated expressions are not re-evaluated when a store is updated. You must disable, reload, then re-enable the ECS circuit to cause statically evaluated expressions to be re-evaluated. See the HP OpenView Communications Event Correlation Services Administrator's Guide.

## Data types

Parameter expressions must evaluate to the correct data type for that parameter. You must ensure that the value returned from the expression you enter will be of the correct type. In the example in Figure 3-1 on page 50, the Max Events parameter is of data type Integer. If you supply a different data type, or the expression you enter returns a different data type, then an error occurs when the expression is evaluated.

Data type errors in dynamically evaluated expressions trigger an error only when they are evaluated. Remember that statically evaluated parameters are evaluated when the circuit is loaded but dynamically evaluated parameters are not evaluated until an event passes through the node, which may be some time later.
The Type column in the table of parameters for each node identifies the
return data type required by each parameter. The Default column shows the value used if you do not supply your own value.

## Input and Output Ports of Nodes

Each node has one or more input and output ports. Some ports are required, others are optional. The Configure dialog, displayed by selecting Node: Configure in the ECS Designer Build mode, controls optional ports. You click check boxes in the Ports page of the Configure dialog, as shown in Figure 3-2 on page 53 to activate optional ports.

Figure 3-2 Configure Dialog—Ports Screen



Ports are positioned in the same order as they are listed in the dialog.

In this example there are three active ports: Input, Output and Fail Output.

Input ports $\quad$ Nodes may have one or more of the following input ports:
Input This is the main input port for most nodes. Generally, at least onemain input port must be connected.

Reset Input When any event arrives at this port, it causes the node to reset to a known condition. Typically, any events waiting at a main input port are flushed out of a Fail output port. Attributes and parameters may be reset

## Common Features of Nodes

|  |  | to initial or default values. This port need not be connected. |
| :---: | :---: | :---: |
| Output ports | Nodes may have one or more of the following output ports: |  |
|  | Output | This is the main output port from which most nodes transmit processed events. |
|  | Error Output | Some parameters have expressions which can include events as arguments. If the expression cannot evaluate the event, the event is transmitted from this port. |
|  |  | The Error Output port need not be connected but, if not connected and an error occurs, an error message is appended to the engine log file if logging is enabled. |
|  | Fail Output | When an event arrives at the Reset Input port, any events waiting at a main Input port are transmitted from the Fail Output port. This port need not be connected. |
|  | Reset Output | Any event received by the Reset Input port of a node is immediately transmitted from the node's corresponding Reset Output port. This lets you chain Reset Output ports to Reset Input ports to selectively reset only certain nodes in a chain, without affecting other nodes. This port need not be connected. |
| Event classes | The table of port details in the description of each node includes a col umn labelled "E vents" that shows what classes of events the port transmits: |  |
|  | P | Primitive |
|  | C | Composite |
|  | T | Temporary |
|  | See Chapter 4, event types. | Events in ECS," on page 147 for a detailed description of |
| Identifying ports | In the ECS Desi To identify a por Connection from identify. The nam canvas. | gner, ports on the canvas are not specifically identified. t in Build mode, right-click on the canvas, select the pop-up menu, and click on the port you want to me of the port is displayed in the status line, below the |

## Figure 3-3 Node Status Dialog-Attributes Screen



## Referencing attributes

Filter example

## Node Attributes

The Count, Rate, and Table nodes have attributes that can be read by other nodes in the circuit. After you have verified a circuit and run the Simulator, you can view the values of these attributes in the Attributes page of the Node Status dialog. You reach this dialog by selecting Simulate: Node Status in Simulate mode. See Figure 3-3 on page 55.

To access a node's attributes you use dot notation:
nodeName. attributeName
For example, to refer to the Count attribute of the count_1_ node pictured above, in an expression you would type:
count_1_. Count
For example, say you want to set up a circuit to detect dropout notifications from a power supply and ignore them unless the rate of notifications is greater than 5 per second.
A Rate node named Intermittent is set with an Interval parameter of 1 s . A Filter node named PassIntermittent is set with a condition parameter of Intermittent. Rate > 5. The Filter node examines the Rate attribute of the Rate node whenever the Condition is evaluated.

## Common Features of Nodes

The result is that PassIntermittent passes events only when Intermittent detects a rate greater than 5 events per second.

## Transit Delays and Simulation Statistics

The Annotate, Combine, Delay, and Unless nodes can delay events during processing. Keep these delays to the minimum necessary for the circuit to operate correctly. Otherwise, the circuit can become inefficient. Chapter 5, "Timing Considerations," on page 165, contains general background details.

After a simulation you can view transit delay times and event counts in the N ode Status dialog to monitor the performance of your circuit and to locate delays. You display this dialog in Simulation mode by selecting Simulate: Node Status and selecting the Statistics button. See Figure 3-4.

Figure 3-4

Interpreting statistics

Node Status Dialog-Statistics Screen


The Node Status dialog presents statistics in the form
portName_statisticType
where statisticType can be maxTD, minTD, numin, or numout. For example, the statistics for the Input port of a Filter node are input_maxTD, input_minTD, and input_numin (numout is inappropriate
for an input port). The following table summarizes the available statistics:

| Statistic <br> Type | Input <br> Ports | Output <br> Ports | Description |
| :--- | :--- | :--- | :--- |
| maxTD | Yes | Yes | Maximum transit time from origin <br> to port |
| minTD | Yes | Yes | Minimum transit time from origin <br> to port |
| numin | Yes | No | Number of events that arrived at <br> an input |
| numout | No | Yes | Number of events that departed an <br> output |

## Restrictions on recursion

Transit delays are propagated through the circuit by nodes which may detain events. The cumulative delays are calculated automatically by the engine. This has the side-effect that recursion within a circuit (connecting a downstream node's output to the input of an upstream node) is disallowed when any of the nodes in such a loop are capable of detaining events. This means that loops must not be constructed from the following nodes:

- Unless node
- Delay node
- Annotate node
- Combine node
- Table node


## Annotate Node

The Annotate node holds events that arrive at its Input port while a request for external data is made. If a response is received within a specified time, a composite event is transmitted containing the original input event annotated with the response data.

Figure 3-5 Annotate Node


The Annotate node is used to export data to, or import data from, an external process called an annotation server, without delaying other events in the circuit. This asynchronous action ensures that a potentially slow lookup or search conducted by the annotation server does not affect other areas of the circuit.

The annotation server is an external process supplied by the user and is not part of the HP OpenView Communications ECS product.

## Annotate Node Processing

Figure 3-6 Annotate Node Processes


On receipt of an event at its Input port, the Annotate node evaluates the Annotate Spec parameter, which must return a List data type. The List is passed directly to the annotation server.
When the annotation server produces a response, it is forwarded directly back to the Annotate node that issued the corresponding request.

If the response is received within the time specified in the Time Limit parameter, it is used to create a temporary event (the response event) that is coupled with the input event in a composite event, and transmitted from the Output port.

The transmitted composite event has the following form:


The response event can contain any number of data elements of any data type. Each element is an attribute of the temporary event, accessed by an integer key starting from 1 . That is, if the temporary event has three attributes, their keys are 1,2 , and 3 . It is the responsibility of the circuit designer and the annotation server developer to ensure that returned data is correctly interpreted.

The following examples show how a downstream Filter node might access data in the composite event:

| Expression | Attribute |
| :--- | :--- |
| input_event 1 "equipmentId" | Returns the "equipmentId" <br> attribute of the input event <br> (the first event). |
| input_event 2 1 | Returns the first attribute of the <br> temporary event <br> (the second event). |


| Expression | Attribute |
| :--- | :--- |
| input_event 2 2 | Returns the second attribute of the <br> temporary event (the second <br> event). |

If the response is not received within the time specified in the Time Limit parameter, the input event is transmitted from the Fail Output port. If an overdue response is finally received it is discarded. If the Fail Output port is not connected then the input event is discarded.

If there is an error evaluating the Annotate Spec parameter, the input event is immediately transmitted from the Error Output port, and no request is generated. If the Error Output is not connected, an error message is logged if logging is enabled.

U pon receipt of an event at the Reset Input port, all pending annotations are abandoned, their originating events are transmitted from the Fail Output port, and the event arriving at the Reset Input port is transmitted from the Reset Output port (or discarded if the Reset Output port is not connected).

See the HP OpenView Communi cations Event Correlation Services Devel oper's Guide \& Reference for a complete explanation of the annotation process and implementing an annotation server.

## Annotate Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Annotate Spec | List (of any type) | none | Dynamic |
| Time Limit | Duration | $0 s$ | Static |

## Annotate Spec

Annotate Spec is a dynamically evaluated expression that determines the data passed to the external annotation server. This parameter is a List that can contain any data type for any member. The circuit designer must ensure that the data in this list is correctly formatted for the appropriate annotation server.
An example Annotate Spec is:

```
[ "getEquipmentDetails", input_event "equipmentId" ]
```


## Annotate Node

In this example, the first member of the list is a string describing the type of annotation request, "getEquipmentDetails". The second member, input_event "equipmentId" is an expression that returns the "equipmentId" attribute from the input event.
There is no limit on the number or types of members in the list, but it is essential that the list conforms to the expectations of the annotation server.

The name of the Annotate node, the name of the circuit, and the Time Limit value are implicitly passed as part of the request. The Time Limit is passed as an absolute time (current time + Time Limit).

Time Limit
Time Limit is a statically evaluated expression that determines the maximum time an input event is held, while a response is pending.

For example, if the Time Limit parameter is 10 s then the input event is held for up to 10 seconds while awaiting the response. If a request is not received within 10 seconds then the input event is transmitted from the Fail Output port.

If a response is received after the time limit has expired it is discarded.

## Annotate Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Reset Input | No | No | PCT |
| Output | Yes | Yes | C |
| Error Output (logged) | No | No | PCT |
| Reset Output | No | No | PCT |
| Fail Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

Any event transmitted from the Output port is a composite event containing the original input event and temporary event carrying data returned from the annotation server.

## Annotate Node Notes

A single Annotate node can annotate many events concurrently. The Annotate node combines the correct response with each input event even when many events are in the process of being annotated.

Do not send the composite events generated by an Annotate node to the Sink node of an ECS circuit. Only primitive events can be transmitted from an ECS circuit.

When determining the Time Limit value you must consider the required annotation server processing time and the two-way communication delay. The Time Limit should be set to the minimum reasonable value. An excessive Time Limit can delay input events, and can have effects on events in other circuit paths.

## Clock Node

The Clock node generates temporary events at specified intervals, after the internal metronome is started by an event on the start Input. It stops generating events when an event is received at the Stop Input.
The Clock node can also be started when the engine starts, and the timing of the first clock event can be aligned with the real-time clock (UTC).

Figure 3-7 Clock Node


## Clock Node Processing

Figure 3-8 Clock Node Processes


| $\square$ Connection required | $\square$ | Parameter ("input") |
| :--- | :--- | :--- |
| $\triangleright$ Connection optional (unconnected o/p discards events) | $\square$ | Parameter evaluated |
| $\rightarrow$ Event flows | $\rightarrow$ | Logic flows |

The Clock node generates empty temporary events and emits them from the Output port. See Figure 3-8. The Interval parameter determines the time between events. If the Autostart parameter is true then the Clock node starts when the correlation engine starts. Otherwise it starts or restarts when an event is received at the start Input port. The Clock node stops generating events when it receives an event at the stop Input port.

The timing of the first event generated by the Clock node after a start or restart depends on the setting of the Time Alignment parameter. Subsequent events are separated by the Interval. This allows events generated by the Clock node to be aligned with the external real-time clock.

Time points are defined by the Time Alignment tuple parameter such that the first component of the tuple describes a time period and the second describes the offset. For example, to define a period of 1 hour and
an offset of 15 minutes 30 seconds set Time Alignment to ( $1 \mathrm{~h}, 15 \mathrm{~m} 30 \mathrm{~s}$ ).
The create_time header attribute of the temporary event generated by the Clock node is set to the value of current time when the event is generated. The arrival_time header attribute is set to the time when the event is transmitted from the node, which may be up to one second later.

Events entering the Start Input or Stop Input are transmitted from the start Output or Stop Output respectively, or discarded if these ports are not connected.

## Clock Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Interval | Duration | 1 s | Static |
| Auto Start | Boolean | false | Static |
| Time Alignment | (Duration, <br> Duration) | (0s, 0s) | Static |

## Interval

Auto Start

Time Alignment

Interval is a statically evaluated expression that determines the period between events generated by the Clock node. The default value is 1 s .

For example,

```
9h21m14s * 2
```

causes a Clock node to generate an event at intervals of 18 hours, 42 minutes, and 28 seconds.

The Interval parameter is not truncated. That is, a value of 0.1 s causes the Clock node to generate 10 temporary events every second, and a value of 0.4 s generates 5 events every 2 seconds.

Auto Start is a statically evaluated expression that determines whether the Clock node starts generating events when the correlation engine is started. If the expression evaluates to true the Clock node starts when the engine starts. If the expression evaluates to false the Clock node does not start until it receives an event on the Start Input port. The default value is false.

Time Alignment is a statically evaluated expression that determines a
series of time points with which generated events can be aligned. It can be any expression that evaluates to a tuple of two duration data types of the form (period, offset). The default value is ( $0 \mathrm{~s}, \mathrm{0s}$ ).

- If Time Alignment is set to ( $0 \mathrm{~s}, 0 \mathrm{~s}$ ), no time alignment occurs and the first clock event is transmitted Interval seconds after the Clock node starts, provided no event has arrived at either the Start Input or the Stop Input in the meantime.
- If the Time Alignment parameter is set to a value other than (0s, 0s), the first event is transmitted on the first available time point defined by the parameter, provided no event has arrived at either the Start Input port or the Stop Input port in the meantime.

The Time Alignment parameter determines the time point of the first event generated by the clock after being started or restarted. The second and subsequent events are generated at regular intervals after that, as specified by the Interval.

The first time point is calculated as follows:
when (current time mod period) $=0$,
wait for offset seconds,
then generate the first time point.
The effect is to align the first time point with the correlation engine's clock such as ' 1 minute past the hour' or '30 seconds past midnight'. This is shown graphically in the following diagram.

## Clock Node

Figure 3-9 Time Alignment
Time Alignment $=(0 h, 5 h, 30 \mathrm{~m})$
Interval $=1 \mathrm{~h}$


1. An event arrives at the Start Input port.
2. When Current time mod period $=0$ (at 04:40) the engine waits for offset to elapse before emitting the first event. 05:10 is the first time point after the start event arrives.
3. The second event is emitted after the duration specified in the interval has elapsed. In this example, at 07:40, which is 2:30 after 05:10. E ach subsequent event is emitted separated by the interval. Note that second and subsequent events do not have to align with time points; only the first event in a series must align with a time point.
4. An event arrives at the stop Input so the clock stops emitting events.

If another event arrives at the Start Input port at any time the process starts over at step 1 again.
There are a number of constraints on the values that the Time Alignment tuple can take:

- period must be a positive number of seconds. If a real number of seconds is specified it is truncated to an integer. Negative numbers are treated as 0.
- period must be less than or equal to Interval.
- offset must be less than or equal to period. Otherwise, offset is set to offset mod period.
Note that the current time is measured from the epoch at 00:00 1 J an 1970 Coordinated Universal Time (UTC). The effect of the Time Alignment parameter is only likely to be meaningful if it exactly divides the number of seconds in a day, hour or minute. If you choose, for example, 7 s for the offset, then the first time point could be at any second of the hour, depending on the date.

The following examples assume the Clock node is started at UTC 00:00:00.

| Time <br> Alignment <br> (period, offset) | Interval | First event | Second event | Third <br> event | Comments |
| :--- | :--- | :--- | :--- | :--- | :--- |
| $(1 \mathrm{~h}, 15 \mathrm{~m} 30 \mathrm{~s})$ | 1 h | $01: 15: 30$ | $02: 15: 30$ | $03: 15: 30$ |  |
| $(15 \mathrm{~m}, 0 \mathrm{~s})$ | 1 h | $00: 00: 00$ | $01: 00: 00$ | $02: 00: 00$ |  |
| $(-1 \mathrm{~m},-1 \mathrm{~s})$ | 1 h | $00: 00: 00$ | $01: 00: 00$ | $02: 00: 00$ | Negative values are <br> treated as 0. |
| $(2 \mathrm{~h}, 0 \mathrm{~s})$ | 1 h | Compile error - see error log. The period must be less than or equal <br> to the interval. |  |  |  |
| $(2 \mathrm{~m}, 3 \mathrm{~m})$ | 1 h | $00: 03: 00$ | $01: 03: 00$ | $02: 03: 00$ | offset $>$ <br> so offset is set to <br> offset mod <br> period. 3 mod $2=$ <br> 1, |

## Clock Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Start Input | No | Yes | PCT |
| Stop Input | No | No | PCT |
| Output | Yes | Yes | T |
| Start Output | No | No | PCT |
| Stop Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

## Clock Node Notes

Do not send the temporary events generated by a Clock node to the Sink node of an ECS circuit. Only primitive events can betransmitted from an ECS circuit.

Place Filter nodes before the Start Input and Stop Input ports to ensure that only the appropriate events start or stop the Clock node.

## Combine Node

The Combine node combines events received at all of its inputs into composite events. When the components of each composite are in a specified order of creation, their creation times fall within a specified interval, and they satisfy a specified condition, the composite event is transmitted. Otherwise, the Composite event is discarded.

## Figure 3-10 Combine Node

Streams of events


The Combine node lets you group events and deal with them as a single entity. It is typically used to group events that have been created almost simultaneously.

For example, you may have a Create node that is extracting attributes from several different events to place into a newly created event for eventual transmission. For this to be possible the events have to be grouped into a composite event upstream of the Create node such that the Create node can be presented with a single composite event at its input port.

Another example is if you want to form pairs of events that were created only a few seconds apart on the same input stream, so you can compute the trend or create an alarm event if an event attribute changes too rapidly (sometimes called the "recent duplicate suppression" scenario). To achieve this you would use a Combine node with two input ports connected to the same input stream, specify that the creation order must be $[1,2]$ such that the composite contains the older event followed by the younger event, and set the condition such that pairs formed by the same event are discarded.

## Combine Node

## CAUTION

Normally event correlation aims to reduce the number of events. Be careful to avoid situations where a Combine node increases the number of events:

- many input ports
- many events arriving at all input ports at very short intervals
- a long Interval parameter
- a Condition parameter that accepts too many events.

In such situations, the number of possible combinations which the Combine node would build and consider for output can easily number in the thousands, reducing throughput greatly.

## Combine Node Description

Figure 3-11 Combine Node Processes


- Connection required
$\star \quad$ Connection optional (unconnected o/p discards events)
- Connection optional (unconnected o/p logs events)
$\square$ Parameter ("input")
-     - Parameter "switches" event to appropriate path
$\square$ References to node attributes, Data and Fact Stores, and Global Definitions

| $\square$ | Events stored |
| :--- | :--- |
| $\square$ | Parameter evaluated |
| $\rightarrow$ | Event flows |
| $\rightarrow$ | Logic flows |
| $\rightarrow$ | Evaluation error |
| 消 | Event discarded |

The Combine node receives primitive, temporary, or composite events at each Input port. It forms composite events from these input events.
As shown in Figure 3-12, a composite event (C) consists of a reference to

Figure 3-12 Composite Event Built by Combine Node


The Combine node forms composite events from all possible combinations of input events where the input events satisfy both of the following conditions:

- they were all created less than Interval apart, and
- they were created in an order that conforms to the specified Order.

Whenever such a composite is formed it is subjected to the condition parameter. Depending on the result of evaluating the condition the composite event is:

- transmitted from the Output port if Condition evaluates to true, or
- removed if Condition evaluates to false, or
- transmitted from the Error Output port if the evaluation results in an error.

E ach time an event arrives at an Input port, the Combine node builds all possible composite events that can be formed from that event and events already retained on the other Input ports. E ach composite event consists of a reference to the newly arrived event and a reference to one event taken from the events retained at each of the other Input ports. All possible combinations are built.

The create_time attribute of each composite event is set to that of its youngest component event and its arrival_time attribute is set to the current time.

The node then tests each composite event with the Order and condition parameters. For each composite event, if both the Order and Condition
parameters evaluate to true the event is transmitted from the Output port. Otherwise, if evaluation of Condition raises an error, the event is immediately transmitted from the Error Output port.
Since there can be many events retained at the input ports, the Combine node repeats the process for every combination of the newly arrived event and the currently retained events on the other ports. If one or more of the other input ports does not have any retained events, no composite event is built.

Finally, the newly arrived event is retained at the input port. It is necessary to retain the event as it may form combinations with events that have not yet arrived or perhaps have not even been created yet.
Events retained at the Input ports are removed from the node whenever the current time advances past the point where all newly arrived events on the other input ports have a creation timethat is morethan Interval after the creation time of the retained event. The maximum transit delays for the input ports are used as a basis for this decision.
When an event arrives at the Reset Input port, all currently retained events are removed from the Input ports and discarded. The Combine node is then in the same state as when the circuit was initialized. The reset event is immediately transmitted from the Reset Output port if the port is connected.

## Combine Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Interval | Duration | None | Static |
| Order | List of up to $n$ <br> Integers $^{\mathrm{a}}$ | []$^{\mathrm{b}}$ | Static |
| Condition | Boolean | true | Dynamic |

a. All elements in the Order list must be between 1 and $n$ inclusive (where $n$ is the number of input ports) and each number may occur once at most.
b. Empty list.

Interval is a statically evaluated expression that determines the time window within which events are combined. There is no default value, so

## Combine Node

a value or expression that resolves to a duration data type must be provided.

The Combine node builds composite events where the creation times of all the component events are separated by less than the value of the Interval parameter.

For example,
42s
causes the Combine node to consider combinations of received events that were created up to 42 seconds apart. That is, if we use D (delta) to represent the difference, then for three events:

```
create_time 1 D create_time }<<=42, and
create_time 2 D create_time 3}<=42, and
create_time 3 D create_time }<<=4
```


## Order

Condition

Order is a statically evaluated expression that stipulates a sequence in which input events must have been created. The expression must resolve to a List data type containing integers between 1 and $n$ inclusive, where $n$ is the number of input ports. Each number is allowed to occur at most once. The default value is [] (empty list), placing no restrictions on the order of creation times of the input events.

After building composite events, the Combine node tests each composite against the order parameter. To pass the test, the creation times of the component events in the composite must be in the sequence specified by the Order parameter.

For example, the order parameter
[2,1,4]
requires that the event received at Input 2 is created before the event that arrives at Input 1 , which must have been created before the event that arrives at Input 4. All events must be received within the time specified by the Interval parameter. If there are other ports not mentioned in the order parameter list, say Input 3, then the relative ordering of events at this port does not matter.

Condition is an expression that is dynamically evaluated once for each composite event. If the expression evaluates to true then the event is immediately transmitted from the output port. If the expression
evaluates to false then the composite event is discarded. The default value is true, allowing all events to pass.

If an unhandled exception is raised or the expression returns something other than a boolean data type then the composite event is transmitted from the Error Output port.

Because the condition parameter is dynamically evaluated it may refer to the composite event using the event name output_event and it may refer to other nodes' attributes and to the Data and Fact Stores.

If the expression refers to the components of the composite event, you must use the event name output_event $n$, where $n$ refers to the required component event. That is, output_event 3 refers to the third component event of the composite. In Figure 3-12 on page 74, this would be a copy of event $I_{3}$ inside the composite event $C$.

For example, if the second and third components are primitive events, the expression:

```
(output_event 2 "generic-trap" = 4)
and
(output_event 3 "generic-trap" = 6)
```

transmits a composite event only if the "generic-trap" attribute of the second component is 4 and the "generic-trap" attribute of the third component is 6 .

Circuit Nodes
Combine Node

## Combine Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input1 | Yes | Yes | PCT |
| Input2 | Yes | Yes | PCT |
| Input3...n | Yes | No | PCT |
| Reset Input | No | No | PCT |
| Output | Yes | Yes | C |
| Error Output | No (logged) | No | C |
| Reset Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary .

A newly placed Combine node has only two Input ports, both of which must be connected. You can add more input ports, each of which is named Input followed by an integer, for example, Input 21. Up to 50 input ports are possible but, in practice, too many ports degrades performance.

## Combine Node Notes

Do not send the composite events generated by a Combine node to a Sink node of an ECS circuit. Only primitive events can betransmitted from an ECS circuit.

Reduce processing overhead by preceding a Combine node with Filter nodes, upstream of the Input ports, that pass only events required by the Combine node. Do not use the condition parameter to test properties of individual input events that could have been tested in a filter before the Combine node.

## CAUTION

Normally, event correlation aims to reduce the number of events. If many events arrive at the input ports within the specified interval then many composite events are built. Even though many of these events may be rejected by the condition specified and are never visible outside the node, they must all be built and tested. To reduce the number of composite events generated, minimize the value of the Interval parameter
consistent with the intended operation of the node.

References to the original input events are used (rather than copies) because each retained event has the potential to be included in several composite events. By using a reference, the identity of the event (its unique_id) is preserved and efficiency is maintained.

## Compound Node

Compound nodes are nodes containing an encapsulated circuit.
Compound nodes can have parameters and attributes, just like primitive nodes.

See Chapter 2, "Circuits and Compound Nodes," on page 23 for details.

## Count Node

The Count node maintains counters (count and count_dict attributes) that can be incremented and decremented by events entering or passing through it. The values of these attributes are made available to other nodes in the ECS circuit.

## Figure 3-13 Count Node

Event increases the value of the relevant counter(s) by 1


The count and Count_dict attributes maintained by the Count nodecan be referenced by expressions in the parameters of other nodes to trigger an action when a specific value is reached. For example, a Filter node may discard events as soon as the count attribute reaches 20.

## Count Node

## Count Node Processing

Figure 3-14 Count Node Processes


| $\square$ | Connection optional (unconnected o/p discards events) | $\square$ | Parameter evaluated |
| :--- | :--- | :--- | :--- |
|  | At least one connection required | $\square$ | Attribute ("output") |
| $\square$ | Parameter ("input") | $\rightarrow$ | Logic flows |

The initial values of the counters are defined by the Initial Count parameter.

The count attribute is incremented by each event entering at the Increment Input port and decremented by each event entering at the Decrement Input port.

Events entering the Increment Input or Decrement Input are transmitted immediately from the Increment Output or Decrement Output port respectively, or discarded if these ports are not connected.

The count attribute is reset to the value of the Initial Count parameter whenever an event arrives at the Reset Input port or the Clear Input port. Events entering these ports are transmitted from the Reset Output and Clear Output ports respectively, or discarded if the
ports are not connected.
The count_dict attribute is a dictionary comprising counter values keyed by the evaluation of the Key Spec parameter.
The optional key spec parameter is an ECDL function that takes the input_event. The evaluation of this parameter must result in a value of type "Any Simple". If the Key Spec parameter is omitted, the Count_dict is not used and will remain empty.
For example, a Key Spec may be: input_event "device_id".
This will extract the device_id attribute from the input event, and increments, decrements, or clears will apply to the Count_dict element keyed by device_id.

When an event arrives at the increment_input, decrement_input, or clear_input ports, the following process takes place:

1. The Key Spec parameter is evaluated for the incoming event. If the parameter evaluates successfully, the process continues at Step 2. If, however, there is an error during the evaluation of the Key Spec parameter, then one of the following actions occurs:

- If the Error Output port has been configured, the input event is sent to the Error Output port.
- If the Error Output port has not been configured, the error is logged in the engine log.

2. The Count_dict attribute is keyed with the result of Step 1. If this is the first time a given key is used, the dictionary element is created and the value set to Initial count.
3. The value of the keyed element of the count_dict attribute is incremented, decremented, or set to Initial Count respectively.

## Count Node

## Count Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Initial Count | Integer | 0 | Static |
| Key Spec | Any Simple | None | Dynamic |

## Initial Count

Key Spec

Initial Count is a statically evaluated expression that determines the initial value of the counters in the count and elements of the Count_dict attributes. It is also the value to which these counters are reset after an event is received on the Reset Input port. The default value is 0 .

Key Spec is a dynamically evaluated expression that determines the keyed element of the count_dict attribute. There is no default value.

## Count Node Attributes

| Attribute | Type |
| :--- | :--- |
| Count | Integer |
| Count_dict | Dictionary |

Count
Count is an integer value that is incremented or decremented by events entering the node. This attribute can be referenced by other nodes using dot notation syntax nodeName. Count, where nodeName is the name of a specific Count node.

For example, to configure a Filter node to pass events through the False Output port if the Count attribute of Count node Counter24 has exceeded 20, you enter the following condition:

Counter 24 . Count $<=20$

Count_dict is a dictionary containing named counts. The integer value of a dictionary element is incremented or decremented by events that cause the key Spec parameter to evaluate to the name of the element. Dictionary elements can be referenced by other nodes using dot notation syntax nodeName.Count_dict elementName, where nodeName is the name of a specific Count node and elementName is the name of an element in the count_dict attribute.
For example, to configure a Filter node to pass events through the false output port if the number of events from the same device is greater than 20 , you configure the Count node Counter 24 with a Key Spec of input_event "device_id" and the use the following condition for the Filter node:

Counter24.Count_dict (input_event "device_id") <= 20

## Count Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Increment Input | No $^{\text {b }}$ | Yes | PCT |
| Decrement Input | No | No | PCT |
| Reset Input | No | No | PCT |
| Increment Output | No | Yes | PCT |
| Decrement Output | No | No | PCT |
| Reset Output | No | No | PCT |
| Clear Input | No | No | PCT |
| Clear Output | No | No | PCT |
| Error Output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.
b. At least one of the Increment Input or Decrement Input ports must be connected.

## Count Node Notes

The count and count_dict attributes can have any value in the range

Circuit Nodes

## Count Node

$-2,147,483,648$ to $2,147,483,647$. If either of these bounds are exceeded the value wraps. For example, if the count attribute has a value of $2,147,483,647$ and an event enters the Increment Input port the count attribute is changed to a value of $-2,147,483,648$.

## Create Node

The Create node creates a new primitive event for each event that arrives at its Input port.

Figure 3-15 Create Node


The Create node, in effect, acts as a management agent because it permits you to create a new event. The event can then be further processed by the ECS circuit and also, once it leaves the correlation engine, by external management applications such as an event browser or alarm manager. The new event can contain data describing a situation that the correlation engine has recognized by correlating a pattern consisting of many received events.

## Create Node Processing

On receipt of an event at its Input port, the Create node creates a new event using the specified Encoding Type and Event Syntax, and then evaluates the Create Spec parameter, which can refer to the input and created events using the event names input_event and created_event, respectively.

## Create Node

## Figure 3-16 Create Node Processes



- Connection required
$\downarrow \quad$ Connection optional (unconnected o/p logs events)
$\square \quad$ Parameter ("input")
-     - Parameter "switches" event to appropriate path

5 References to node attributes, Data and Fact Stores, and Global Definitions

- Parameter evaluated
$\rightarrow$ Evaluation error
$\rightarrow$ Logic flows
$\rightarrow$ Event flows
I Input event
C Created event

```
input_event identifies the incoming event.
created_event
```

identifies the incoming event.
identifies the event created by the node after evaluation of the create Spec parameter.

If an error occurs during the evaluation of the create Spec parameter then the incoming event is immediately transmitted on the Error Output port and the created event is discarded.

## Create Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Encoding Type | String | None | Static |
| Create Spec | Void or Event | None | Dynamic |
| Event Syntax | String or Oid | None | Static |

Encoding Type The Encoding Type is a statically evaluated expression that evaluates to a String data type. The Encoding Type parameter identifies the endecoder that is used to create the event. It must be one of the String values listed in Table 3-2:

## Table 3-2

Valid Encoding Types for Create Nodes

| Event | Value |
| :--- | :--- |
| ASCII | "mdl" |
| CMIP and SNMP | "ber" |
| OVO Messages | "OpC_Msg" |

## Event Syntax

Event Syntax is a statically evaluated expression that evaluates to either a String or Oid data type.
Table 3-3 Valid E vent Syntax Values for Create Nodes

| Event | Example | Values |
| :--- | :--- | :--- |
| ASCII | "SimpleEvent" | Any String value that identifies a <br> currently loaded MDL event syntax. |
| CMIP | 2.9 .3 .2 .10 .4 | Any Oid value that represents a <br> GDMO notification specification. |
| SNMP | "Trap-PDU" | The ASN.1 syntax identifier for <br> SNMP Traps. "Trap-PDU" is the only <br> valid value. |
| OVO Messages | "OpC_Msg" | The OVO Message syntax identifier. <br> "OpC_Msg" is the only valid value. |

Circuit Nodes

## Create Node

```
Create Spec Create Spec is a dynamically evaluated expression that fills in the
    attribute values of the created event. It is executed for its effect on the
    event only and not for its return value. The value returned by the
    expression is ignored.
    An example Create Spec for an ASCII event (with the syntax
    "SimpleE vent") is:
```

```
let
```

let
val t = explode_time (time.now ())
val t = explode_time (time.now ())
in
in
created_event alter
created_event alter
(
(
"createTime.date.year" => t time.year,
"createTime.date.year" => t time.year,
"createTime.date.month" => t time.month,
"createTime.date.month" => t time.month,
"createTime.date.day" => t time.day,
"createTime.date.day" => t time.day,
"createTime.time.hour" => t time.hours,
"createTime.time.hour" => t time.hours,
"createTime.time.minute" => t time.minutes,
"createTime.time.minute" => t time.minutes,
"createTime.time.second" => t time.seconds,
"createTime.time.second" => t time.seconds,
"deviceId" => "Ephor",
"deviceId" => "Ephor",
"messageType" => "LNKUP",
"messageType" => "LNKUP",
"severity" => 2,
"severity" => 2,
"text" => ""
"text" => ""
)
)
end

```
end
```

The Create Spec parameter can also refer to other node's attributes, the Data and Fact Stores, and surrounding values, as explained in "Expressions" on page 49.

All non-optional event attributes must be set in the Create Spec parameter. In many instances it is easier to use the M odify node to copy and alter an existing event, rather then creating a new event from scratch.

Optional attributes Optional attributes are identified by the ASN. 1 "OPTIONAL" keyword; or by square brackets [...] surrounding a set of MDL attributes. All OVO message attributes are optional. These are the only attributes that can be left out of a create Spec parameter; you must supply values for all other attributes. If an optional attribute is left out of a create Spec expression then it is set to Void or, in the case of OVO messages, to an empty String.

## Create Node Ports

| Port | Connected | Activated | Events $^{\mathbf{a}}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Output | Yes | Yes | P |
| Error output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

Any event transmitted from the Output port is a primitive event conforming to the syntax specified in the Event Syntax parameter.
The Error Output port is inactive by default and, if it is unconnected, all error events are logged to the engine log file if logging is enabled.

## Create Node Notes

In the following example expression for the Create Spec parameter, data is extracted from an SNMP trap (input_event) and placed in a newly created CMIP notification (created_event). The expression is described in steps:

1. Set the "managedObjectClass" to a (fictitious) object class for this Create node. (Note that only the ASN. 1 globalForm of managedObjectClass is supported, so there is no need to express this as a Tuple (CHOICE ).)
```
created_event alter("managedObjectClass" => 1.2.3.4.5.6.7)
```

2. Set the "managedObjectInstance" to a (fictitious) instance representing the managed object that this event is being issued from. N ote that the naming attribute for this object is assigned its value from the "agent-addr" attribute of the SNMP input event:
```
created_event alter
( "managedObjectInstance" =>
    ( "distinguishedName",
        [ [(1.2.3.4.5, "myParentNamingValue") ],
            [(1.2.3.4.6, input_event "agent-addr")]
        ]
    )
)
```


## Create Node

3. Now set up the "eventInfo" part of our new event. Let's first set "probableCause" from the "specific-trap" attribute of the SN MP event.
```
created_event alter("eventInfo.probableCause" =>
    ("localValue", input_event "specific-trap"))
```

4. Assume that all traps have at least one member in the variable bindings and that this member has a string value. Put this value in the "additionalText" attribute of the CMIP event. The string value is inside two levels of ASN. 1 choice, hence the complex syntax.
```
let
    val (_, (_, str)) = input_event
"variable_bindings[0].value"
in
    created_event alter("eventInfo.additionalText" => str)
end
```

5. Set optional attributes that are not present to Void. For example:
```
created_event alter ( "optionalData" => () )
```


## Delay Node

The Delay node detains each input event until the difference between the current time of the correlation engine and the creation time of the event reaches a specified duration. The node then transmits the event.

Figure 3-17 Delay Node
Events arriving out of creation time order
Delayed events depart in creation time order


In some management networks, events may arrive at the correlation engine with wide variations in transit delays. This means that events may not arrive at the correlation engine in the order in which they were created. Although many of the circuit nodes are designed to correlate events arriving out of order, some correlation problems are more easily solved if the events are sorted into creation time order with a Delay node.

## Delay Node Processing

The processing of an event in the Delay node depends on the event's age, which at any time is determined as the difference between current time and the event's creation time.

An event entering the Input port is detained at the Input port if its age is less than the duration specified by the Delay Until parameter. It is detained until current time changes such that the event is the age specified by the Delay Until parameter.

Circuit Nodes

## Delay Node

Figure 3-18 Delay Node Processes


| - | Connection required | $\cdots$ | Events stored |
| :--- | :--- | :--- | :--- |
| $\triangleright$ | Connection optional (unconnected o/p discards events) |  | Time window |
| $\square$ | Parameter ("input") | $\rightarrow$ | Logic flows |
|  |  |  | Event flows |

If the difference between the current time of the correlation engine and the creation time of the event is at least the duration specified by the Delay Until parameter, then the event is transmitted from the Output port immediately.
The effect of the Delay node on the transmitted event stream is illustrated by the following examples:

| Arrival Time | Creation Time | Delay Until | Output Time |
| :--- | :--- | :--- | :--- |
| $00: 00: 00$ | $00: 00: 00$ | 15 s | $00: 00: 15^{\mathrm{a}}$ |
| $00: 00: 10$ | $00: 00: 00$ | 15 s | $00: 00: 16^{(\mathrm{a})}$ |
| $00: 00: 20$ | $00: 00: 00$ | 15 s | $00: 00: 20$ |
| $00: 00: 20$ | $00: 00: 05$ | 15 s | $00: 00: 21^{(\mathrm{a})}$ |
| $00: 00: 25$ | $00: 00: 05$ | 15 s | $00: 00: 25$ |

a. After all other events at that time.

When current time is updated, the Delay node transmits all currently detained events which are greater or equal to the age specified by the Delay Until parameter.

When an event enters the Reset Input port it is immediately transmitted from the Reset output port, or discarded if not connected. All detained events at the Input port are then transmitted in creation time order from the Fail Output port, or discarded if this port is not connected. These events are not old enough to be transmitted from the output port. The Delay node is now in the same state as when the circuit was first initialized.

## Delay Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Delay until | Duration | 0 s | Static |

## Delay until

Delay Until is a statically evaluated expression that determines the length of time that the del ay node detains events entering the Input port. The default value is 0 s .

## CAUTION

The default value of 0 s is a special case that causes the maximum transit delay of the Delay node's Input port to be substituted. The substituted duration is the minimum duration required to cause the Delay node to sort all incoming events into creation-time order.

For example, a Delay Until value of 25 s causes the Delay node to detain incoming events until the difference between the event's creation time and the correlation engine's current time is less than or equal to 25 seconds.

If the duration specified for the Delay Until parameter is less than or equal to the minimum transit delay on the Delay node's Input port (and is not 0 s) then the Delay node will have no effect.

## Delay Node

## Delay Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Reset Input | No | No | PCT |
| Output | Yes | Yes | PCT |
| Fail Output | No | No | PCT |
| Reset Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

## Delay Node Notes

The Delay node does not delay every event for the duration specified by the Delay Until parameter. An event may be detained by the Delay node for any duration between Os and the duration defined by the Delay Until parameter, depending on the creation time of the event and the current time when it arrives at the Delay node.

## Extract Node

The Extract node combines input events with one or more events selected from a Table node's contents attribute to form a composite event, which is immediately transmitted. Configuration parameters define the number and identity of the events combined with the input event.

Figure 3-19 Extract Node


The Extract node can be used in conjunction with a Rearrange node to select and transmit an event stored earlier by a Table node.
Alternatively, the Extract node can be used to combine an event with related events to form a composite so that downstream processing is simplified.

## Extract Node Processing

Each event entering the Input port is combined with one or more events from the Table node's contents attribute to form a new composite event as defined by the Extract Spec and Max Events parameters.

## Extract Node

Figure 3-20 Extract Node Processes


| - | Connection required | $\square$ | Parameter evaluated |
| :---: | :---: | :---: | :---: |
| $\triangleright$ | Connection optional (unconnected o/p discards events) |  | Time window |
| $\checkmark$ | Connection optional (unconnected o/p logs events) | $\stackrel{ }{ }$ - | Evaluation error |
| $\square$ | Parameter ("input") | $\rightarrow$ | Logic flows |
| $\bigcirc$ | Event flows | 1 | Input event |
| $\infty$ | References to node attributes (including Table | $\mathrm{T}_{n}$ | Extracted Table events |

The Extract Spec parameter is an expression, which may reference the input event with the event name input_event. It must return a list of one or more events called the extract list. The members of the list are copies of events in the Table node.
The Max events parameter truncates the extract list returned by the Extract Spec parameter to the specified number of event elements.

A composite event is created from the input event and the extract list.

The created composite event is transmitted immediately from the Output port.

If the number of extracted events is zero, a composite event cannot be created. In this case the input event is immediately transmitted from the Fail Output port, or discarded if the port is not connected.

If there is an error during evaluation of the Extract Spec parameter, the input event is transmitted from the Error Output port, or an error is logged to the engine log file if the port is not connected and logging is enabled.

The created composite event is constructed so that the first component is the event which entered the node at the input port, the second component is the first event in the extract list, the third component is the second event in the extract list, and so on.


The create_time and arrival_time attributes of the created composite event are set to the current engine time. The creation and arrival time attributes of the original and extract events are not changed.
The input event and the events in the list returned by Extract Spec can be any combination of primitive, temporary, or composite events.

## Extract Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Extract Spec | List of Events | none | Dynamic |
| Max Events | Integer | 1 | Static |

## Extract Spec

Max Events

Extract Spec is a dynamically evaluated parameter that selects and returns a list of events. The input event is referred to in the Extract spec expression by using the event name input_event. There is no default value for this parameter.

For example to select all the events from the Table myTable the expression is simply:

```
myTable.contents
```

To select all of the events stored in theTable tableT1 that have the same "ManagedObjectI nstance" as the input event:

```
select ev from ev in tableT1.Contents where
    input_event "managedObjectInstance" =
        ev "managedObjectInstance"
```

Alternatively, to select the most recent event stored in Table tableT2 with the same "ManagedObjectI nstance" as the input event:

```
let
    val (ev, _) = find ev in tableT2.contents where
        input_event "managedObjectInstance" =
        ev "managedObjectInstance"
in
    [ev]
end
```

The Extract Spec expression can be arbitrarily complex, selecting events from a combination of Table nodes or referencing the Data Store, Fact Store, and other nodes' attributes.

Max Events is a statically evaluated parameter that is used to truncate the extract list returned by the Extract Spec parameter to the specified number of elements. The default value is 1 , which means that the created composite will consist of two components-the input event and just the first element in the extract list.

## Extract Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Output | Yes | Yes | C |
| Fail Output | No | No | PCT |
| Error Output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary .

## Extract Node Notes

The Extract node copies selected events from the contents attribute of the Table node. It never deletes them or alters them in any way.
The copied events have their creation and arrival times set to the time they were extracted from the Table node. This ensures that they are not discarded because their transit delay is too large. The input event in the composite output event is the original input event (not a copy).
The Extract Spec expression must always return a list, even when only one event is extracted.

It is more efficient to specify an Extract Spec that creates a list with the required number of events than to rely on Max Events to truncate a long list.

## Filter Node

The Filter node evaluates a bool ean condition for each event that arrives at its Input port, and transmits that event immediately from either the True Output port or the False Output port.

Figure 3-21 Filter Node


The Filter node can be used to transmit only events that match a given condition, or as a true/false event flow branching mechanism in a circuit.

## Filter Node Processing

When an event arrives at its Input port, the Filter node evaluates the condition parameter. The condition can refer to the input event using the event name input_event.

- If the Condition parameter evaluates to the boolean value true, the event is immediately transmitted from the True Output port, or discarded if the port is not connected.
- If the condition parameter evaluates to the boolean value false, the event is transmitted from the False Output port, or discarded if the port is not connected.
- If there is an error evaluating the condition expression, the event is immediately transmitted from the Error Output port, or sent to the error log if the port is not connected and logging is enabled.

Figure 3-22 Filter Node Processes


- Connection required $\square$ Parameter evaluated
- At least one connection required $\quad$ Evaluation error
$\triangleright$ Connection optional (unconnected o/p logs $\square$ Parameter ("input") events) $\rightarrow$ Event flows
-     - Parameter "switches" event to appropriate path $\rightarrow$ Logic flows
© References to node attributes, Data and Fact Stores, and Global Definitions


## Filter Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Condition | Boolean | true | Dynamic |

Condition
Condition is a dynamically evaluated Boolean expression that determines the port from which the event is transmitted. The expression can refer to other node's attributes, the Data and Fact Stores, and surrounding values, as explained in "Expressions" on page 49.

The following simple example passes only events whose

## Filter Node

"perceivedSeverity" is critical (where 1 is critical):

```
input_event "eventInfo.perceivedSeverity" = 1
```

To block events where the "percei vedSeverity" of the event is not critical and an event from the same "managedObjectI nstance" is already present in a Table node named myTable, you might enter:

```
not (
    input_event "eventInfo.perceivedSeverity" != 1 and
        exists table_event in myTable.contents where
            table_event "managedObjectInstance" =
                        input_event "managedObjectInstance"
    )
```


## Filter Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| True Output | No | Yes | PCT |
| False Output | No | No | PCT |
| Error Output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

At least one of the True Output or False Output ports must be connected.

## Modify Node

The M odify node allows event attributes to be added, deleted, or changed.

Figure 3-23 Modify Node


The Modify node should be used instead of the Create node where the required event syntax of the output event is the same as the input event.

NOTE
The M odify node actually creates a copy of the input event, discards the input event, and then modifies the copy. This ensures that references to the input event in other paths through the circuit, or in other circuits, are not affected.

## Modify Node Processing

Figure 3-24 Modify Node Processes


| - | Connection required | $\square$ | Parameter ("input") |
| :--- | :--- | :--- | :--- |
| $-\quad$ Connection optional (unconnected o/p logs events) | $\square$ | Parameter evaluation |  |
| -- | Parameter "switches" event to appropriate path | $\bullet$ | Evaluation error |
| $\rightarrow$ | $\rightarrow$ | Logic flows |  |
| Event flows | References to node attributes, Data and Fact <br> Stores, and Global Definitions | I | Input event |
|  | C | Modified copy of input event |  |

On receipt of an event at its Input port the Modify node creates a copy of the event and evaluates the Modify Spec parameter. The modified copy is then immediately transmitted from the Output port.

Because a copy is created and modified there is no need to be concerned about other references to the input event elsewhere in the circuit. Even the components of composite events can be freely modified. The copy of the event is identical in all respects except for the unique_id attribute and the event's creation time.

Themodify Spec parameter can refer to the input event using the event name input_event.
If there is an error during the evaluation of the Modify Spec then the (unchanged) incoming event (not a copy) is immediately transmitted

## from the Error Output port.

## Modify Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Modify spec | Void or Event | none | Dynamic |

Modify Spec
Modify Spec is a dynamically evaluated expression that allows modification of the input_event. For example:

```
input_event alter ("AlarmCode" => "B")
```

In this example the event has the value of the attribute "AlarmCode" changed to the string "B".
The expression can refer to other node's attributes, the Data and Fact Stores, and surrounding values, as explained in "Expressions" on page 49. It is executed for its effect on the event only and not for its return value. The result of evaluating the Modify Spec parameter is ignored.

## Modify Node Ports

| Port | Connected | Activated | Events $^{\mathbf{a}}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Output | Yes | Yes | PCT |
| Error output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary .

The Error Output port is inactive by default. If it is not connected an error message is written to the engine log file for each discarded event.

## Modify Node Notes

The incoming event is copied before being passed to the Modify Spec parameter and hence has a different unique_id event attribute and creation time. If it is passed through without change and then compared for identity with the original event the comparison returns false despite

Circuit Nodes
Modify Node
the fact that the events are identical in all other attributes.
If a required event attribute is removed from the event it may fail to encode when transmitted from the correlation engine.
In the following example expression, one attribute is modified and two (optional) attributes are removed from the input event.
If the "probableCause" event attribute is greater than 0 , decrement it:

```
let
    val (_, pc) = input_event "eventInfo.probableCause"
in
    if pc > 0 then input_event alter("eventInfo.probableCause"
=>
    ("localValue", pc - 1))
    else
        () -- do nothing, return Void
    end
end
```

Next, remove the "additionalText" event attribute and "additionall nformation" event attribute from the notification:

```
input_event alter(
    "eventInfo.additionalText" => (),
    "eventInfo.additionalInformation" => ()
    );
```


## Rate Node

The Rate node measures the number of events per second that passed through the node during the last interval of a specified duration.

Figure 3-25 Rate Node
Stream of received events are transmitted immediately


The Rate node's Rate and Rate_dict attribute can be referenced by expressions in the parameters of other nodes to trigger an action when a specific value is reached. For example, a Filter node may discard events whenever the rate exceeds 20 events per second.

## Rate Node Processing

Events entering the Input port are immediately transmitted from the Output port or discarded if the Output port is not connected.
The initial value of the Rate attribute is zero.
The initial value of the Rate_dict attribute is an empty dictionary.

## Rate Node

Figure 3-26 Rate Node Processes


The following formula is used to cal culate the rate:

$$
\text { Rate }=\frac{\text { Number of events in input during interval }}{\text { Interval length in seconds }}
$$

The Rate node has two modes of operation: fixed-interval and moving-interval, as described in the following paragraphs.

Fixed-interval mode

The Rate and Rate_dict attributes are calculated periodically, every interval seconds (where interval is the duration in seconds as specified by the Interval parameter). The values of the attributes gives the rate of event input for the last completed interval.
Input events have no immediate effect on the Rate node attributes in fixed interval mode. A count of incoming events is kept and used to calculate the rate at the end of the current interval.

## Moving-interval mode

## Reset action

Clear action

Action occurs only at the end of the current interval. When the end of the interval is reached, the rate for the interval just completed is cal culated using the above formula and all rate attributes are updated to this value.

Rates are recalculated at the end of each second. Only events that arrived within the previous interval seconds, or since the last reset, are considered in calculating the rate.

The rate is also recal culated following the arrival of each event.
If an event is received on the Reset Input port, the Rate attribute and all values in Rate_dict are reset to 0 . No input events received prior to the reset event will be considered in future rate calculations.

If the Rate node is operating in fixed interval mode then time alignment occurs. See Time Alignment.
Events entering the Reset Input are immediately transmitted from the Reset Output, or discarded if the port is not connected.

When an event arrives at the clear Input port, the Rate attribute is reset to 0 and all entries are cleared from the Rate_dict attribute.
Events entering the clear Input are immediately transmitted from the Clear Output, or discarded if the port is not connected.

Rate Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Interval | Duration | none | Static |
| Moving | Boolean | false | Static |
| Time Alignment | (Duration, <br> Duration) | (0s, 0s) | Static |
| Key Spec | Any Simple | None | Dynamic |

## Interval

Interval is a statically evaluated parameter that specifies the duration over which the rate is to be calculated.

The Interval parameter must be equivalent to a whole number of seconds. If it is not then it is truncated to an integral number of seconds.

There is no default value for this parameter.

## Rate Node

| Moving | Moving is a statically evaluated parameter that determines the mode of <br> operation of the Rate node. |
| :--- | :--- |
| The default value of this parameter is false, which selects fixed interval |  |
| mode. To select moving interval mode enter an expression that evaluates |  |

Time Alignment Time Alignment is a statically evaluated parameter that may be used to control the placement of interval boundaries in fixed interval mode. It can be any expression that evaluates to a tuple of two duration data types of the form (period, offset).
This parameter has no effect if the moving-interval mode of operation is selected.

The default value of this parameter is ( $0 \mathrm{~s}, 0 \mathrm{~s}$ ). No time alignment occurs and the first interval begins when the correlation engine is enabled or the Rate node is reset. Thefirst interval ends interval seconds later.

If the Time Alignment parameter is set to any value other than (0s, $0 s$ ) then the first interval ends on the first available time point after the correlation engine is enabled or the Rate node is reset. Subsequent intervals will end every Interval seconds after the first one.

The first available time point is determined as follows:

- When the engine is enabled or the Rate node is reset
- wait until (current time mod period) $=0$
- wait for offset seconds
- perform the initial rate calculation

Or in other words, the next timet, such that ( $\mathrm{t}-$ offset) mod period $=0$

## NOTE

The initial rate calculation assumes that the interval starts Interval seconds before the interval end time but does not consider events received prior to the last time the node was reset.

There are a number of constraints on the values that the Time Alignment tuple can take:

- period must be a positive number. If a real number is specified it is
truncated to an integer. Negative numbers are treated as 0.
- period must be less than or equal to Interval. If this constraint is violated a compilation error is generated.
- offset must be less than or equal to period. Otherwise, offset is set to offset mod period.
The following examples assume the Rate node is started or reset at 02:35:14.

| Time Alignment | Interval | First <br> interval <br> ends | Second <br> interval <br> ends | Third <br> interval <br> ends | Comments |
| :--- | :--- | :--- | :--- | :--- | :--- |
| $(1 \mathrm{~h}, 15 \mathrm{~m} 30 \mathrm{~s})$ | 1 h | $03: 15: 30$ | $04: 15: 30$ | $05: 15: 30$ |  |
| $(15 \mathrm{~m}, 0 \mathrm{~s})$ | 1 h | $02: 45: 00$ | $03: 45: 00$ | $04: 45: 00$ |  |
| $(-1 \mathrm{~m},-1 \mathrm{~s})$ | 1 h | $03: 35: 14$ | $04: 35: 14$ | $05: 45: 14$ | Negative values are <br> treated as 0. |
| $(2 \mathrm{~h}, 0 \mathrm{~s})$ | 1 h |  |  | Compile error. See error <br> log. |  |
| $(2 \mathrm{~m}, 3 \mathrm{~m})$ | 1 h | $02: 37: 00$ | $03: 37: 00$ | $04: 37: 00$ | offset <br> offset peris set to |

## Key Spec

Key Spec is a dynamically evaluated expression that determines the keyed element of the Rate_dict attribute. There is no default value.
When an error occurs during the evaluation of the input event's Key spec parameter, one of the following actions occurs:

- If the Error Output port has been configured, the input event is sent to the Error Output port.
- If the Error Output port has not been configured, the error is logged in the engine log.


## Rate Node

## Rate Node Attributes

| Attribute | Type | Initial value |
| :--- | :--- | :--- |
| Rate | Real | 0.0 |
| Rate_dict | Dictionary | (empty) |

Rate

Rate_dict

The Rate attribute is of data type real. This attribute can be referenced from other nodes by using the syntax nodeName. Rate, where nodeName is the name of a specific Rate node.
For example, to configure a Filter node to pass events through its True Output port if a Rate node called "Traffic" measures a rate less than 20 events per second, you enter the following condition:

```
Traffic.Rate < 20.0
```

Rate_dict is a dictionary containing named rates. Dictionary elements can be referenced by other nodes using dot notation syntax nodeName. Rate_dict key, where nodeName is the name of a specific Rate node and key is a specific key spec value, usually an event attribute value. For example:

```
Traffic.Rate_dict 2.9.3.2.10.4
```


## Rate Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Reset Input | No | No | PCT |
| Output | No | No | PCT |
| Reset Output | No | No | PCT |
| Clear Input | No | No | PCT |
| Clear Output | No | No | PCT |
| Error Output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

## Rate Node Notes

The rate calculation is based on the arrival time of events at the Rate node, not the arrival time of events at the engine.
The moving-interval mode of operation requires more frequent recomputation than the fixed-interval mode and may affect the performance of the ECS circuit if too many moving-interval rate nodes are used.

## Rearrange Node

The Rearrange node receives a composite event and transmits a new event constructed from one or more of the events contained in the original composite event.

Figure 3-27 Rearrange Node
Stream of composite events Stream of rearranged events


Composite events are created as part of the normal operation of the Annotate, Combine, and Extract nodes, and as the error output of the Unless node. It is not possible to transmit a composite event from an ECS circuit.

The Rearrange node is often used to extract a single primitive event from a composite event. The Rearrange node can also be used to produce a new composite event which is a rearrangement of selected component events of the input event.

## Rearrange Processing

The Rearrange node accepts composite events at its Input port. Events of any other type are immediately transmitted from the Error Output port.

## Figure 3-28 Rearrange Node Processes



| - | Connection required | $\square$ | Parameter evaluation |
| :--- | :--- | :--- | :--- |
| $\square$ | Connection optional (unconnected o/p logs events) | $\rightarrow$ | Evaluation error |
| $\square$ | Parameter ("input") | $\rightarrow$ | Event flows |
| -- | Parameter "switches" event to appropriate path | $\rightarrow$ | Logic flows |

The Rearrange node creates a new composite event if the Rearrange spec parameter specifies a composite output event. Otherwise, if a single primitive event is being extracted, the original primitive event is extracted from the composite input event and transmitted from the output port. The composite input event and all component events that are not transferred to the output event are discarded.

If the creation of a new event fails, either because the input event has the wrong format or because the output event is incomplete, then the input event is immediately transmitted from the Error Output port.

Format errors arise if the Rearrange Spec parameter refers to a component of the input event that either does not exist or is a composite event.

Circuit Nodes
Rearrange Node

## Rearrange Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Rearrange Spec | List of Tuples <br> of type Integer | none | Static |

Rearrange Spec Rearrange Spec is a statically evaluated parameter that specifies which component events of the input event to include in the new event, and their position in the new event.

A Rearrange Spec consists of a list of pairs (Tuples) of the form (from_component, to_component).

The from_component is a list of integers that specify the location of the component events to be extracted from the composite input event. The specified events must be either primitive or temporary events, otherwise an error occurs at runtime.
The to_component is a list of integers that identify the positions in the new event of the events specified by the from_component.

## Extracting a single event

The Rearrange node is frequently used to extract a primitive or temporary component from a composite event.
In this case the Rearrange Spec should consist of a list with a single (from_component, to_component) pair. The from_component should specify the location of the desired primitive or temporary event within the input composite event. The to_component should be the empty list [].


For example, to extract the primitive $\mathrm{P}_{3}$, you would write the list expression

```
[ ([2, 1], []) ]
```

The left-hand list $[2,1]$ in the tuple specifies the event $P_{3}$ to be extracted, and the empty right-hand list [] specifies that it is to be an isolated event and not a component of a composite.

Restructuring a The Rearrange node can also be used to build a new composite event composite with a different structure. (Remember that you do not have to use all the components in the new composite.) To do this, the Rearrange Spec parameter maps from each component event in the received composite to a component in the event to be transmitted.

## Rearrange Node



For example, to map the components of composite $\mathrm{C}_{1}$ to the new composite $\mathrm{C}_{2}$, you would write the list expression

```
[ ([4], [1] ),
    ([3], [3] ),
    ([2], [2, 1]),
    ([1], [2, 2]) ]
```


## Rearrange Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Output | Yes | Yes | PCT |
| Error Output | No (logged) | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

If anything other than a composite event arrives at the Input port it is transmitted immediately from the Error Output port.

## Rearrange Node Notes

If the output event is a composite event, then the Rearrange spec must ensurethat all components of the output event are present. For example, it is an error to specify a Rearrange spec such as [([1], [2,2])]. This results in all input events being transmitted from the Error Output port.

Do not connect the Output port directly to a Sink node in an ECS circuit if the Rearrange node is configured to transmit a composite event. Only primitive events can be transmitted from an ECS circuit.

The Error Output port can also transmit composite or temporary events. It is advisable not to connect this port directly to a Sink node of an ECS circuit.

## Sink Node

The Sink node accepts events from inside a circuit through its single input port and transmits them to the enclosing environment. If the enclosing environment is a circuit then all events are transmitted. Otherwise, if the enclosing environment is the external environment then only primitive events are transmitted.

## Figure 3-29 Sink Node



The Sink node must be used to transmit events from an ECS circuit.
A Sink node has a single Input port and no Parameters.

## Sink Node Processing

The Sink node is the path by which a Compound node transmits events to the enclosing circuit. In a top-level circuit the Sink node is the path by which an ECS circuit transmits events to the external environment.

In a Compound node, each Sink node inside the Compound node circuit is visible from the enclosing circuit as an output port of the Compound node, as shown in Figure 3-30.

Figure 3-30 Sink Nodes in a Compound Circuit


When the same circuit is operating as a top-level circuit instead of a Compound node, the output streams from all Sink nodes are merged into a single event stream, as shown in Figure 3-31.

Events that are output from a circuit may still not be output from the engine. Engine output is determined by the engine policy. See "Streams" on page 27 for details.

## Sink Node

Figure 3-31 Sink Nodes in a Top-level Circuit

## ECS circuit

## External environment



## Sink Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

## Sink Node Notes

Only primitive events are transmitted through a Sink node when it is the output port of a top-level ECS circuit. Temporary and composite events are not transmitted from a top-level ECS circuit, although they will pass through a Sink node when it is functioning within a Compound node.

## Source Node

The Source node accepts events from the enclosing environment and transmits them into an ECS circuit through its single output port.

## Figure 3-32 Source Node

Stream of events from outside
circuit or compound node

## Source Node Processing


#### Abstract

Purpose The Source node is the path by which a Compound node receives events from the encl osing circuit. In a top-level circuit the Source node accepts events arriving at the ECS engine, via the associated input port which may be configured to accept only certain events.

In a Compound node, a Source node inside the compound is represented as an input port when the compound is viewed from outside, as shown in Figure 3-33.


## Source Node

Figure 3-33 Source Nodes in a Compound Circuit


In a top-level ECS circuit, the external event stream is available to all of the external input ports. Inside the circuit, each Source node is associated with an input port. Using the External Tab on the ECS Designer, external input ports can be created and configured to accept events of a particular encoding type, event syntax, event type, and/or create time, as shown in Figure 3-34. You can also use a filter condition to filter input events. For more information, see "External Event Filtering" on page 35.
Use the Configure Source dialog box to change the input port name, and to change the Auto Connect setting. When Auto Connect is on, the input port is automatically connected when the node is dropped on an existing connection.

Figure 3-34 Source Nodes in a Top-level Circuit


## Source Node Ports

| Port | Connected | Activated | Events $^{\text {a }}$ |
| :--- | :--- | :--- | :--- |
| Output | Yes | Yes | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary.

The Source node takes events from the input port of an ECS circuit or Compound node, and transmits them from the output port to be processed by downstream nodes. The output port is al ways activated and must be connected.

## Table Node

The Table node maintains a table of attributes or a table of events that have passed through it, and a count of the number of events currently stored in the table. The contents of the table and the current count of events in it is made available to other nodes. The Table node provides short term as well as long term storage of entire events or just some extracted attributes of an event.

Figure 3-35 Table Node
Stream of received events ... are transmitted immediately


## The Table node works in two modes, namely

## - Event Collection mode

When the Table node works in the Event Collection mode, the Table node stores the entire event and a count of the number of events that are currently stored in the node. Events stored by the Table node are saved in a list that is publidy accessible as the node's Contents attribute.

NOTE
The Table node works in the Event Collection mode by default. All explaination hereinafter assumes it works in the Event Collection mode
unless otherwise mentioned.

## - Attribute Collection mode

In the Attribute Collection mode, the Table node stores only the values of extracted attributes of an event and a count of the number of events(whose attributes have been extracted) that are currently stored in the node. Extracted attributes are stored as a list of dictionaries that can be accessed using the node's contents attribute. These attributes are specified in the Attribute List discussed later.

Table node contents can be used to trigger an action when a specific sequence of events has been identified. For example, when four consecutive events from a specific device indicate that its temperature has been steadily increasing.

It is possible to store events indicating the state of a device as it was hours or days ago and extract this information for use in further correlation. Stored events can be extracted into composite events using an Extract node and can re-enter the correlation some time after they initially entered the ECS circuit.

## Table Node Processing

The processing of an event in the Table node depends on the event's age, which is the difference between the current time and the event's creation time. Events entering at the Input port are first filtered according to their age. The filtering compares each event's age with the Save Until parameter.TheTable node processing differs depending on which mode it is working.

## Event Collection mode

If the event's age is less than or equal to the Save Until parameter, referenceto the event is saved(the event itself is not copied and the unique_id is the same for both events) and the event is immediately transmitted from the output port. Otherwise, if the event is too old to store it is immediately transmitted from the Error Output port. If the Error Output port is not connected the event is recorded in the engine log file. If a reference to the event is saved, the event is subjected to further tests depending on the events already stored in the contents attribute.

## Table Node

The Contents attribute is a list of events in creation time order, with younger events at the beginning and older events at the end.
The Contents attribute list is conceptually divided into two regions:

- the current region is limited in size and is based on event age, and
- the retained region can be any size and is controlled by logical expressions in the form of retain and delete conditions.
The division between the two regions is controlled by the values of the parameters Save Until and Max Events to maintain the following conditions:
- All events in the current region are at most Save Until old.
- There are at most Max Events events in the current region.

There are no limits on the age of events or the number of events in the retained region.

## Attribute Collection mode

If the event's age is less than or equal to the Save Until parameter, the extracted attribute values are stored as a list of dictionaries. Othewise, it is immediately transmitted from the Error Output port. If this list is saved, the event is subjected to further tests.
The contents attribute is a list of dictionaries, with each dictionary corresponding to an event stored in extraction time order. It displays a list of dictionaries with each dictionary containing the Attribute values specified while configuration of the Table node.

Figure 3-36 Table Node Processes- I


| - | Connection required | Connection optional (unconnected o/p discards events) | $\square$ |
| :--- | :--- | :--- | :--- | Pvents stored



Events arriving at a Table node go through one or more phases from arrival until they are no longer stored in the Table node (the numbers refer to Figure 3-36 on page 131):

1. arrived: becomes new or is passed through the Error Output port
2. new: becomes current or retiring
3. current: becomes retiring
4. retiring: becomes retained or is deleted
5. retained: is deleted if a retiring event causes it to be deleted

When an event nolonger fits in the current region it is retired. A retiring event is moved to the retained region if the Retain Condition parameter is true or deleted if it is false.

When current time is updated, all events in the current region that have become older than the value of the Save Until parameter are selected for retirement and retired one by one.

An event may be selected for retirement immediately after it arrives at
the Table node, provided that it is young enough, as specified by the save Until parameter.

When an event is retained, all events currently held in the retained region are reconsidered for deletion. For each currently retained event the Delete Condition parameter is evaluated, and if it evaluates to true the event is deleted.

When an event arrives at the Reset Input port the Contents attribute is reset to an empty list and the count attribute is reset to 0 . Events entering the Reset Input port are transmitted immediately from the Reset Output port or discarded if the port is not connected.

## Table Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Save Until | Duration | 0s | Static |
| Max Events | Integer | 0 | Static |
| Retain Condition | Boolean | false | Dynamic |
| Delete Condition | Boolean | true | Dynamic |
| Attribute List | List | None | Static |


#### Abstract

Save Until

Max Events Save Until is a statically evaluated parameter that determines the maximum age of events entering the table, and also determines the longest time an event may stay in the current region of the table. The default value is 0 s. When an event arrives, and at every engine cycle, all events are examined to see if they should be retired because they are now too old.

Retain Condition The Retain Condition is a dynamically evaluated parameter that determines whether a retiring event is deleted or retained. The retiring event is referred to in the condition by using the event name


# Circuit Nodes 

Table Node
current_event. The default value is false, causing no events to be retained.

When an event is selected for retirement, the Retain Condition parameter is evaluated. If it evaluates to the bool ean value true, the event is retained. Otherwise, if it is false or an error occurs, the event is deleted.

Delete Condition The Delete Condition is a dynamically evaluated parameter that determines whether a retained event is deleted or stays retained when another, possibly younger, event is retiring. The retiring and retained events are referred to in the condition with the event names current_event and retained_event respectively. The default value is true, causing retained events to be deleted when another event is retired.

The Delete Condition parameter may depend on both the current (retiring) event and the retained event. This makes it possible to compare the two events. For example, to keep only the most recent temperature reading from a device you would specify that an older event from that device must be deleted when a younger event from the same device is retired.

When an event is selected for retirement and it has been determined that it will be retained, the Delete Condition parameter is evaluated for every event currently in the retained region. If it evaluates to the boolean value false, the retained event stays retained, otherwise the retained event is deleted. Finally, the newly retiring element is inserted in the retained region.

## Attribute List The Attribute List is a statically evaluated parameter that determines the attributes of the event that will be stored in the Table node. Specifying parameters in the Attribute List window enables the Table node to work in the Attribute Collection mode.

The attributes specified in the Attribute List window will beextracted and stored in a dictionary. If there is no list specified in this window the Table node will function in the Event Collection mode, storing the entire event itself.

An empty list [] could be specified in the Attribute list to extract a minimum of the createTime and arrivalTime attributes. Any other attributes specifically listed will also be extracted. For example, to extract the createTime, ArrivalTime, managedObjectClass and event Type, type
["managedObjectClass", "eventType"]
Any attribute being accessed in the Delete Condition or Retain Condition must also be listed in the attribute list. If not, the extracted attributes will not be evaluated with both Delete and Retain conditions, since they are not present in the attribute list.

Table Node Ports

| Port | Connected | Activated | Events $^{\mathbf{a}}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Reset Input | No | No | PCT |
| Output | No | Yes | PCT |
| Error Output | No (logged) | No | PCT |
| Reset Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary .

## Table Node Attributes

| Attribute | Mode | Type | Initial value |
| :--- | :--- | :--- | :--- |
| Contents | Event Collection Mode | List of Events | [] |
|  | Attribute Collection <br> Mode | List of Dictionaries | none |
|  | Event Collection Mode | Integer | 0 |
|  | Attribute Collection <br> Mode | Integer | 0 |

## Contents

The contents attribute is a List data type used to access the list of events stored in the Table node. This attribute can be referenced in other expressions by using the syntax nodeName. Contents, where nodeName is the name of a specific Table node.

For example, to configure a Filter node to pass events only when a previous event has been received with the same
"managedObjectI nstance" (a CMIP event attribute), you enter the following filter condition:

```
exists table_event in TableTemp.Contents where
    table_event "managedObjectInstance" =
        input_event "managedObjectInstance"
```

If the Table node is working in Attribute Collection mode, ensure that the "managedObject Instance" is listed in the Attribute List window if you want to store only this attribute and not the entire event.

## NOTE

The Contents of a table node cannot be accessed from an Extract Node when the table node is operating in the Attribute Collection mode because the <TableNode-Name>. Contents returns a list of dictionaries and not the list of events itself.
Count
Retain the
youngest event

Count is an integer attribute that is the number of events currently stored in the Table node. This attribute can be referenced in other expressions by using the syntax nodeName. Count, where nodeName is the name of a specific Table node. The following identity is always true:
nodeName. Count $=$ length nodeName. Contents

## Table Node Examples

Configure a Table node to al ways retain only the youngest event that has passed through it:

```
Save Until: 24h (long duration)
Max Events: 1
Retain Condition: false
Delete Condition: true
```

Note that in a Table node configured like this, if transit delays cause an older event to arrive at the node after a younger event, the ol der event is retired and immediately deleted.

## Retain the latest event

Configure a Table node to always retain only the latest event that has passed through it:

```
Save Until: 24h (a long duration)
Max Events: 0
Retain Condition: true
```


#### Abstract

\section*{Retain the latest event for a managed object}


## Delete Condition: true

Note that in a Table node configured like this, if transit delays cause an ol der event to arrive at the node after a younger event, the older event will be the one that is selected for retirement and hence it will delete the younger one from the retained region.

Configure a Table node to al ways retain only the latest event for a particular "managedObjectInstance" that has passed through it, from as many different "managedObjectInstance"s as therehappen to be. Set the parameters as follows:

```
Save Until: 24h (a long duration)
Max Events: 0
Retain Condition: true
Delete Condition: current_event "managedObjectInstance" =
retained_event "managedObjectInstance"
```



## Table Node Notes

Time Action

> Save Until The default value for the Save Until parameter is rarely useful. The default value of 0s causes events to be stored only if they arrive at the Table node at the same time as they are created. This is not appropriate behavior for most real circuits. It is therefore nearly always necessary to set Save Until to a suitable duration. To ensure that all events enter event may be due for retirement. Events are retired one by one, and a newly retiring event may cause previously retained events to be deleted even though they were selected for retirement together. However, the order of retirement is not specified and your circuit should not make any assumptions about the order.

# Circuit Nodes 

Table Node
the Table node, set Save Until to a large number (e.g. 100000000s), or use the predefined global definition infinite_duration.

Retain Condition $\quad$| If evaluation of the Retain Condition parameter results in an |
| :--- |
| evaluation error or a type error then the retiring event is deleted and a |
| message is recorded in the engine log file. |

Delete Condition If evaluation of the Delete Condition results in an evaluation error or a type error then the event is deleted and a message is recorded in the engine log file.

Duplication of events

## Retain Condition and Delete Condition

The Table node stores a reference to the original event and does not copy the event itself while functioning in the Event Collection mode. In effect, the same event exists simultaneously in more than one place in the circuit. For example, a downstream node could test the unique_id header attribute of an event that has passed through the Table node against all the events stored in the Table contents to confirm whether the event is in the Table's contents or not.

The Delete Condition and Retain Condition may refer to other nodes' attributes. However, they should not depend on the current Table node's count or contents attributes as the results of doing so are undefined.

There is no way to reference only the current region or the retained region.
The list of events is ordered such that younger events come before older events in the list. However, the ordering of events of the same age is not specified, and your circuit should not make any assumptions about that ordering.

To force all events into the retained region, with the current region effectively disabled, set Save Until very large, Max Events to 0, and Retain Condition to true. Alternatively, to disable the retained region, simply set the Retain Condition to false.

If the Table Node works in the Attribute Collection mode, memory usage is less as only the values of extracted attributes of the events are stored.

## Unless Node

The Unless node transmits input events unless prevented by an appropriate inhibiting event. An inhibiting event is an event that satisfies a given condition, and whose creation time is within a certain window of time before or after the creation time of an input event.

Figure 3-38 Unless Node

Streams of candidate and inhibiting events


The Unless node may be used to block events that are a logical consequence of another event. For example, when a link fails a stream of consequential events may be generated by a device. If a LinkDown event has already been transmitted, the Unless node can prevent transmission of further events about the failure of individual connections on that link.

The Unless node is al so useful for processing events that are inverses of each other such as PowerFail and PowerRestore events, On and Off events, or StartUp and ShutDown events. For example, a transient power failure can be suppressed by transmitting a PowerFail event only if it is not followed by a PowerRestore event within ten seconds.
The Unless node can also be used to filter out "jabber" events created, for example, by consecutive duplicate readings from a device monitoring temperature or voltage levels.

The Unless node detains any input event until no more inhibiting events for that event can arrive at the node. The additional transit delay imposed by the Unless node must be taken into account during design of

## Unless Node

the downstream circuit.

## Figure 3-39 Unless Node Processes



| - | Connection required | $\vdots$ | Events stored |
| :--- | :--- | :--- | :--- |
| $\triangleright$ | Connection optional (unconnected o/p discards events) | $\square$ | Parameter evaluated |
| $\triangleright$ | Connection optional (unconnected o/p logs events) |  | Time window |
| $\square$ | Parameter ("input") | $\rightarrow$ | Evaluation error |
| -- | Parameter "switches" event to appropriate path | $\rightarrow$ | Event flows |
| $\square$ | References to node attributes, Data and Fact Stores, <br> and Global Definitions | $\rightarrow$ | Logic flows |

## Unless Node Processing

The Unless node receives events on its Input port and Inhibitor Input port. An event arriving on the Input port is detained until no further inhibiting events can arrive, before being transmitted. Events arriving at the Inhibit Input port are retained only so long as they could inhibit input events which have not yet arrived. Inhibiting events are never transmitted, except as a component of a composite event on the Error Output port.

The processing of an event in the Unless node depends on the event's age, defined as the difference between the current time and the event's creation time. The processing of pairs of input and inhibiting events depends on the difference between the input event's creation time and the inhibiting event's creation time. That is, how much older or younger the inhibiting event is.

The Unless node considers all possible pairs of input events and inhibiting events that may prevent an input event from being transmitted. All pairs are considered as soon as both events have arrived at the node, without adding any unnecessary delays to the processing of the input events.

When considering each pair of input and inhibiting events, the Unless node compares their creation times as specified by the Window parameter. The Window parameter specifies how much older or younger the inhibiting event may be.

If a pair satisfies the relative age criteria as specified by the Window parameter, the Unless node then evaluates the condition parameter The two events are referred to by the event names input_event and inhibitor_event.


Event pairs considered within the window are: $(1,2.2)$ and $(1,2.3)$

The evaluation of the condition parameter should return a boolean data type. If the evaluation result is true, the input event is transmitted from the Inhibited Output port, and that input event is not considered for any further pair formations. If the evaluation result is false or an evaluation error occurs, the inhibiting event is not capable of inhibiting the input event.

If an evaluation error occurs when considering a pair, a composite event is formed containing the pair of events. The composite event is immediately transmitted from the Error Output port, or logged if the port is not connected. The composite event consists of the input event as the first element and the inhibitor event as the second element. The creation time of the composite event is the creation time of the youngest event, and the arrival time of the composite event is set to the current time.

Figure 3-40 Structure of Unless Node's Composite Error Event


Composite event

To be able to form all possible pairs of input events and inhibiting events, the Unless node may need to detain input events for a while:

1. When an input event arrives it is paired with all inhibiting events currently detained. If any inhibiting event is found, the input event is immediately transmitted from the Inhibited Output port, and is not considered for any further comparisons. If no inhibiting event is found, the input event continues to be detained at the Input port.
2. When an inhibiting event arrives it is paired with all input events currently detained. Any input event that is inhibited by theinhibiting event is immediately transmitted from the Inhibited Output port. Finally, the inhibiting event is detained at the Inhibitor Input port if it is possible that this event may fit in the time window of input events which have not arrived yet.

When current time is updated, the Unless node transmits all detained input events that are now so old that future inhibiting events will be too
young to fit into the specified time window. Also, retained inhibiting events are deleted if they are now so old that they will not fit within the specified time window of any future input event.

An event arriving on the Reset Input port of the Unless node is immediately transmitted from the Reset Output port, if connected. All currently detained events on the Input Port are immediately transmitted from the Fail Output port. These events have not been kept long enough to determine whether they should be inhibited or not. Finally, the Unless node discards any inhibiting events retained at its Inhibitor Input port.

The Unless node is now in the same state as it was when the ECS circuit was first initialized. All events received at the Input port have been transmitted from one of the node's output ports.

## Unless Node Parameters

| Parameter | Type | Default | Evaluation |
| :--- | :--- | :--- | :--- |
| Window | (Duration, Duration) | none | Static |
| Condition | Boolean | true | Dynamic |

## Window

Window is a statically evaluated parameter that must consist of a tuple containing a pair of duration data types. The window parameter determines limits for the age difference between the inhibiting event and the input event, that is:

```
(inhibitor_event create_time) - (input_event create_time)
```

The first element of the pair determines the minimum age difference, the second element determines the maximum age difference. There is no default value. The following examples illustrate the effect of various window sizes:

| Window | To inhibit the input event the inhibitor event must be <br> created: |
| :--- | :--- |
| $(5 \mathrm{~s}, 10 \mathrm{~s})$ | anywhere between five and ten seconds after the input event |
| $(-5 \mathrm{~s}, 5 \mathrm{~s})$ | anywhere between five seconds before and five seconds after <br> the input event |

Circuit Nodes
Unless Node

| Window | To inhibit the input event the inhibitor event must be <br> created: |
| :--- | :--- |
| $(0 \mathrm{~s}, 0 \mathrm{~s})$ | at exactly the same time as the input event |

## Condition

The condition is a dynamically evaluated parameter that determines whether an inhibiting event inhibits an input event. The events are referred to in the condition by using the event names input_event and inhibitor_event respectively. The default value is true, causing any inhibiting event which fits into the time window of an input event to inhibit input events.

When an input event and an inhibiting event pair is selected for evaluation of the condition parameter, the inhibiting event inhibits the input event only if the evaluation of the condition parameter returns true. If the evaluation of the condition parameter fails or returns a value which is not of bool ean type, a composite event is formed from the two events and transmitted from the Error Output port for possible further processing in the ECS circuit.

## Unless Node Ports

| Port | Connected | Activated | Events $^{\mathbf{a}}$ |
| :--- | :--- | :--- | :--- |
| Input | Yes | Yes | PCT |
| Inhibitor Input | Yes | Yes | PCT |
| Reset Input | No | No | PCT |
| Output | No | Yes | PCT |
| Inhibited Output | No | No | PCT |
| Error Output | No (logged) | No | C |
| Fail Output | No | No | PCT |

a. Events are: $\mathrm{P}=$ primitive, $\mathrm{C}=$ composite, $\mathrm{T}=$ temporary .

At least one of the Output port and the Inhibited Output port must be connected.

## Unless Node Examples

Configure an Unless node to pass a "PowerOff" event only if there was no "PowerOn" event created by that same equipment less than ten seconds later than the "PowerOff" event.
Precede the Unless node with Filter nodes to direct the "PowerOff" events to the Input port and the "PowerOn" events to the Inhibitor Input port.

Set the parameters as follows:
Window (0s, 10s)
Condition input_event "managedObjectInstance" = inhibitor_event "managedObjectInstance"

## Unless Node Notes

| Node delay | The Unless node introduces a transit delay because of the window <br> parameter. You should minimize the length of the Window consistent with <br> the correct operation of the node. You can view transit delays through the <br> Unless node in the Node Status dialog when running a simulation. |
| :--- | :--- |
| Event sorting | The Unless node sorts the events on the output port as a side-effect of its <br> processing. Sorting the Input and Inhibitor Input event streams by <br> placing Delay nodes upstream of an Unless node does not affect the <br> processing load in the node. |
| Time action | When current time is updated more than one event may be transmitted <br> from the output port. Events are transmitted one by one. However, the <br> order in which events with the same creation time are transmitted is <br> undefined and your circuit should not make any assumptions about that <br> order. |
| Window | If a Window parameter is specified that will allow inhibiting events to <br> arrive a long time after the input events, the Unless node must retain <br> input events for a long time as well, unless they have already been <br> retained upstream of the Unless node. |
| Efficiency | Place suitable filtering upstream of an Unless node such that only the <br> proper events arrive at the Input and Inhibitor Input ports, and |
| configure a short time window. Otherwise, the amount of processing |  |
| incurred when an event arrives at the node may become excessive as |  |

Circuit Nodes

## Unless Node

pairs must be made with all events already detained or retained on the other port.

NOTE
Do not connect the Error Output port to a Sink node of an ECS circuit. Only primitive events can be transmitted from an ECS circuit.

## 4 Events in ECS

Events are the "electrons" of an ECS circuit. ECS primitive events represent external information of some sort. They can represent network messages of any type (SNMP traps, CMIP notifications, OVO messages, etc.) or even log file entries or database records. Primitive events are described in detail in:

- "Primitive Events" on page 154.

ECS also uses events internally to pass information around a circuit, and to package information (including other events) together. These special event types are described in the following sections:

- "Temporary Events" on page 157
- "Composite E vents" on page 159.

The section on each type of event explains

- what the event is and how it is generated
- the structure of the event and its attributes
- how to read and test the header and body attribute values of an event
- how to modify the body of an event.

The way a primitive event appears in ECS, and in particular the way its attributes are addressed, is dependent on the event encoder/decoder. See the following related publications for details:

- HP OpenView Event Correlation Services ASCII Module
- HP OpenView Event Correlation Services SNMP Module
- HP OpenView Event Correlation Services CMIP M odule

HP OpenView OVO Messages are described in Appendix A, "Event Correlation in OVO," on page 401.

## Introduction

Inside an ECS circuit, information is moved around in the form of events. All events consist of two main parts, as illustrated in Figure 4-1:

Figure 4-1 Structure of an Event

## Body

Specific information carried by the event

Header
The same for all events

| Contents of the event body are <br> dependent on the event_class | ECS attributes: <br> unieque_id <br> event_class <br> create_time <br> arrival_time <br> encoling_type <br> event_syntax <br> event_type |
| :--- | :--- |

- The event header contains information about the event. Header attributes are the same for all events, and describe the event's identity, when it was created, when it arrived at the ECS engine, and how the event is encoded. One of the header attributes, the event_class, determines the class of the event and what the body of the event contains.
- The event body contains information specific to the event's class.

There are three classes of event in ECS:

- Primitiveevents represent an externally defined unit of data, such as a network message, trap or notification. The body of a primitive event consists of the externally defined attribute values.
- Temporary events are generated inside an ECS circuit by the Clock node and the Annotate node, and cannot be transmitted outside the circuit. The body of a temporary event depends on whether it was generated by a Clock or an Annotate node.
- Composite events contain other nested events, and cannot be transmitted outside the circuit. The body of a composite event can contain any number of events nested in a tree structure.


## Event Flow

Events in the ECS engine are moved through the nodes in a circuit by reference. This avoids ambiguity problems and maintains efficiency.
The ECS Engine maintains a list of all current events. When a network message, report, notification or trap arrives at the engine, a primitive event is created and entered in the list. When the event is output from the engine, or if it is suppressed by a correlation circuit, it is del eted from the list. Temporary and composite events are also placed in the list as they are created within a circuit, and removed from the list when they are no longer needed.
Each event is stored just once and the engine ensures that, at most, only one copy of an event is ever instantiated on output to a given stream.

Each event has a uniqueID assigned to it when it is placed in the list. Effectively, it is only the event's unique ID that is passed around the circuit, not the event itself. Because the event is addressed by reference rather than being copied, the exact same event can be processed in many places at once within a circuit. It can even be present simultaneously in different circuits running in different streams.

Problems could arise if an event was changed in one place while being tested in another place, such as in a different circuit. This potential problem is circumvented by the following rules:

- Events are read-only. You cannot change the contents of an existing event.
- The Modify node does not actually modify the input event. Instead, it creates a copy with a different uniqueID, and modifies that. The result is a new event, complete with a new unique ID. This way, any other references to the input event still refer to the original event rather than the modified event.


## Manipulating Events

An event is a subtype of the ECDL Dictionary data type. A Dictionary type is an unordered collection of key-value pairs. To access a dictionary value you need to identify the particular dictionary containing the value you want, and you need a valid key. Given these two items you can retrieve a value from the dictionary.

To identify the dictionary you use one of the predefined event names, such as input_event, as described in "Event names" on page 50. So, if

```
you wanted the event_class header attribute of the input_event you
would write:
```

```
input_event event_class
```

```
input_event event_class
```

This would return one of the three Token values EVPrimitive, EVTemporary or EVComposite, as appropriate for the particular event. You can test to ensure an event is primitive with an expression such as:

```
if (input_event event_class = EVPrimitive)
then
else
end
```

The previous example illustrates standard Dictionary syntax. However, the Event subtype supports a wider range of key addressing schemes than the Dictionary does. These schemes make it possible to address the various classes and parts of events in a natural way that reflects the structure of the underlying data. For example, a composite event contains other events arranged in a tree structure, so a numbering scheme that reflects a tree graph is used.

To manipulate all the data in an event you must know all the keys for that event. There is no way to enumerate the keys or to traverse all the key-value pairs in an event without knowing each and every key. Since there is a potentially infinite number of different events, this begs the question of knowing what type an event is. Usually, a circuit is designed so that low-level comparisons of event attributes are performed only after extensive filtering of the event stream. So, at the point where you need to address particular event attributes, you know exactly what type of event you are dealing with, and what the keys are. This eliminates the possibility of errors raised by attempted access of non-existent or incorrect event attributes keys.

The event header, and the body of each of the three classes of event, has its own scheme for addressing the data contained in it. The remainder of this chapter describes the key addressing schemes used to access the various parts of an event.

## Event Header Attributes

A primitive event consists of a header and a body-see Figure 4-1 on page 149. The header consists of a set of keys that are the same for all events, regardless of their class or type. The header attributes are listed in Table 4-1.

Among other things, the event header attributes identify the structure of the event's body. This means that, by configuring an ECS circuit to filter and route events based on their header attributes, you can be confident that when you delve into the event's body for more interesting information, you know how the event is constructed and consequently how to address the information you want.

In practice you will generally filter on the encoding_type and event_syntax header attributes. For certain events you may also need to filter on the event_type.

The header attributes of an event are key-value pairs, where the key is always a Token data type. Tokens are text values that represent discrete alternatives, similar to an enumerated data type.

You can test and read header values, but you cannot alter them (you cannot use the alter function).
Table 4-1 Event Header Attributes

| Attribute name (Token) | Data type | Value and data type returned |
| :--- | :--- | :--- |
| unique_id | Token | This attribute returns a value that uniquely identifies <br> an event. You can test whether one event is the same <br> as another by comparing their unique_id <br> attributes. |
| event_class | Returns a value of EVPrimitive, <br> EVTemporary or EVComposite. You test this <br> attribute when you need to determine whether an <br> event is a primitive, temporary, or composite event. |  |
| create_time | Time | The time at which the event was created (UTC). |
| arrival_time | Time | The time at which the event arrived at the ECS <br> circuit (UTC). |

## Table 4-1 Event Header Attributes

| Attribute name (Token) | Data type | Value and data type returned |
| :--- | :--- | :--- |
| encoding_type | String | This attribute returns the value "ber", "mdl" or <br> "OpC_Msg". SNMP and CMIP events are always <br> ber-encoded, ASCII events are always mdl-encoded <br> and OVO messages are encoded using the <br> OpC_Msg endecoder.. |
| event_syntax | String | If encoding_type is "ber" then <br> event_syntax identifies the GDMO syntax. For <br> SNMP, the event_syntax is "Trap-PDU". If the <br> encoding_type is "mdl" then it identifies the syntax <br> of a particular message in the MDL source. <br> Otherwise, if encoding_type is "OpC_Msg" then <br> event_syntax is also "OpC_Msg". |
| event_type | Oid for CMIP <br> notifications, <br> otherwise <br> String. | Returns the value "generic-trap" for an <br> SNMP trap, a legal object identifier (Oid) <br> corresponding to the "eventType" attribute for a <br> CMIP notification, a syntax-specific value for <br> ASCII events, or the value of the "MSGTYPE" <br> attribute for OVO messages. |
| data_store | Data Store | Returns the data store corresponding to the content <br> that existed when the event was created. |
| fact_store | Fact Store | Returns the fact store corresponding to the content <br> that existed when the event was created. |
| num_attrs | Integer | Returns the number of attributes in a temporary <br> event. |

## Primitive Events

Primitive events are the internal ECS representations of externally defined network messages, traps, notifications and reports. They are created when an externally generated event arrives at the ECS engine, when a Create node is triggered, or when an event passes through a Modify node in an ECS circuit.

Primitive nodes are output from the circuit through a Sink node. Primitive events are the only events that can be transmitted from a circuit (temporary and composite events can exist only inside a circuit).

## See also

Overview

Body attributes

- HP OpenView Event Correlation Services CMIP M odule
- HP OpenView Event Correlation Services ASCII Module
- HP OpenView Event Correlation Services SNMP Module
- "Create Node" on page 87
- "M odify Node" on page 105


## Structure of Primitive Events

A primitive event consists of a header and a body-see Figure 4-1 on page 149. The header contains attributes required for internal housekeeping as the event travels through an ECS circuit, and the body consists of one or more attributes.

The exact details of how the attributes of a primitive event are addressed is dependent on the event endecoder. The following description illustrates the general idea. Refer to the appropriate module guide (as listed in the Seal so list above) for specific details.

Primitive event attribute keys are String data types. The names themselves are taken from the external event description:

- "ber" encoded events use the GDMO/ASN. 1 metadata
- "mdl" encoded events use the MDL metadata
- "Opc_Msg" messages use the OVO message attribute keys listed in Appendix A, "Event Correlation in OVO," on page 401.

|  | Working with Event Attributes |  |
| :---: | :---: | :---: |
| Reading and testing attributes | To identify the event whose attributes you want to read, you use one of the event names defined for that particular node parameter (see "Event names" on page 50). For example, to access the event entering a Filter node you would use the event name input_event. |  |
|  | The event name is followed by the name of the event attribute you want to test. Body attribute keys for primitive events are always of String data type. So to filter on the attribute "messageType" you would enter: |  |
|  | N ote the double-quotes around the attribute key—primitive event body attributes keys are always Strings. |  |
| Structured attributes | Some event encodings support structures such as sequences and sets. In general these are supported as described in Table 4-2. Refer to the appropriate module guide for details. |  |
| Table 4-2 | Attribute Names |  |
| Structure | Naming notation | Example Event Attribute Name |
| Simple attribute | Field name | "managedObject Instance" |
| SEQUENCE, SET or subtype | Higher-level attribute name, followed by a '.', followed by the member name | "eventInfo.probableCause" |
| SEQUENCE OF or SET OF, or list. | Higher-level attribute name, followed by a number in square brackets, representing an index into the list, with the first element starting at 0 . | "managedObject Instance [0] " |
|  | For lists that contain other lists, multiple bracketed indices may be specified, separated by a dot $\because$. | "managedObject Instance[0].[0]" |

## Altering attributes To set an attribute value you use the alter function in a Modify or Create node.

For example, to change the "severity" value in an ASCII event to the

## Primitive Events

value 2 , you would write the following expression in the parameter of a Modify node:

```
input_event alter ("severity" => 2)
```


## Creating a new primitive event

When you create a new primitive event using the Create node, ECS generates a new event and sets the header attributes. The body of the event must be specified in the Create spec parameter using the alter function to set each attribute value in the created event.

For example, the Create spec for an ASCII event with the SimpleEvent syntax might be:

```
let
    val t = explode_time (Time.now ())
in
    created_event alter
    (
        "createTime.date.year" => t Time.year,
        "createTime.date.month" => t Time.month,
        "createTime.date.day" => t Time.day,
        "createTime.time.hour" => t Time.hours,
        "createTime.time.minute" => t Time.minutes,
        "createTime.time.second" => t Time.seconds,
        "deviceId" => "Ephor",
        "messageType" => "LNKUP",
        "severity" => 2,
        "text" => ""
    )
end
```

Values must be supplied for all non-optional body attributes, including values such as "createTime" (above) from which the create_time header attribute is derived. The derivation of the create_time header attribute works in one direction only; even though a new event has the current time placed in the create_time header attribute, this does not automatically set the body attributes from which create_time is derived. In this example, Time. now returns a Time data type value, which is converted to a dictionary named $t$ by the built-in function explode_time. The exploded time values are extracted from this dictionary with expressions such as t Time.year.

## Temporary E vents

Temporary events are created within an ECS circuit and cannot be transmitted from the circuit. They are used to transport information between nodes, or to trigger other nodes into action.

## How temporary events are produced

Temporary events are created by:

- An Annotate node when it receives a response from an external annotation server. The temporary event contains the data returned from the annotation server and is transmitted as a component of the composite event created by the Annotate node.
- A Clock node on each tick of the clock.


## See also • "Annotate Node" on page 58

- "Clock Node" on page 64


## Structure of Temporary Events

Overview The header of a temporary event always has an event_class attribute value of EVTemporary. The contents of the body of a temporary event depends on whether the event was created by a Clock node or an Annotate node:

- Temporary events created by an Annotate node are embedded within the composite event created when the annotation server responds. The temporary event consists of attributes with keys of type Integer. Keys are numbered sequentially from 1 (one) where each number corresponds to an item in the response List. See Figure 4-2 on page 158.
- Temporary events created by a Clock node have an empty body.


## Temporary Events

Figure 4-2
Temporary Event Created by an Annotate Node


## Composite Events

What composite events are
How composite
events are
produced

See also - "Annotate Node" on page 58

Reading and testing composites' attributes

- "Combine Node" on page 71
- "Extract Node" on page 97
- "Rearrange Node" on page 116
- "Unless Node" on page 139


## Working with Attributes and Components

Composite events are combinations of events packaged together so that they can be treated as a single event. The component events can be primitive, temporary, or composite events.

Composite events avoid the overhead of dealing with related events separately. For example, you might package several events together so that a Create node can extract information from several events to include in a single new event.

This mechanism is fundamental to the ability of ECS to concentrate information taken from the event stream over time, allowing fewer delivered events to have a higher information content.

The Annotate, Combine, and Extract nodes create composite events as their normal output. The Unless node creates composite events only as an error output. The Rearrange node can create new composite events out of the components of a received composite, but is most often used to extract a primitive or temporary event out of a composite.

Composite events exist only within an ECS circuit and cannot be transmitted from a circuit.

The header of a composite event always has an event_class attribute value of EVComposite. The body of a composite event is comprised of other events. To obtain the value of an attribute from a contained event you must specify the path the event and the key.

For example, to decode composite events with the structure shown in the following diagram, you may want a Filter node to pass a composite $\mathrm{C}_{1}$ if

## Events in ECS

## Composite Events

the primitive component $P_{4}$ on path 12 has a "messageType" attribute with the value "LnKDN":


You specify the path to the component event whose attribute you want to test using a series of numbers. Each number represents the nth event at the current level in the tree. Starting from $\mathrm{C}_{1}$, the path we want goes through $\mathrm{C}_{2}$, which is path $\mathbf{1}$. From $\mathrm{C}_{2}$ we want to access $\mathrm{P}_{4}$, which is the second event at the next level, so we append the number $\mathbf{2}$ to the path to form the complete path $1 \mathbf{2}$. To test the "messageType" attribute, you would use the following expression in the Condition parameter:
input_event 12 "messageType" = "LNKDN"


#### Abstract

Modifying components' attributes

The Modify node duplicates an event and allows the attributes of the new event to be altered with the alter function. For example, in the composite $\mathrm{C}_{1}$ (below) you may want to change a value in the event $\mathrm{P}_{5}$ :


## Extracting a primitive or temporary



To change the value of the "severity" attribute of the composite event $\mathrm{P}_{5}$ to 9 , you would write the following expression:
(input_event 3 2) alter ("severity" => 9)
The Rearrange node is frequently used to extract a primitive or temporary component from a composite event. For example, from the composite event $\mathrm{C}_{1}$ (below) you might want to extract the primitive event $P_{4}$, which is a component of the composite component $C_{2}$.


To extract the primitive $\mathrm{P}_{4}$, for the Rearrange Spec parameter you

## Events in ECS

## Composite Events

would write the following expression:

$$
[([2,1],[])]
$$

The Rearrange Spec parameter defines a List of Tuples. In this case there is just one Tuple. E ach Tuple encloses a pair of Lists. The left-hand list ( $[3,2]$ ) specifies the event to be extracted, and the empty right-hand list [] specifies its destination is to be an isolated event and not a component of a composite.

Rearranging composite events

The Rearrange node can also be used to build a new composite event with a different structure. You can use as many or few components as required in the new composite.

For example, given an event with the structure of composite event $\mathrm{C}_{1}$ (below). You might want to build the new composite $\mathrm{C}_{2}$ containing all the components of composite $\mathrm{C}_{1}$ in a different structure.


To map the components of composite $\mathrm{C}_{1}$ to the new composite $\mathrm{C}_{2}$, you would write the Rearrange Spec parameter as a List containing four Tuples:

```
[ ([4], [1] ),
    ([3], [3] ),
    ([2], [2, 1]),
    ([1], [2, 2]) ]
```

The left-hand list in each Tuple specifies the path to the source
component in composite $\mathrm{C}_{1}$, and the right-hand list specifies the path to the destination component in composite $\mathrm{C}_{2}$.

Events in ECS
Composite Events

## 5 Timing Considerations

ECS correlates events in real time. However, an event may take a substantial amount of time to travel from the device that generated it to the ECS Engine that will process it. Events can even arrive at a correlation engine in a different order to that in which they were created.
A correctly designed circuit can compensate for network delays that would otherwise disrupt the recognition of event patterns. This chapter provides essential background on the mechanics of time and event delay management within the ECS engine, and the timing considerations you need to take into account when designing a circuit.
The chapter covers:

- "Time Synchronization" on page 167
- "Times and Durations" on page 168
- "Current Time" on page 169
- "Event Time Attributes" on page 170
- "Transit Delays" on page 171

NOTE
To compensate for network delays, events must be stamped with an accurate creation time. If creation time is not known (such as with SNMP traps) then ECS substitutes the time the event arrived at the ECS Engine, and the circuit loses its ability to compensate for network delays.

## Time Synchronization

Many correlation problems depend on the timing of the events being compared. For example, receipt of a link-down event may be relatively unimportant if it is followed soon after by a link-up event. The problem is that network and other delays can separate events in time. This can even result in a link-up event arriving beforethe corresponding link-down.

Because the notion of time is central to the concept of correlation, some form of synchronization is essential, and that requires a knowledge of the time an event was created. Unfortunately, many factors combine to reduce the reliability of event creation times. For example:

- inaccurate clocks that gain or lose time,
- lack of synchronization between clocks,
- use of different time standards on different clocks,
- the complications of daylight saving time when local time is in use.

These problems are further compounded by devices that create events without any creation time stamp at all, and major differences between protocols in the way creation time is defined, if it is defined at all.
HP OpenView ECS expects events to be stamped with the time they were created using Coordinated Universal Time (UTC). Since this is frequently not the case, creation time is approximated by setting it to the arrival time of the event on the following occasions:

- When the "eventTime" attribute of a CMIP notification is absent.
- For all SNMP events.
- For ASCII events, if a create_time part is not specified in the event description and the user-provided event I/O process does not specify a create_time. See the HP OpenView ECS Devel oper's Guide and Reference
- When the creation time is deemed incorrect by the correl ation engine. For example, when creation time is later than the time the event arrives at the engine.


## Times and Durations

Times and durations in ECS are measured with a precision of microseconds. Time values and functions work correctly during and beyond the year 2000.

Time Time is implemented using the Time data type. It is assumed that all times are in Coordinated Universal Time (UTC). Both the create_time and arrival_time event header attributes are Time data types. The body of an event may contain additional Time values.

Duration

See also

Relative times in the correlation engine are implemented using the Duration data type. Transit delays and parameters such as time windows, intervals, and time-outs are implemented as durations.

- "Duration Data Type" on page 216
- "Time Data Type" on page 217


## Current Time

In the ECS Designer in simulation mode, current time is manipulated in a variety of ways so that you can see the passage of events through the circuit. In a running engine, however, current time is derived from the host machine. Current time always runs forwards. Even in simulation mode you cannot reverse the direction of time.

| Resolution | The current time is incremented in one second steps. Although the <br> engine is capable of calculating time values down to the microsecond, the |
| :--- | :--- |
| resolution of "current time" is limited to an integral number of seconds. |  |

Simulation current The current time of the correlation engine is displayed in the top-left time corner of the ECS Designer circuit window while a simulation is running.

In simulation mode, current time is initialized and manipulated by the event log file. See "Simulating Time in the ECS Designer" on page 173.
You can then step through the time line established by the event log by clicking the appropriate Step By button: [Activity], [Event], or [Time].

## Event Time Attributes

The ECS correlation engine uses the create_time and arrival_time event header attributes to manipulate an event while it is in the engine.

## Creation Time

The create_time event header attribute is an absolute Coordinated Universal Time (UTC) value representing the time at which the event was created.

Where an accurate event creation time is available it is used, otherwise it is approximated. By default, create_time is set to the time at which the event arrives at the ECS engine. However, this is overridden if a more accurate creation time is available:

- If the message itself carries a creation time attribute then this value may be used to set create_time (e.g. CMIP notifications and ASCII events where a create_time part is specified in the MDL message description).
- If creation time information is available from some other source then events delivered through the ECS Event I/O API can set the create_time parameter to EIO_sendEvent(3).

See the HP OpenView ECS Devel oper's Guide and Reference for further information on setting the create_time attribute in the event I/O process.

## Arrival Time

The arrival_time event header attribute is set to the current time when the event arrives at the ECS correlation engine.

## Transit Delays

Event transit delay is the difference between current time and the event's creation time. An understanding of transit delays will assist you to design circuits that are efficient in their use of memory, and that add the minimum additional delay to transit times.

As a circuit designer, you are interested in two aspects of event transit delay:

- External transit delays that occur prior to an event's arrival at the ECS correlation engine.
- Internal transit delays that occur inside an ECS circuit, as introduced by the circuit designer.

The circuit designer must deliberately introduce transit delay in order to correlate events that may already have been delayed and may therefore arrive out of time order. For example, the Delay node is often used to ensure that events arriving within a certain time window are in creation time order. It does this by delaying events until they are a certain age.
A circuit may also introduce delay for other reasons. For example, an U nless node may be used to pass a PowerOff event unless a PowerOn event occurs within 30 seconds. In this case PowerOff events transmitted by the circuit will always be at least 30 seconds old because the Unless node cannot transmit the PowerOff event until it is sure that a PowerOn event has not been generated.

When both external and internal delay is present, the designer must know the maximum and minimum delays that events are subject to when they arrive at the engine. Given this information the engine can propagate the delay values through the circuit.
The External tab on the ECS Designer window lists all of the circuit's input ports and allows a transit delay window to be specified for each one. When a circuit contains nodes that store events for a time based on the event's creation time (Delay, Unless, Annotate, Table, or Combine nodes) then you must specify values for both the maximum and the minimum transit delay for each port. If the circuit does not contain nodes that store events then there is no need to specify transit delays.

If you do not know what the minimum transit delay is, then 0 (zero) should be substituted. If you do not know what the maximum transit

## Event bypass

Performance
delay is then you must make a reasonable estimate and you may need to provide additional circuitry for events that take longer than this.

If a late event arrives after the maximum transit delay that you specify then its handling is determined by the stream policy, unless there is another (more broadly specified) input port that it can enter. In other words, events that are older than the allowed maximum may bypass the circuit. This behavior occurs if the stream policy is "Output" (the default). If you need to suppress such events then you have two options:

- change the stream policy to "Discard" (ecsmgr -stream stream -policy discard)
- provide a parallel circuit that provides a route for events that are older than the maximum specified for the primary circuit(s).

In both the external network and the ECS circuit, transit delays can only be increased. The ECS engine processes events in real time and it is no more possible to undo a transit delay than it is to go backwards in time. The longer the transit delay, the longer events must be held in memory awaiting the possible arrival of related events. In extreme situations the additional memory requirements can impact the performance of the engine.

Sometimes, you need to know the maximum and minimum transit delay that events are subject to at a particular node in the circuit. You can do this in simulation mode by clicking the Statistics tab on the Node Status dialog box.

## NOTE

Events that have their create_time attribute set to the time the event arrives at the engine, as described in "Time Synchronization" on page 167, always have a transit del ay of zero. If you set a minimum transit delay of anything other than 0 (zero), your circuit will not receive any of these events.

## Simulating Time in the ECS Designer

When you run an ECS circuit in the ECS Designer in simulate mode, the time at which the engine is started and the arrival times of events are controlled by the event log. In addition to events, the event log contains additional control information that the Designer uses to control its notion of time.

The ECS event log file can be created using the ECS engine to capture a stream of real events off the network and write them to a file. See the HP OpenView ECS Designer's Guide for details on how to capture an event log file. However, you may need to add or modify information in this event log file to manipulate time. This is easily done using a conventional text editor.
A typical output event log, as captured by the ECS Engine, is shown in the following example:

```
# eventid(0:1)
+0
!1
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:
# eventid(0:2)
+0
!1
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:
# eventid(0:3)
+0
!1
1997/01/03 12:05:00,Vili,PWRUP,5,"Power Up"
% mdl:SimpleEvent:
```

Note that time control lines are all +0 , and that there is no transit delay (there is no number specified at the end of the \% lines). Before using such a file in the ECS Designer for simulation, you may need to modify the event arrival times. You may also need to add transit delays. The following discussion describes how to specify these, and how the engine interprets time.

Initial engine time When specifying event arrival times it is easiest to think in terms of "engine time". If you use the log file above in the ECS Designer (by selecting Simulate:Load Input Events from the menu in simulate

## Simulating Time in the ECS Designer

mode) then the initial engine time is set to the create_time of the first event in the event log. In this example, if the create_time header attribute is derived from the event itself then the initial engine time (and the first event's arrival time) is 12:03:00 on 3rd J anuary 1997.

The initial engine time is set for specific kinds of events as follows:

- For OVO messages, the time is set to the creation time of the first message in the event log (as indicated by the CREATION_TIME attribute).
- ASCII events (events with an encoding_type of "mdl") may or may not have a create_time part defined, depending on the MDL message description. If a create_time part is present in the MDL message description then the time in the first event is interpreted as local time, converted to UTC, and used to set the engine's initial time. If a create_time part is not present, the engine's initial time is set to the current real time (UTC). See HP OpenView Event Correlation Services ASCII Module Guide for details.
- If the first event in the log is a CMIP notification with a valid "eventTime" attribute, then this time is used to set the engine.
- If the first event in the log is an SNMP Trap, or a valid create_time cannot be derived from the first event, then the engine start time is set to the start of the current day (00:00:00). (SNMP v1 does not define a creation time attribute, only a "time-stamp" that counts ticks since the last device reset.)


## NOTE

If a time zone or UTC is not specified for the event's create_time, then local time is assumed. Since the ECS Engine (and the ECS Designer) work in UTC, the time displayed in the Input Events and Output Events windows are always in UTC.

Adjusting arrival times

The create_times of events after the first have no effect on the engine time during simulation. If you run a simulation using an event log that does not specify any time increments, then all events arrive at the engine at the same time-the create_time of the first event.

To control when events arrive at the engine in the simulator, you must add information to the raw engine output event log. The first task is to manipulate arrival times by adding lines of the form +119 to the event log. This increments engine time by 119 seconds before inputting the

```
event. For example:
# eventid(0:1)
+0
!1
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:
# eventid(0:2)
+2
!1
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:
# eventid(0:3)
+119
!1
1997/01/03 12:05:00,Vili,PWRUP,5,"Power Up"
% mdl:SimpleEvent:
The second event now arrives at 12:03:00 + 2 seconds (12:03:02) and the third event arrives at 12:03:02 + 119 seconds (12:05:01). N ote that unless you increment the time explicitly with \(+n\) directives, the engine time remains the same forever.
```


## Setting transit delays

```
You can also adjust the create_time of an event relative to its arrival time by specifying a "transit delay" value. To do this you add a number representing the seconds of transit delay to the end of the event syntax line (the lines starting with \% ). Continuing the example, to bring the creation times of the events in the simulator into line with the time stamps on the events, we add the following transit delays:
```

```
# eventid(0:1)
```


# eventid(0:1)

+0
+0
!1
!1
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:0
% mdl:SimpleEvent:0

# eventid(0:2)

# eventid(0:2)

+2
+2
!1
!1
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:1
% mdl:SimpleEvent:1

# eventid(0:3)

# eventid(0:3)

+119
+119
!1
!1
1997/01/03 12:05:00,Vili,PWRUP,5,"Power Up"
1997/01/03 12:05:00,Vili,PWRUP,5,"Power Up"
% mdl:SimpleEvent:1

```
% mdl:SimpleEvent:1
```

The first event has no transit delay specified. The second event has a

## Timing Considerations <br> Simulating Time in the ECS Designer

transit delay of 1 second (as indicated by \% mdl:SimpleEvent:1). The ECS Designer subtracts one second from the arrival time and deems the create_time of this event to be 12:03:02-1 second (12:03:01).
The third event also has a transit delay of 1 second and so the create_time is deemed to be 12:05:01-1 second (12:05:00).

If a transit delay is specified for the first event then the arrival time of the first event (which is derived from the create_time attribute of the first event) is incremented by this amount and the engine time is set to the computed arrival time.

## Engine Input Event Logs

A typical input event log, as captured by the ECS Engine, is shown in the following example:

```
# eventid(0:1)
+0
!1
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:49
# eventid(0:2)
+1
!1
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:48
# eventid(0:3)
+119
!1
1997/01/03 12:05:00,Vili,PWRUP,5,"Power Up"
% mdl:SimpleEvent:29
```

Note that time control lines (starting with +) increment the simulator's clock and that transit delays are specified (49 seconds for the first event).

## Order of Processing

The order of processing within an engine step is not defined. This is generally not a problem. However, when you need to refer to an attri bute of another node, such as the value of a Count node's Count attribute, you must be careful to arrange the nodes in series rather than in parallel, in order to avoid undefined results.

For example, consider the following simple circuit that is intended to transmit only the 10th event that appears at the input 1 node:

## A poorly designed

 circuit

Incoming events are transmitted to both the Count node and the Filter node. Conceptually, the same event arrives simultaneously at both these nodes.

The Count node is incremented by each event, and the Filter node Condition parameter passes the event to the output1 node only if the count is exactly 10. The problem is that you cannot rely on one node being evaluated before the other. Consequently, the Filter node might pass either the 10th or the 11th event or it might not pass any events at all, depending on which node is evaluated first.

On the other hand, the next circuit avoids this problem by processing events serially instead of in parallel:

## Timing Considerations

## Order of Processing

## A well designed

 circuit

## Engine Time Management

This section explains how time is managed by the engine when it is running. The way in which the engine steps through time is described, followed by an explanation of the processing done in each of the two phases within a step.

The ECS correlation engine sees time change in one second increments. Unless it is overloaded, the engine has a full second in which to process all the events that were in the circuit at the start of that second. Time is then incremented by one second, and the process repeated. The engine proceeds in this step-wise fashion for as long as it is running.

Within each step there are two distinct processing phases:

- The input phase
- The cleanup phase.


## Input Phase

The input phase processes all events entering the engine. It starts at the beginning of a second and ends when all events have been processed through the circuit.

The input phase can take as long as necessary. It does not have to complete before the start of the next second.

When the input phase is complete the cleanup phase is started.

## Cleanup Phase

The cleanup phase starts immediately after the input phase. The main purpose of the cleanup phase is to perform time-based actions in the engine, such as releasing an event from a Delay node when the event is sufficiently old.

A cleanup phase is executed for each second the engine is running, regardless of whether an input phase was executed for that second.

During the cleanup phase, nodes that retain events for a given time period are cleaned up. For example, Delay, Unless, Combine and Table nodes all retain events, but Filter, M odify, Create and Extract nodes do not.

## Timing Considerations

Engine Time Management

## Input Phase Overload Processing

If a step lasts longer than one second, input phases are skipped until the engine catches up. A cleanup phase is never skipped no matter how loaded the engine becomes because this can adversely affect the order that events are delivered to nodes.

The following time line illustrates input and cleanup phase processing in a fully loaded engine (that is, in an engine which, for three seconds, has no free time).

Figure 5-1 Input and Cleanup Phases in a Fully Loaded Engine


The subscripts on the I nput $\left(I_{n}\right)$ and Cleanup $\left(C_{n}\right)$ phases in Figure 5-1 represents the engine time for work performed. Notice that $I_{28}$ is skipped altogether and $\mathrm{C}_{28}$ does not occur until the 29th second. This means that for a heavily loaded engine, engine time may lag behind real time.

Where long transit delays could be introduced by this process it may be worthwhile adding a second path to the circuit, specifically for events that have been delayed. The second path would do less processing than the primary path, but would ensure that events were not lost.

A second path might, for example, simply create a new event for every 100 input events, effectively dividing the downstream event rate by 100.

## 6 <br> Data Store and Fact Store

Networks are subject to constant change. Equipment is replaced, additional equipment is installed, and new backbones and management facilities are added. However, the event patterns recognized by your correlation circuits change much less frequently and for different reasons.

The Data Store and Fact Store allow you to separate the environmental aspects of event correlation from the basic logic and rules that are "hard-coded" into a circuit. The Data Store and Fact Store let you:

- Build data-driven ECS circuits customized by external parameters.
- Separate the task of ECS circuit design from the tasks of installing and administering circuits.
- Install the same ECS circuit in different nodes of a network by redefining local topology relationships and operating parameters.
Data can be accessed from Data Stores and Fact Stores in either the current context or the event contexts. This is discussed in detail in:
- "Data Store and Fact Store Contexts" on page 183.

The Data Store holds ECS circuit and external network details such as intervals, threshold values, and other constants. It is discussed in detail in:

- "Data Store" on page 186.

The Fact Store holds details about relationships such as the network topology. It is discussed in detail in:

- "Fact Store" on page 193.

Multiple Data Stores and Fact Stores can be loaded into an ECS Engine. This is discussed in detail in:

- "Multiple Data and Fact Stores" on page 200.

It is possible to simulate event contexts when using the ECS Designer in Simulate mode. This is discussed in detail in:

- "Simulating Event Contexts in the ECS Designer" on page 201.

In some circumstances the Annotate node may provide a better mechanism for accessing volatile data than the Fact and Data Stores. See "Annotate Node" on page 58 for details.

## Data Store and Fact Store Contexts

## Overview

A Data or Fact Store entry is accessed as either the most recently updated version of an entry (using the current context) or the version of the entry that existed when the event was created (using the event context). ECDL operations are provided to access the required context.

Two logical contexts are supported:

- The current context, comprising the collection of the most recently updated entries in the Data and Fact Stores.
- The event context, for an event currently existing in the correlation circuit, comprising the collection of Data and Fact Store entries that were current when the event was created. That is, the event context of an event is the current context that existed when the event was created. An event context is a property (an attribute) of an event. This means it is only accessible where the event is accessible.


## Contexts and Multiple Circuits

There can be any number of Data Stores and Fact Stores loaded into a correlation engine, and there can be any number of correlation circuits loaded into the engine. A correlation circuit can associate with any one Data Store and any one Fact Store without restriction. Any Store can be shared between any number of correlation circuits.
Also, an event can enter any number of correlation circuits. The same current context of a store is visible in the scope of all correlation circuits that are associated with that store. It can be considered to be a global value for the entire engine, accessible from any ECDL expression in any correlation circuit that is associated with that store.

The event context of a store is visible in all circuits where the event is visible, because it is an attribute of the event. An event context is a snapshot of history and is immutable. It is conceptually a universal read-only value. In this sense, an event context can be considered to be global to the engine, but only accessible by expressions configured for the dynamic parameters of any correlation node (in any correlation circuit) through which it is processed.

Data Store and Fact Store<br>Data Store and Fact Store Contexts

## Event Creation

The read-only create_time event header attribute is used to associate the appropriate event context values with the data_Store and fact_Store attributes when an event is received by the correlation engine or created by a correlation node. Correlation nodes variously create primitive, composite, and temporary events, setting the create_time attribute and associating context values with the data_Store and fact_Store attributes.

A summary of how event contexts are assigned when correlation nodes create new events is shown in Table 6-1. This table does not include event creation as a result of run-time errors.

Table 6-1 Context Assignment for Events Created by Correlation Nodes

| Node Type | Event Type | Source of Event Context |
| :--- | :--- | :--- |
| Annotate | composite | create_time equals engine time, so the current context is <br> used as the event context. |
| Clock | temporary | create_time equals engine time, so the current context is <br> used as the event context. |
| Combine | primitive | create_time set to that of youngest component, so the <br> created event inherits the event context of the youngest <br> component. |
| Create | create_time equals engine time, so the current context is <br> used as the event context. |  |
| Extract | primitive | The create_time of the composite event and the copies <br> of extracted events are equal to engine time, so the current <br> context is used as the event context for the composite event <br> and extracted components. |
| Modify | primitive <br> composite <br> The input event is copied, so the created event uses the same <br> Rearrange | primitive: no new event is created. <br> composite: create_time equals engine time, so the <br> current context is used as the event context. |

## Store Versions and Transit Delays

For either a Data Store or a Fact Store, a new (logical) version is created whenever an update is made to the Store. The latest version is by definition the current context. The previous version of the Store will remain in existence as long as an event is using it as a context, or it may be held as a potential event context if an event may yet arrive to claim it (determined from circuit transit delays).

When a Store version is deleted, it is no longer available for association with any new events, and any entries that were unique to the particular version are deleted from the engine.

For a circuit without a maximum transit delay configured, a new event may arrive after its true context has been deleted. The event will be assigned the youngest context that is older than the event's creation time; otherwise the oldest context available will be assigned. This can lead to timing inaccuracies, so it is recommended that transit delays al ways be used to avoid the problem.

Loading and unloading of circuits with and without configured transit delays will cause dynamic adjustment of the rules governing context deletion.

Table nodes present special problems. An event may be held in a table node indefinitely. Since every event may have an event context for each Store, this implies that the associated event context may exist indefinitely. This context will be available to any event which is subsequently accepted by the engine.

An event retained by a Table node consumes resources, including the resources consumed by the Store versions associated with contexts. In the worst case, where all entries in the Data and Fact Stores referenced by a circuit change between events, every retained event will have unique Store versions. If the Stores are very large, system memory requirements may become excessive. ECS imposes no limits on the size of a Store or the number of versions that can exist.

## Data Store

This section describes the format and use of the Data Store, a data structure that contains key-value pairs. These key-value pairs can be referenced using the dataStore function anywhere a value is required.

The Data Store is held in memory for fast access. It is loaded and updated from Data Store files that define a series of actions (add or delete) and key-value pairs.

The ECS correlation engine loads the Data Store from the Data Store file when the ecsmgr command is run. In the case of the ECS Engine simulator in the ECS Designer, the Data Store is loaded when the running of events is started. A Data Store can be modified while the ECS correlation engine is running, subject to some restrictions.

## Generating Data Store files

When designing circuits, you can create Data Store files with any text editor. But in a production environment, you will probably program applications or shell scripts to generate these files automatically from your network database. Other sources of data could be the MIBs of the managed objects in your network and the GDMO metadata of HP OpenView DM.

The file suffix must be *. ds for the file to found by the ECS Designer, although files with any name can be loaded into the ECS correlation engine.

## Loading Data Stores

In the ECS Designer, Data Stores are loaded from the Simulate menu before a simulation is run.

In the ECS Engine, Data Stores are loaded using the ecsmgr command line tool, or by accessing the appropriate ECS Manager CORBA object. For more information about the ecsmgr command line tool, see the ecsmgr(1M) reference page.

Data Stores in the Engine can be updated at any time. You cannot, however, change Data Stores during simulation in the Designer.

For more information, see the HP OpenView ECS Administrator's Guide and the HP OpenView ECS Developer's Guide and Reference.

## File Formats

This section discusses the format and contents of the files used to load and modify the Data Store.

General file format A Data Store file is an ASCII file following ECDL syntax. The first line is a mandatory header with a special format. Following the header are any number of commands that add or delete key-value pairs.

Header line
The header line is the first line in the file and has the format:
\#path\#date\# version\# future
where:

- a hash sign \#precedes each component
- path is the full path and file name of the Data Store file, and must be in UNIX form even for Windows NT
- date is the date the file was created
- version is the Data Storefile version number. This value is displayed when ecsmgr -info is entered from the command line. (See HP OpenView Event Correlation Services Administrator's Guide.)
- future is reserved for future use. Currently it must have a value of 0 .

For example, Version 3 of the Data Store file
/usr/telecom/telcodata.ds created on August 171997 would have the following header line:

```
#/usr/telecom/telcodata.ds#Thu Aug 17 13:27:30 1997#3#0
```

Commands Commands appear after the header and have the format:
ADD DATA (keyValue, ReturnValue) offset
or
DELETE DATA (keyValue) offset
where:

- Case is important: add data is not equivalent to ADD DATA. The two words may be separated by any amount of whitespace (space, tab or newline characters).
- keyValue is the value used to identify this Data Store tuple


## Data Store and Fact Store <br> Data Store

- returnValue is the value returned by the dataStore function when it is passed keyValue.
- offset is an optional ECDL duration specifying the time after the start time that this command is logically executed.
The offset only applies to ECS Designer in Simulate mode. It is ignored by the ECS Engine. The offset enables event contexts to be simulated in the ECS Designer.


## Comments

A comment is started by two hyphens (--). All text from the start of the comment to the end of the current line is ignored when the file is loaded. Consequently, if a Data Store file is loaded and then unloaded from the ECS Engine, all comments are lost.

## NOTE

Commands can contain whitespace characters (space, tab and newline). The logical end of a command is defined by the closing bracket. This means that long commands can be wrapped to the next line for easy reading if required.

## Adding data

Deleting data

Modifying data

To add new data entries, use the ADD DATA command:

```
ADD DATA ( "OverHeated", 80 )
ADD DATA (( "Reverse", "Trigger"), 60s )
ADD DATA ( "BRKData",[([4],[1]),([2],[3]), ([3],[2,1])] )
```

To delete existing data entries, use the delete data command:

```
DELETE DATA ("OverHeated")
DELETE DATA (("Reverse", "Trigger"))
```

To modify existing data entries, use the ADD DATA command to overwrite the existing value in the Data Store. For example, the existing entry "OverHeated" can be updated with the command:

ADD DATA ( "OverHeated", 82 )

## Data Types

Data Store file entries are tuples containing key-value pairs. This section explains the data types that can be used for both key and return values.

Format of entries Each entry in a Data Store file has this general syntax

```
(keyValue, returnValue)
```

A typical entry looks like this
ADD DATA ("HeartBeat", 25s )
The keyValue is used as the argument to the dataStore function which returns returnValue. Once a file containing the previous command has been loaded into the Data Store the value can be accessed from, say, the Interval parameter of a Clock node with the expression
dataStore "HeartBeat"
This causes the Clock node's Interval parameter to be set to 25 s when the correlation engine is started. In this example, the key value "HeartBeat" has a string data type, and the return value 25 s has a duration data type.

## Permitted data types

Both keyValue and returnValue can be any of the following data types:

- Simple types:
- Duration
- Time
- Integer
- Oid
- Real
- String
- Structured types:
- List
- Tuple

In addition, returnValue can also be a dictionary. keyValue, however, cannot be a dictionary.

In addition the returnValue must be compatible with the type expected where it is used.

Simple data types In the previous example, the Data Store file entry was
ADD DATA( "HeartBeat", 25s )
It could just as easily have been

# Data Store and Fact Store <br> Data Store 

ADD DATA ( 444, 25s )
Using the following expression in a Clock node I nternal parameter:
dataStore 444
will return the value 25 s .

| Computed values | Both the key value and the return value must consist of explicit <br> constants. For example, the Data Store file entry |
| :--- | :--- |
| illegal | ADD DATA( "Heart Beat", 25 s ) |

is legal for an Interval parameter, with a string key and duration return value. But this next example,

ADD DATA( "Heart " + "Beat", 5s * 5 )
consisting of (apparently) equivalent implicit values is illegal. This is because expressions in entries are not evaluated when entries are loaded into the correlation engine.

## See also

- Chapter 3, "Circuit N odes," on page 43
- Chapter 8, "Data Types," on page 211


## Using the Current Context to Access the Data Store

The only function that provides access to Data Store values is the dataStore function. The dataStore function takes a single parameter, the keyValue, and returns the associated returnValue from the Data Store. For example, if the following entry has been added to the Data Store:

ADD DATA( "AlarmDelay", 30s )
then the expression
dataStore "AlarmDelay"
will return the duration 30s. The return type of the dataStore function is always the same type as returnValue. It is up to the circuit designer and the Data Store admi nistrator to ensure that the returned type is compatible.
If the key is not found, the dataStore function raises an exception which can be handled by providing a default value like this:

```
( dataStore "AlarmDelay" )
handle
```

```
    NotFound => 5s
end
```

Note the parentheses () surrounding the call to the dat ast ore function. These are necessary to ensure that the handle applies to the whole expression. Without them, the handle would be applied only to the immediately preceding "AlarmDelay" argument.

## Using Event Contexts to Access the Data Store

An event has an associated Data Store context. The Data Store event context of an event is an attribute of the event. This means the event context is accessed in a manner similar to any other event header attribute (cf. create_time). In this case, the value data_Store is the name of the attribute. It fetches the logical version of the Data Store that existed when the event was created.
For example, for a filter node condition parameter expression we can write:

```
input_event "severity" = input_event data_Store "critical"
```

Here the term input_event data_Store returns the Data Store context from the event. Then the "critical" (string type) Data Store key is applied to the context to get the entry from the Data Store. This is then compared for equality with the input_event attribute identified by the "severity" key.

Note that the same mechanism is used to access the current context and the event context. The expression:

```
input_event data_Store "critical"
```

is evaluated as:

```
(input_event data_Store) "critical"
```

and the term (input_event data_Store) evaluates to a Data Store context which is the same type as the global value dataStore.
One event may be considered in the Data Store context of another event. For an Unless node, the following condition can be configured:

```
let
    val context = input_event data_Store
in
    input_event "device" = context "device"
and
```

Data Store and Fact Store
Data Store

```
    inhibitor_event "device" = context "device"
```

end

Here we get the Data Store context for the input_event, which is the context that existed when the event was created. This context is then used for both the input_event and the inhibitor_event comparisons.

## Fact Store

This section describes the format and usage of the Fact Store, a data structure that contains relationship definitions. These relationships usually describe the local network topol ogy.

Relationships need not be confined to network topol ogies. You could also describe the hierarchy of an organization, or the relationships of your organization with other organizations: suppliers, customers, service providers, carriers, regulatory authorities, standards organizations, and so on. All these relationships could be relevant to correlating events in the network with your operations and services.

The ECS correlation engine loads the Fact Store when the ecsmgr command is run. In the case of the ECS Engine simulator in the ECS Designer, the Fact Store is loaded when the running of events is started. Once loaded, a Fact Store can be modified while the ECS correlation engine is running.

## Generating Fact Store files

Fact Store file names

Loading Fact Stores

When designing circuits, you can create Fact Store files with any text editor. But in a production environment, you will probably program applications or shell scripts to generate these files automatically from equipment or topology databases.

The file suffix must be *. fs for the file to be found by the ECS Designer, although files with any name can be loaded into the ECS correlation engine.

In the ECS Designer, Fact Stores are loaded from the Simulate menu before a simulation is run.

In the ECS Engine, Fact Stores are loaded using the ecsmgr command line tool, or by accessing the appropriate ECS M anager CORBA object. For more information about the ecsmgr command line tool, see the ecsmgr(1M) reference page.

Fact Stores in the Engine can be updated at any time. You cannot, however, change Fact Stores during simulation in the Designer.

For more information, see the HP OpenView ECS Administrator's Guide and the HP OpenView ECS Developer's Guide and Reference.

## Fact Store

## File Formats

This section discusses the format and contents of the files used to load and modify the Fact Store. Both load and update files use the same Fact Store file format.

General file format A Fact Store file is an ASCII file following ECDL syntax. The first line is a header with a special format. The header line is required. Following the header are any number of commands that add or delete facts to or from the Store.

## Header line $\quad$ The header line is the first line in the file and has the format:

```
#path#date# version# future
```

where the format is the same as for the Data Store file described in "H eader line" on page 187.

Commands Commands appear after the header and have the format:
COMMAND(leftValue, relation, rightValue) offset
where:

- COMMAND can be either ADD FACT or DELETE FACt. Case is important: add fact is not equivalent to ADD FACT. The two words may be separated by any amount of whitespace (space, tab or newline characters).
- leftValue is the value returned by the fact_find_left function.
- rightValue is the value returned by the fact_find_right function.
- relation defines the relationship between the two values and forms a key in conjunction with either one of them. relation is passed as the second parameter to both fact_find_left and fact_find_right.
- offset is an optional ECDL duration specifying the time after the start time that this command is logically executed.

The offset only applies to ECS Designer in Simulate mode. It is ignored by the ECS Engine. The offset enables event contexts to be simulated in the ECS Designer.

Comments A comment is started by two hyphens (--). All text from the start of the comment to the end of the current line is ignored when the file is loaded. Consequently, if a Fact Store file is loaded and then unloaded from the

ECS Engine, all comments are lost.

NOTE
Commands can contain whitespace characters (space, tab and newline). The logical end of a command is defined by the closing bracket. This means that long commands can be wrapped to the next line for easy reading if required.

## Adding facts To add a new fact entry use the ADD FACT command:

```
ADD FACT (("Switch", "SW_71"), 176, ("PowerSupply", "Solar
BAT82"))
ADD FACT (1.43.23.1, 24, 1.43.225.5)
ADD FACT ("SprayCom Corp", 76, "Microwave Link LK47")
ADD FACT (("Switch", "SW_71"), 766, ("Supplier", "Mercata"))
```

Deleting facts To delete an existing fact entry use the DELETE FACT command:

```
DELETE FACT (("Switch", "SW_71"), 176, ("PowerSupply", "Solar
BAT82"))
```

When deleting a fact entry the complete fact must be supplied, just as it was when the fact was added.

You can also delete a series of related facts by replacing the left or the right value with (). For example:

```
DELETE FACT (("Switch", "SW_71"), 176, ())
```

would delete all fact entries that matched the left value and relation supplied, regardless of the right value.

## Data Types

Fact Store file entries consist of two values of any allowed data type, "joined" by a relationship. The relationship must always be an integer.

Format of entries Each entry in a Fact Store has this general syntax

```
(leftValue, relation, rightValue)
```

A typical entry looks like this

```
ADD FACT( "SprayCom", 76, "LK47" )
```


## Data Store and Fact Store

## Fact Store

## Permitted data types

Both leftValue and rightValue can be any of the following data types:

- Base types:
- Duration
- Time
- Integer
- Oid
- Real
- String
- Structured types:
- List
- Tuple

In addition both values must be compatible with the type expected in the node parameter where they are ultimately used.

The relation must always be an integer. Typically, a standard list of relationships is defined and loaded as a set of Global Definitions so that circuit designers do not need to remember the relation numbers. For example:

```
-- STANDARD RELATIONSHIP DECLARATIONS
-- Only these relationships are to be used in
-- SprayCom Support Corporation ECS circuits
\begin{tabular}{lll}
-- & Relation & Equivalent integer \\
& & \(=24\) \\
val & is_connected_to & \(=176\) \\
val & is_powered_by & \(=1029\) \\
val & is_owned_by & \(=43\) \\
val & is_serviced_by & \(=766\) \\
val & is_leased_from & \(=76\) \\
val leases & \(=891\) \\
val is_routed_via & \(=666\) \\
val & is_registered_with & \(=6\) \\
val has_service_contract & \(=113\)
\end{tabular}
```

Once entered as Global Definitions, these bindings can be used in ECS circuits instead of the relation numbers. You must, however, use numbers when defining relationships in the Fact Store file.

A text file containing your standard relationship dedarations is a useful reference for circuit designers.

## Computed values illegal

Values in a Fact Store entry must consist of explicit constants. For example, the Fact Store file entry

ADD FACT ( "SprayCom", 70 + 6, "LK" + "47" )
is illegal because expressions in entries are not evaluated when entries are loaded into the correlation engine.

See also

- Chapter 3, "Circuit Nodes," on page 43
- Chapter 8, "Data Types," on page 211


## Using the Current Context to Access the Fact Store

## Overview The functions that access the Fact Store are:

- fact_exists (leftValue, relation, rightValue), which checks whether a complete relationship is present, and returns a boolean value of true or false
- fact_find_right (leftValue, relation), which returns the right values of relationships as a list containing none, one, or more el ements
- fact_find_left (rightValue, relation), which returns the left values of relationships as a list containing none, one, or more el ements

N ote the following:

1. The arguments to fact_exists is a tuple with three elements and the argument to fact_find_left and fact_find_right is a tuple with two arguments.
2. The order of the elements in the lists returned by the fact_find_right and fact_find_left functions is undefined.

## Example relationships

To understand how the Fact Store functions work, consider these example relations in a Fact Store:

```
("aaa", 1, "bbb")
("aaa", 1, "ccc")
("ddd", 1, "ccc")
("eee", 2, "fff")
```


# Data Store and Fact Store <br> Fact Store 

```
("eee", 3, "ffff")
```

where the relationship values 1,2 and 3 mean:
1 is contained in
2 is the parent of
3 is equal to

## Checking the existence of a relationship

When you use the fact_exists function, you must give the entire relationship as the argument, that is a tuple containing leftValue, relation, and rightValue. For example, given the Fact Store above,

- fact_exists ("aaa", 1, "bbbb") returns true
- fact_exists ("aaa", 3, "bbb") returns false
- fact_exists ("eee", 1, "bbb") returns false

Returning the right When you use the fact_find_right function, you provide a tuple value containing the leftValue and the relation as arguments. Thereturned list contains a copy of all right Values that match the supplied leftValue and relation. An empty list is returned if no match is found. For example, given the Fact Store above,

- fact_find_right ("aaa", 2) returns []
- fact_find_right ("ddd", 1) returns ["ccc"]
- fact_find_right ("aaa", 1) returns ["bbb", "ccc"]


## Returning the left When you use the fact_find_left function, you provide a tuple value

See also

- Chapter 3, "Circuit N odes," on page 43
- Chapter 8, "Data Types," on page 211
- "fact_exists" on page 267
- "fact_find_left" on page 268
- "fact_find_right" on page 269


## Using E vent Contexts to Access the Fact Store

Alternative forms of the in-built E CDL Fact Store access functions are provided to access a given Fact Store context. For example, for the input_event accessible in a filter node condition:

```
fact_exists_ctxt (input_event fact_Store) (left_fact,
relationship, right_fact)
```

The (input_event fact_Store) argument is a Fact Store context, here the one from input_event. The function returns true if the fact exists, else false.

```
fact_find_left_ctxt (input_event fact_Store) (right_value,
relationship)
```

returns a list of all facts which satisfy the requirement. Similarly we have:

```
fact_find_right_ctxt (input_event fact_Store) (left_value,
relationship)
```

As in the Data Store example, a single Fact Store context can be used in multiple places:

```
let
val facts = input_event fact_Store
in
fact_exists_ctxt facts (f1, r1, g1)
or fact_exists_ctxt facts (f2, r2, g2)
end
```

The global value factStore can be passed to these fact functions to represent the current Fact Store context, as in:

```
fact_exists_ctxt factStore (left_fact, relationship,
right_fact)
```


## Multiple Data and Fact Stores

Any number of named Data and Fact Stores may be loaded into an engine. When a circuit is loaded it may be associated with at most one Data Store and one Fact Store. Multiple circuits may reference a common Store.

In fact, any store file (Data or Fact) may contain either or both Data Store entries and Fact Store entries. A useful adaptation of this is to use a single Fact Store file to hold common facts and data and individual Data Stores for each circuit to hold per-circuit private data and facts.

## Simulating Event Contexts in the ECS Designer

## Overview

ECS Designer allows only a single Data Store and a single Fact Store to be loaded, before a circuit is simulated. U pdating the Data and Fact Stores is not supported in the Designer, only in the ECS Engine.
offset values are used after the ADD and Delete commands in the Data and Fact Stores to allows contexts to be simulated.

It is assumed that the sequence of commands that add and delete Data and Fact Store entries are intended to emulate the initial loading and subsequent real-time updates to the respective Stores. This means that the order of the commands is important, with subsequent commands being deemed to be effected later in time.

## ADD Commands

An ADD command causes an entry to be added. If a relative time is provided, the entry will be added to the appropriate Store version. If the entry is the only entry for this data or fact entry and has no relative time parameter, it is assigned to the first version. If the entry already exists (was previously added to a version), and the subsequent entry contains a relative time, it is assigned to the appropriate version. If a subsequent entry does not contain a relative time, it is assigned to the first version. See Table 6-2.

Table 6-2 Adding Store Entries

| Relative <br> Time | Existing <br> Entry | Data Store Action | Fact Store Action |
| :--- | :--- | :--- | :--- |
| no | no | Add to first version. | Add to first version. |
| no | yes | Replace entry in first version. | If identical, no action. |
| yes | no | Add to version with time stamp equal <br> to simulation start time plus offset. | Add to version with time stamp <br> equal to simulation start time plus <br> offset. |

## Simulating Event Contexts in the ECS Designer

## Table 6-2 Adding Store Entries

| Relative <br> Time | Existing <br> Entry | Data Store Action | Fact Store Action |
| :--- | :--- | :--- | :--- |
| yes | yes | Replace entry in version with time <br> stamp equal to simulation start time <br> plus offset. | If identical, no action. |

When an entry is added to a Store version, it is automatically added to all subsequent Store versions (i.e. the entry persists in time) unless the entry is deleted or changed for a subsequent version. For backward compatibility, for input files with no relativetimes, there is only one Data Store version and one Fact Store version, and these are used for all contexts.

## DELETE Commands

A delete command causes an entry to be deleted. There may be multiple instances of a data item or a fact participating in different Store versions. If the relative time is not given, the command applies to all versions (supporting backward compatibility).

If a relative time is given, the entry will be del eted from the identified Store version and later Store versions. Any earlier versions of the entry will continue to exist. If a different entry should exist in later versions, it must be added by subsequent commands with appropriate relative times.

Any reference to a deleted (or otherwise non-existent) data entry in a Store version will result in a NotFound exception being raised.

## Dumping Stores

The Data Store or Fact Store can be dumped from the ECS Engine via the ecsmgr utility or via an operation on an interface of the ECSM anager object. When a Store is dumped, it is dumped with a relative time on each entry. The oldest version will have a "relative time" of zero. This format will allow the file to beloaded intothe ECS Designer simulator. If the resultant files are reloaded into an ECS Engine, only the most recent entries will be used, since time offsets are ignored and (by definition) all events must occur after the latest entries are loaded.

## 7 <br> Identifiers, Comments, and Reserved Words

This chapter summarizes the requirements for constructing identifiers, inserting comments, and lists reserved words and symbols:

- "I dentifiers" on page 205.
- "Comments in ECDL" on page 206.
- "Reserved Words and Symbols" on page 207.


## Identifiers

I dentifiers are names constructed with a subset of characters between ASCII code 32 (" " space) and 126 ( $\sim$ tilde). An identifier must begin with a letter, followed by any combination of letters, digits, underscores, and apostrophes ('). Other characters are not permitted. Be careful with underscores ( _ ) -an underscore on its own has a special meaning.
Letters can be upper-case (A) or lower-case (a), but ECDL is case-insensitive, and treats all letters as lower-case (except within strings). For example, Event, event, and eVeNt are all equivalent.

Identifiers must The words and symbols summarized in Table 7-1 and Table 7-2 have not be reserved special meanings and cannot be used for ECDL identifiers such as value names. Doing so will cause a syntax error when the circuit is compiled.

NOTE
Correlation circuit file names are identifiers and so must follow the same rules.

## Comments in ECDL

ECDL comments begin with a double-hyphen "--" and terminate with either the end of the line, or another double-hyphen. For example:
-- This entire line is a comment
val myValue $=23$-- This is a comment after my value -- definition

```
val myValue = --23-- 24 -- This is a comment after my value
    -- definition, where the value 23 has
    -- been "commented out", and the
    -- value 24 is used instead.
```


## Reserved Words and Symbols

Table 7-1 lists the reserved words that you cannot use for identifiers in ECDL expressions. Table 7-2 lists the symbols that have special meaning in ECDL expressions.
Table 7-1 Reserved Words

| and | annotate | any | as |
| :--- | :--- | :--- | :--- |
| attributes | case | choose | clock |
| combine | compound | count | create |
| data | delay | dict | else |
| end | exception | exists | external |
| extract | false | filter | find |
| fn | handle | from | fun |
| graphics | in | infix | if |
| implies | modify | module | node |
| internal | of | open | or |
| mod | parameters | pragma | prefix |
| not | then | select | simple |
| outputs | typecase | token | true |
| raise |  |  | val |
| table | type | where |  |

## Table 7-2 Reserved Symbols

| Symbol | Names or Functions |
| :---: | :---: |
| () | precedence operator; tuple delimiter; dictionary delimiter; void |
| [] | list delimiter |
| , | separator (of elements in lists, tuples, and dictionaries) |
| - | separator (of components in nodes, modules, Oid's) |
| ; | - |
| : | is of data type |
| : | prepend (element to list) |
| + | add (numeric); concatenate (string) |
| - | subtract |
| * | multiply (numeric); construct (tuple pattern) |
| / | divide |
| ^ | exponentiate |
| $=$ | equal |
| ! $=$ | not equal |
| > | greater than |
| >= | greater than or equal to |
| < | less than |
| <= | less than or equal to |
| \| | where (list builder); alternative (type declarations) |
| :- | such that |
| => | consists of, is (function declarations); maps to (dictionaries) |
| -> | returns (function type declarations) |

## Table 7-2 Reserved Symbols

| Symbol | Names or Functions |
| :--- | :--- |
| -- | comment delimiter |
| - (underscore) | Place holder in a binding pattern. |
| $\{$ | reserved |
| $\}$ | reserved |
| $\#$ | reserved |
| @ | reserved |

- Chapter 9, "Operators and Built-in Functions," on page 233
- Chapter 10, "Writing ECDL Expressions," on page 319


## 8 <br> Data Types

This chapter summarizes the data types used in the ECDL language, which in turn is used for all parameters and values used in the correlation engine.

Simple data types • "I nteger Data Type" on page 213

- "Real Data Type" on page 214
- "Boolean Data Type" on page 215
- "Duration Data Type" on page 216
- "Time Data Type" on page 217
- "Oid Data Type" on page 218
- "String Data Type" on page 219
- "Token Data Type" on page 221

Structured data - "Tuple Data Type" on page 222
types

- "List Data Type" on page 223
- "Dictionary Data Type" on page 225

Other data types - "F unction Data Type" on page 228

- "F unction Data Type" on page 228
- "Opaque Data Type" on page 229
- "Dynamic and User Defined Types" on page 230

More advanced information about the ECDL type system and its use is presented in Chapter 10, "Writing ECDL Expressions," on page 319.

## Integer Data Type

| Definition | Integers are 32-bit whole numbers. |
| :---: | :---: |
| Representation | Integers are represented as a string of digits. Numbers without a leading |
|  | +sign are assumed to be positive. Commas are not allowed. For example: |
|  | $21+47-4759$ |
|  | The range of numbers is $+\boldsymbol{+} \mathbf{2}, 147,483,647$. |
| Operators and functions | Integer expressions can be manipulated with the arithmetic operators, |
|  | + - * / mod div ^ |
|  | compared with the relational operators, |
|  | = ! $=$ \gg= \ll= |
|  | converted to a string representation, |
|  | stringof chr |
|  | and treated as bitmasks with the bit string operators, |
|  | bitand bitor bitxor bitinv bitleft bitright |
| NOTE | A negative number must be enclosed in parentheses when used as a |
|  | literal argument to a function, to ensure that it binds correctly. For |
|  | (-2.5) instead. |

## Real Data Type

Definition | Real numbers are implemented as 64 bit IEEE format double precision |
| :--- |
| floating point numbers, which have a range from |
| -2.2250738585072014 E 308 to 1.7976931348623157 E 308 . The smallest |

Representations Real numbers are expressed with a decimal point, and may include an exponent. Note that 100 is an integer, whereas 100.0 is a real number. Further examples of real numbers are:

```
21.0 -629.4 2.34E-7
```

Operators and functions

Real numbers can be manipulated with the arithmetic operators,

+     -         * / ^
converted to integers,
truncate round
compared with the relational operators,
$=\quad!=\quad>\quad>=\quad<=$
and converted to a string,
stringOf

NOTE
A negative number must be enclosed in parentheses when used as a literal argument to a function, to ensure that it binds correctly. For example, instead of writing round -2.5 , write round ( -2.5 ) instead.

## Boolean Data Type

| Definition | Boolean values are either true or false. No other value is allowed and no <br> type conversions are available. Specifically, you cannot substitute an <br> integer for a bool ean data type, as is possible in some other languages. |
| :--- | :--- |
| Representation | A boolean expression evaluates to one of only two values: true or false. <br> Operators and <br> functions |
| Boolean expressions can be manipulated with the logical operators, <br> and or implies not <br> compared with the relational operators, |  |
|  | $=\quad$ and converted to a string representation with |

## Duration Data Type

Definition The duration data type is used to store relative times. A relative time is the elapsed time between two absolute time points, as represented by the time data type. Duration is a signed data type with a resolution of 1 microsecond and a range of approximately $+/-596,523$ hours (68 years).

Representation
A duration is specified in hours, minutes and/or seconds. The specification can consist of any or all of these parts. Each part is followed immediately by one of the letters $h, m$, or $s$, as appropriate. For example:
13.5h 13h30m 44s

There must be no spaces between the components of a duration value, for example, 12 h 27 m 52.5 s is correct but 12 h 27 m 52.5 s is incorrect.

If there is only one number, the letter can be omitted and the value is read as seconds. For example, 44 is interpreted as 44.0 s in a context where a duration is expected. It is recommended to always add the trailing s for clarity. The decimal point is optional: 44 s is the same as 44.0s.

Operators and functions

Duration expressions can be manipulated with the arithmetic operators, + - * /
compared with the relational operators,
and converted to a string representation with
stringOf
Integer and Real values are converted to Duration in any mixed context. For example, in the expression $24 \mathrm{~h}+1$, the 1 is converted to a duration of 1 s before being added to the 24 h to give a result of 24 h 0 m 1 s .

## Time Data Type

Definition \begin{tabular}{l}
The time data type is used to represent absolute time. That is, a date and <br>
a time of day on that date. <br>
For example, event creation times are represented using the time data <br>
type. Elapsed times, such as network delays, are represented with the <br>
duration data type. <br>
A time value (absolute time) is represented internally as the time <br>
elapsed since 00h 00m 00s, 1 J anuary 1970 (UTC). Time data type has a <br>
resolution of 1 microsecond and can represent any time up to the year <br>
2038.

 Representation $\quad$

Literal time values are represented in the following format: <br>
yyyymmadhhiss. uuuuuuz
\end{tabular}

| For example, the date/time value J anuary 3, 1997 at 1:59:59.123456 PM |
| :--- |
| (UTC) would be: |

Operators and Time values can be manipulated by adding or subtracting a duration to functions produce a time,
subtracting from another time to produce a duration,
compared with the relational operators, and converted,
explode_time
The current time can be obtained by calling,
Time.now ()

## Oid Data Type

## Oid Data Type

Definition The oid data type is used to represent Object I dentifiers.
The term "oid" is the abbreviation of object identifier used to uniquely identify information objects.

Representation
An oid value is a sequence of integers separated by periods. For example:

$$
1.43 .67 .52
$$

An oid value must contain at least two dots. For example, the value 1.23.689.12 is interpreted as an oid but the correlation engine treats the value 1.23 as a real number and raises a type error.

Operators and functions

Oid expressions can be compared with the relational operators:

```
= != > >= < <=
```

or converted to a string:

```
stringOf
```


## String Data Type

Definition The string data type is an array of multibyte characters. Each character is a value that represents a character in the current character encoding. There is no practical implementation limit to the size of a string.

Representation A string is entered as a sequence of characters enclosed in double-quotes (". . ."). For example:
"notificationIdentifier"
"A newline starts after this $\ n$ "
Two quotes typed together represent an empty string:
""
A string can be continued over several lines by ending the current line with a backslash:

```
"To break a string \
over several lines \
end each line with a backslash."
```

The backslash ( $\backslash$ ) is also used as an escape character to allow special characters to be entered. For example, " $\backslash t$ " represents a tab character and " $\backslash \backslash$ " represents a single backslash character. Table 8-1 summarizes additional special characters and character combinations for representing special characters and symbols.
Table 8-1 Special Character Combinations in String Expressions

| Combination | Description | Comment |
| :--- | :--- | :--- |
| $\backslash$ | Special combination <br> escape | Signals the start of one of the following special <br> combinations. |
| $\backslash "$ | Double-quote | Enters a double-quote (") in the string. |
| $\backslash \backslash$ | Backslash |  |
| $\backslash t$ | Tab |  |
| $\backslash n$ | New line |  |
| $\backslash r$ | Carriage return |  |

## Data Types

String Data Type

Table 8-1 Special Character Combinations in String Expressions

| Combination | Description | Comment |
| :--- | :--- | :--- |
| $\backslash n n n$ | Character code | Enters a character corresponding to the octal code <br> $n n n$, where $n n n$ must represent a valid character in the <br> current character encoding, otherwise an InvalidArgs <br> exception is raised. |

Event arguments Event arguments mapped to ECDL strings may not support localization. This is because some endecoders do not support anything other than ASCII (e.g. MDL supports 8-bit ASCII events only).

## Operators and Strings can be concatenated, functions

| Relational | The relational operators $=,!=,>,>=,<$, and $<=$ compare pairs of string <br> operations <br> arguments character-by-character using the current character encoding <br> prefix of a longer string is always "less than". |
| :--- | :--- |
| String search | Sophisticated string search and compare is available with the match <br> functions. |
| stringOf | The stringof function produces a string representation of any data <br> type, including strings. |

## Token Data Type

Definition The token data type describes a set of discrete alternatives. A token is one value from the set.

Token data types are used as keys for the header attributes of events within an ECS circuit. Token types are also used as keys in the dictionary returned by the explode_time function.

Representation Tokens are identified by name anywhere within the scope of their type definition.

Examples
You can create your own tokens by writing a type definition. The following statement defines a new token type to identify a set of warning levels:

```
type Warning = token( low, medium, high, panic )
```

The four warnings low, medium, high and panic can now be used freely to identify an error anywhere in the enclosing scope. For example:

```
if errorLevel = panic
then throwHandsInAir
else process errorLevel
end
```

There are also a number of predefined tokens. For example, the following tokens are used as keys in event headers:

```
unique_id event_class create_time
```

Some Token names may be qualified by a module name in order not to contaminate the user's own name space. The following predefined tokens are used in the dictionary returned by the explode_time function. These token names are of the form time. Name because they bel ong to a module named time. For example,
time.year time.hour
Qualifying token names in this way makes it possible to write:

```
let
```

    val year \(=(\) explode_time (time.now()) time.year)
    
## Tuple Data Type

Definition

Representation

## Examples

Operators and functions

A tuple is a data structure consisting of a fixed collection of elements, where each element can be any type. Tuples allow you to define your own structured type.
A tuple type definition is composed of two or more types separated with the "*" type constructor.

Tuple values are represented as a sequence of expressions separated by commas and enclosed in parentheses. Each expression in the Tuple must conform to the type specification for that element.

To define a Tuple type that consists of an integer and a string:

```
type xType = (Integer * String)
```

Having defined the new type xType, you can now create a named value of this type:

```
val x : xType = (4, "abc")
```

The presence of the type constraint ensures that x must match the number and type of the elements.
Individual elements in a Tuple may be accessed by binding identifiers to the elements in the Tuple:

```
let
    val (High, Low) = input_event "Threshold"
in
    High > 60
end
```

Tuple expressions can be compared for equality or inequality:

```
= !=
```


## List Data Type


#### Abstract

Definition A list is a variable-length ordered set of values. All the elements in a list must be of the same nominal data type. However, by making use of dynamic data types, lists can be defined that hold a variety of data. If a data type for the list is not defined, then it is set by the compiler to the most restrictive type that includes all the elements of the list.

Representation A list can be defined to hold any combination of data types. For example,


 a list to hold integers can be defined as:type IntList $=$ list integer
and a list to hold integers and reals can be defined as:
type NumList $=$ list ( integer $\mid$ real )
Having defined these types we can assign values by providing a list of values separated by commas and enclosed in square brackets, as follows:

```
val l : IntList = [1, 2, 3, 4]
val n : NumList = [1, 2.12, 3, 4.0]
val u = [1, 2.12, "Aspro"]
```

The last example above defines a list with no type restriction. In this case $u$ is assigned a type of list any type.

Using lists Lists are described in detail in "List Manipulation" on page 327.
List expressions A list can be inspected element by element using the expressions:
exists forall
A list can be inspected element by element until the first time a specified condition is true. The result is a tuple consisting of the first element for which the condition is true plus the remainder of the list:
find element in $[2,4,1,7]$ where element $=4$
See "Searching Lists" on page 328 for details.

## Building lists See "Deriving Lists from Lists" on page 330

## Operators \& A list can have an element prepended to it:

 functions
## Data Types

## List Data Type

This results in a list of three elements [1,2,3].
The first element of a list can be accessed by binding a named value to the first element and another name to the remainder of the list. In the following:

```
val HeadElement :: TailList = [1,2,3]
```

HeadElement is bound to the value 1 and TailList is bound to the List value $[2,3]$.

An arbitrary element of a list can be accessed using the nth function:

```
nth 2 [1,2,3]
```

The length of a list can be found:
length [1, 2, 3]
A new list can be created whose elements are in the reverse order:

```
reverse [1,2,3]
```

A list can be built from two list expressions using the set operators:

```
union inter unionc interc
```

A list can be inspected element by element and a data value calculated for the whole list:

```
foldl foldr
```


## Dictionary Data Type

| Definition | A dictionary is an unordered collection of key-value pairs. Values are <br> retrieved from a dictionary by specifying the key. |
| :--- | :--- |
| Representation $\quad$ | A dictionary type is defined as (for example): <br> type Postal = dict (string, integer) |
| $\quad$The typical dictionary shown in this example has a string type for the <br> keys and an integer type for the values in each attribute. To definea <br> dictionary of type Postal, you use the mapping operator => to map values <br> to keys: |  |
| val aust $:$ Postal="Burwood" $=>3125, ~ " B l a c k b u r n "=>3103, ~$ <br> "Perth" $=>6001$ ) |  |

To retrieve a value from this dictionary you specify the key:

```
val PostCode = aust "Perth"
```

which returns the integer value 6000 .
The following represents an empty dictionary:
(=>)
Element data A dictionary key must be a simple data type such as an integer or string. types

Operators and $\quad \begin{aligned} & \text { Dictionaries are used on the right-hand side of the alter function to } \\ & \text { change the attributes of an event. For example: }\end{aligned}$

```
created_event alter ("severity"=>6, "deviceId"=>"TUNA")
```


## Event Data Type

Definition An Event behaves like a Dictionary with the type:

```
dict( (Any Simple), (Any Data) )
```

in a context where a dictionary is expected. Some built-in functions have restrictions where an event cannot be used but a dictionary can.

## Element data types

An Event key must be a simple data type such as an integer or string. Values can be of any type, and may be different from one key-value pair to the next within the same dictionary. Event keys are generally called "attributes". By convention event header attributes are defined as Tokens and primitive event body attributes are defined as Strings.

Using dictionaries Like Dictionaries, Events are read-only except in two specially defined circumstances: in the Create Spec parameter of a Create node and in the Modify Spec parameter of the Modify node. The alter built-in function can be used in these parameters to set event attributes. See "alter" on page 252 for details.

Operators and functions

The only dictionary operator is alter, which is restricted for use with events in the Create and M odify nodes only.

## NOTE

 Events effectively have a predefined set of attributes whose values may be set (using alter) but that cannot be removed. To "remove" an optional attribute, set its value to Void:```
created_event alter ("optionalAttribute"=>())
```


## Void Data Type

Definition $\quad \begin{aligned} & \text { Void is a special type. It has only one value that represents nothing } \\ & \text { (void). }\end{aligned}$

Representation A Void type is defined as:
type Nothing = Void
However, this syntax is rarely used as it is sufficient to use () to indicate a Void value. For example:
val $t=$ Time.now ()
where Time. now is a built-in function that requires a Void argument.
Using void The Void value "()" is typically used to represent the ASN. 1 NULL value or OPTIONAL attributes that are not present.

Operators and Tests for equality are the only operations supported on void. functions

Data Types<br>Function Data Type

## Function Data Type

J ust as the numerals 3,10 or -20 are values of the type integer, functions are values of the type function. A function type expression describes the types of the arguments and the type of the result. Each combination of types in a function type is a different function type. There may be many different functions that have the same function type.

Definition The basic form of a function type expression is:

```
param_1_type -> param_2_type -> ... -> param_n_type ->
```

return_type
where param_1_type to param_n_type is a list of parameters expected by the function when it is called, and return_type is the type returned by the function.

## Using Function types

Operators and functions

For example, nth is a built-in function that has an integer as its first parameter, a list as its second parameter, and returns any type (because it depends on the list element type) and may have been defined something like:

```
fun nth (i : Integer) (l : List (Any Type)) : (Any Type) = ...
```

A call to this function may look like this:

```
val result = nth 2 [1, 34, 2]
```

where result will be the integer value 34 .
The nth function has the following function type expression:

```
Integer -> List (Any Type) -> (Any Type)
```

The parentheses around (Any Type) are optional, but are included here for clarity.

See also "User-Defined types" on page 231.
There are no operators defined for function data types. In particular, you cannot use the equality operator to compare function types for identity.

## Opaque Data Type

| Definition | Opaque is a special data type that ECDL cannot manipulate. Typically, <br> an Opaque data type is used as a "handle" to a block of data that needs to <br> be passed around but never examined. It is rarely used, however, the <br> ECDL string pattern matching functions are an example of the use of an <br> Opaque type. |
| :--- | :--- |
| Representation | There is no representation for values of the Opaque data types. |
| Using Opaque | The ECDL match.make function is the only place you will probably use <br> types |
|  | let Opaque data type: |
|  | in val pat = match.make ("cat", "", IgnoreCase ) |

Here, the name pat is associated with the Opaque data type value returned by the match. make function.

Operators and There are no operators defined for Opaque data types. In particular, you functions

## Dynamic and User Defined Types

ECS performs type checking when a circuit is verified or compiled. This catches many type errors before the circuit is even run. However, the data types of event attribute values may not be known until run-time. ECDL supports dynamic data types for these values and performs type checking at run-time.

ECDL compile-time type checking minimizes the amount of run-time type checking by trying to determine a subset of types that a run-time type may take. You can assist this process by specifying dynamic types that support all required subtypes, and exclude support for unwanted subtypes. This brings type-checking forward to compile-time and consequently improves error reporting.
The data types Integer, Real, Boolean, String, Duration, Time, Oid, and Token are referred to as the Simple data types in the following sections.

## Dynamic types

A dynamic type consists of the keyword Any, followed by one of the keywords in Table 8-2.
For example:

```
val someSimpleValue : Any Simple =
    input_event "eventInfo.additionalText"
```

Table 8-2 Dynamic Types

| Keyword | Details |
| :--- | :--- |
| simple | All simple types (including any user-defined simple types and <br> token types). |
| list | All list types (including any user-defined list types). |
| dict | All dict types (including any user-defined dict types). |

## Table 8-2 Dynamic Types

| Keyword | Details |
| :--- | :--- |
| data | All types except functional types. |
| type | Any type at all, including functional types. |

Dynamic types are also useful when defining functions, as described in "User-Defined types" on page 231.

## Union types

Union types serve a similar purpose to dynamic types, the difference being that you can explicitly define the allowed types. The union type is expressed by the joining of the possible event types by "|" (vertical bar character). For example:

```
val fred : (Integer | String) = 2
```


## User-Defined types

A type declaration may be used to name a user-defined type. The declaration takes the following form:

```
type new_type_identifier = type_expression
```

Where new_type_identifier is the name of the user defined type, and type_expression is the ECDL type definition. For example:
type StringInteger $=$ (Integer | String)
This creates a new type called StringInteger, which is a union type of integer and string types. The user defined type name can now be used anywhere you would specify a type expression.
User Defined types should be defined in the "Global Definitions" for your circuit.

Data Types
Dynamic and User Defined Types

# $9 \quad$ Operators and Built-in Functions 

This chapter describes all the operators and built-in functions provided with the Event Correlation Description Language (ECDL):

- "Operators" on page 235
- "Built-in Functions" on page 247

See also Chapter 8, "Data Types," on page 211 and .

## Operators

Operators are the arithmetic and logical functions represented by the usual character symbols $+->=$ etc. Most operators are heavily overloaded. That is, they are implemented for more than one data type. For example, the + operator is implemented for integer, real, duration and string data types.

The + and - operators also support two separate forms: they can each operate on numbers as a unary operator taking one operand, as well as an infix operator taking two operands.

Particular care must be taken with the unary minus operator - because, under some circumstances, it can be interpreted as subtraction. To avoid this it is a good idea to place parentheses around negative numbers when used as literal arguments to functions: (-15).
Table 9-1 Summary of ECDL Operators

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| + | Add two number or concatenate two <br> strings | "+ (add, string concatenate, positive)" <br> on page 237 |
| - | Subtract the second number from the <br> first; unary minus | "- (subtract, negative)" on page 238 |
| $*$ | Multiply two numbers | $" *$ (multiply)" on page 240 |
| $/$ | Divide a real number or duration by <br> another number | "/ (real divide)" on page 241 |
| ^ | Raise the first number to the power of <br> the second | "^ (exponentiate)" on page 242 |
| $=$ | Test the first value for equality with the <br> second | "= and != (equality operators)" on <br> page 243 |
| $!=$ | Test that the first number is NOT equal <br> to the second | $"=$ and != (equality operators)" on <br> page 243 |
| $>$ | Test that the first value is greater than <br> the second | $" \gg=\ll=$ (relational operators)" on <br> page 244 |

## Operators and Built-in Functions

## Operators

## Table 9-1 Summary of ECDL Operators

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| $>=$ | Test that the first value is greater than or <br> equal to the second | $" \gg=\ll=$ (relational operators)" on <br> page 244 |
| $<$ | Test that the first value is less than the <br> second | $" \gg=\ll=$ (relational operators)" on <br> page 244 |
| $<=$ | Test that the first value is less than or <br> equal to the second | $" \gg=\ll=$ (relational operators)" on <br> page 244 |
| $::$ | Add the first value to the list represented <br> by the second value | ": (list constructor)" on page 246 |

## + (add, string concatenate, positive)

| Syntax | + integer |
| :---: | :---: |
|  | + real |
|  | + duration |
|  | string1 + string2 |
|  | integer1 + integer2 |
|  | reall + real2 |
|  | duration1 + duration2 |
|  | time + duration |
|  | duration + time |
|  | Where: |
|  | If one of the arguments is duration and the other is integer or real, then the integer or real is promoted to a duration with a unit of seconds. Likewise, if one of the arguments is a real and the other is an integer, then the integer is promoted to a real before the addition. |
| Description | The first form, with a single argument, returns its argument unchanged and is a redundant form. |
|  | If both arguments are of string data type then the result is the concatenated string result of the two arguments: string1string2. Otherwise, the result is the numeric addition of the two arguments. |
|  | For numeric addition the data type of the returned value depends on the data type of the arguments. If both arguments are of the same data type then the result is the same data type. If the arguments are of different data types then the result is the same data type as the "wider" of the two arguments, where the order from narrowest to widest is: integer, real, duration. |
|  | Adding a time and a duration produces a time. |
| Examples | The redundant form +23 returns its argument: 23. |
|  | Adding a real and an integer in the expression $1.414+1$ returns the real number 2.414. |
|  | The expression "Hewlett" + " Packard" results in the string "Hewlett Packard". |

## Operators and Built-in Functions

## Operators

## - (subtract, negative)

Syntax |  | - integer |
| :--- | :--- |
|  | - real |
|  | - duration |
|  | integer1 - integer2 |
|  | reall - real2 |
|  | duration 1 - duration2 |
|  | time - duration |
|  | time - time |
|  | Where: |

If one of the arguments is duration and the other is integer or real, then the integer or real is promoted to a duration with a unit of seconds. Likewise, if one of the arguments is a real and the other is an integer, then the integer is promoted to a real before the subtraction.

## Description

## Examples

The first form, with a single argument, is equivalent to 0 - arg. Otherwise, the result is the numeric subtraction of the second argument from the first.

The data type of the returned value depends on the data type of the arguments. The first form al ways returns a value of the same type as its single argument. If both arguments are of the same data type then the result is the same data type. If the arguments are of different data types then the result is the same data type as the "larger" of the two arguments, where the order from smallest to largest is: integer, real, duration

Subtracting a duration from a time produces a time value. Subtracting a time from a time produces a duration.

The form - 42 returns the value -42 .
Subtracting an integer from a real in the expression 1.414-1 returns the real number 0.414.

NOTE In an expression such as round (-1.5), the parentheses are essential. Without them the expression round -1.5 is interpreted as an attempt to subtract 1.5 from round. That is, round - (1.5).

See also

- "Time Data Type" on page 217


# Operators and Built-in Functions 

## Operators

## * (multiply)

Syntax | integerl * integer2 |
| :--- |
| reall * real2 |
| Where: |
| Any combination of integers, reals and durations can be multiplied |
| together, except that two durations cannot be multiplied. If one of the |
| arguments is duration and the other is integer or real, then the |
| integer or real is promoted to a duration with a unit of seconds. |
| Likewise, if one of the arguments is a real and the other is an integer, |
| then the integer is promoted to a real before the multiplication. |

Description

| The result is the numeric multiplication of the arguments. |
| :--- |
| The data type of the returned value depends on the data type of the |
| arguments. If both arguments are of the same data type then the result |
| is the same data type. If the arguments are of different data types then |
| the result is the same data type as the "wider" of the two arguments, |
| where the order from narrowest to widest is: integer, real, duration. |

Examples | Multiplying an integer and a real in the expression $1.414 * 2$ returns |
| :--- |
| the real number 2.828. |

## / (real divide)

Syntax ..... reall / real2
duration / integerduration / realduration / durationWhere:If either or both arguments are integer then they are promoted toreal. If one argument is duration then the other is promoted to aduration. In other words, all division using the / operator is donewith reals or durations. Use the div built-in function for integerdivision.
Description The result is the numeric division of the first argument by the second.The data type of the return value depends on the data type of thearguments. If the first argument is a duration then the return type isalso of type duration. Otherwise the return type is real.If the divisor is 0 (zero) then a DivideByZero exception is raised.
Examples Dividing a real by an integer in the expression 1.414 / 2 returns the real number 0.707. In this case the integer 2 is promoted to a real (2.0) before the division.Dividing a duration by a real in the expression $1 \mathrm{~m} 30 \mathrm{~s} / 2.0$ returns theduration 45 s
See also ..... - "div" on page 263

## Operators and Built-in Functions

## Operators

## ${ }^{\wedge}$ (exponentiate)

| Syntax | integer1 ^ integer2 |
| :---: | :---: |
|  | real ^ integer |
|  | Where: |
|  | The second argument is always an integer. If the second argument is not an integer then a TypeMismatch exception is raised. |
| Description | The result is the first argument raised to the power of the second argument. |
|  | The data type of the returned value is the same as the data type of the first argument. |
|  | If the first argument is integer then the second must be greater than or equal to 0 (zero). |
| Examples | Raising a real number to a power in the expression $1.414 \wedge 2$ returns the real number 1.999. |
|  | $2 \wedge 2$ results in the integer 4. |

## = and != (equality operators)

```
Syntax
boolean1 = boolean2
integer1 = integer2
real1 = real2
string1 = string2
oid1 = oid2
duration1 = duration2
time1 = time2
list1 = list2
token1 \(=\) token2
token1 \(=\) token2
Where:
If one of the arguments is duration and the other is integer or real, then the integer or real is promoted to a duration with a unit of seconds. That is, the real number 1.1 is promoted to a duration of 1.1 seconds.
Description The result is a boolean that is true if the two values are the same and false otherwise. String comparisons are done character by character, are case sensitive, and use the current character encoding collation sequence. Lists are compared element by element. Tuples are compared member by member. Oid types are compared numerically item by item.
The inequality operator in the expression \(\mathrm{a}!=\mathrm{b}\) is equivalent to not ( a = b).
A TypeMismatch exception is raised if \(=\) or \(!=\) is not defined for the data types being compared.
Examples Comparing an integer with a real as in the expression \(1=1.0\) returns true because the integer is promoted to a real before the comparison.
"cat" = "cat" returns true, but "cat" = "Cat" returns false because the comparison is case-sensitive.
See also • "Match.test" on page 293
```


## Operators and Built-in Functions

## Operators

## $\gg=\ll=$ (relational operators)

```
Syntax integer1 op integer2
reall op real2
string1 op string2
oid1 op oid2
duration1 op duration2
time1 op time2
token1 op token2
Where:
op is one of the following relational operators:
> greater than
>= greater than or equal to
< less than
<= less than or equal to
```

If one of the arguments is duration and the other is integer or real, then the integer or real is promoted to a duration with a unit of seconds. That is, the real number 1.1 is promoted to a duration of 1.1 seconds.

Description

Examples

The result is a boolean. String comparisons are done character by character, are case sensitive, and use the current character encoding collation sequence. A short string that is the prefix of a longer string is less than the longer string. Lists are compared element by element. Tuples are compared member by member. Tokens must be of the same type and are compared by ID code where the first ID code decl ared is less than ID codes that are declared later. Oid types are compared numerically item by item.

Comparing two integers in the expression $2>1$ returns the boolean result true and $2>2$ returns false because 2 is not greater than itself.
"cat" >= "cat" returns true, and "z" > "a" returns true but "z" > "A" returns false because the comparison is case-sensitive.
"cat" < "cats" returns true because "cat" is shorter than "cats".

## See also • "=and != (equality operators)" on page 243

- "Match.test" on page 293


# Operators and Built-in Functions 

## Operators

## :: (list constructor)

| Syntax | any : : list <br> Where: <br> any is an element of any data type that is to be prepended to the list. <br>  <br>  <br> list is the list to which the element any will be prepended. |
| :--- | :--- |
| Description | The returned result is list, with a new element any prepended to it. <br> The :: symbol is also used in binding patterns to bind a name to the first <br> element in a list. |
| Examples | $1::[2,3]$ results in the list $[1,2,3]$. <br> See also$\quad$- "List Data Type" on page 223 |
|  | - "List Manipulation" on page 327 |

## Built-in Functions

Built-in functions are those supplied with ECS. This section lists all built-in functions alphabetically and describes each in detail.
Although ECDL is not case-sensitive, most of the examples use lower-case consistently. Capitals are used only where it makes the text clearer.

Some functions, such as the string search functions Match.make, Match.test and Match.testVars are actually implemented in a separate module. However, you must use the full name as shown, including the dot, to reference these functions. See "M odules and Name Search" on page 356.

## Table 9-2 Summary of ECDL Built-in Functions

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| alter | Modifies the contents of a dictionary | "alter" on page 252 |
| and | True if both arguments are true | "and" on page 253 |
| append | Adds to correlation information to <br> an event | "append" on page 254 |
| bitand | The bitwise and operation on its <br> arguments | "bitand" on page 255 |
| bitinv | The bitwise inverse of the argument | "bitinv" on page 256 |
| bitleft | Shift bits left | "bitleft" on page 257 |
| bitright | Shift bits right | "bitright" on page 259 |
| bitor | True if either argument is true | "bitor" on page 258 |
| bitxor | The bitwise exclusive-or of the two <br> arguments | "bitxor" on page 260 |
| chr | Returns a character string of one <br> character with the value specified | "chr" on page 261 |
| datastore | Returns a value from the Data Store | "datastore" on page 262 |

## Operators and Built-in Functions

## Built-in Functions

## Table 9-2 Summary of ECDL Built-in Functions

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| div | Integer divide (see also the / <br> operator) | "div" on page 263 |
| explode | Returns a list containing the <br> individual characters in the string <br> supplied | "explode" on page 264 |
| explode_time | Returns a dictionary containing the <br> components of the time value <br> supplied | "explode_time" on page 265 |
| fact_exists | True if the specified fact is in the <br> Fact Store | "fact_exists" on page 267 |
| fact_find_left | Returns a list of matching left-parts <br> from the Fact Store | "fact_find_left" on page 268 |
| fact_find_right | Returns a list of matching right-parts <br> from the Fact Store | "fact_find_right" on page 269 |
| feed | Identifies the events that are to be <br> sent to other circuits | "feed" on page 270 |
| feedall | Feeds back events to all circuits <br> including source circuit. | "feedall" on page 271 |
| feedothers | Feeds back events to all circuits <br> other than source circuit | "feedothers" on page 272 |
| gen_ccall | Flushes the event | "flush" on page 273 |
| foldr | Iterates a function over the elements <br> of a list from left to right | "foldl" on page 274 |
| foldl | Iterates a function over the elements <br> of a list, from right to left | "foldr" on page 275 |
| Returns tuple that contains the status <br> of the call and the return value(s) of <br> the function | "gen_ccall" on page 276 |  |

## Table 9-2 Summary of ECDL Built-in Functions

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| gen_perlcall | Returns the tuple that contains the <br> status of the call and the return <br> value(s) of the perl function in the <br> form of a list | "gen_perlcall" on page 276 |
| implies | Returns true unless the first <br> argument is true and the second is <br> false | "implies" on page 276 |
| implode_time | Returns a time value based on the <br> contents of the dictionary supplied | "implode_time" on page 278 |
| Ip.fromOctet | Converts an IP address from a 4 byte <br> ASN.1 IpAddress representation to <br> its ASCII text form. | "Ip.fromOctet" on page 282 |
| Ip.toOctet | Converts an IP address from ASCII <br> text form to its 4 byte ASN.1 <br> IpAddress representation. | "Ip.toOctet" on page 283 |
| integerOf | Converts a string containing a <br> decimal number to an integer. | "integerOf" on page 280 |
| inter | Returns the intersection between two <br> lists | "inter" on page 280 |
| interc | Returns the intersection between two <br> lists using the supplied function to <br> perform the comparison | "interc" on page 281 |
| Match.test | Returns a string from a list of <br> characters | "join" on page 284 |
| join | Returns the length of a string or list | "length" on page 285 |
| Match.make | Compiles a search pattern for use <br> with the test or testVars function | "Match.make" on page 286 |
| Returns true if the supplied string <br> contains the specified pattern | "Match.test" on page 293 |  |
|  | Menth | \begin{tabular}{l}
\end{tabular} |

## Operators and Built-in Functions

## Built-in Functions

## Table 9-2 Summary of ECDL Built-in Functions

| Operator | Description | Section and Page |
| :---: | :---: | :---: |
| Match.testVars | Returns tags matching the search pattern from the supplied string | "Match.testVars" on page 294 |
| mod | Returns the first integer modulus the second integer | "mod" on page 296 |
| not | The logical inverse of the argument | "not" on page 296 |
| nth | Extracts an element from a list | "nth" on page 297 |
| oid. append | Appends an integer to the end of an oid and returns the resulting oid | "oid.append" on page 298 |
| oid.join | Appends a number of integers to form an oid and returns the resulting oid | "oid.join" on page 299 |
| oid.last | Returns part of the oid as an integer | "oid.last" on page 300 |
| oid.split | Returns a list of integers representing all the elements in an oid | "oid.split" on page 301 |
| oid.split_at | Splits an oid into two oids after the specified element, and returns the oids as a tuple | "oid.split_at" on page 302 |
| oid.split_nnm | Removes the last number from an oid and returns a tuple comprising the truncated oid and the number | "oid.split_nnm" on page 303 |
| or | True if either or both arguments are true | "or" on page 304 |
| ord | Returns the integer ASCII value of the single-character string passed | "ord" on page 304 |
| reverse | Reverses the order of the elements in a list | "reverse" on page 306 |
| round | Returns the nearest integer to the real number supplied | "round" on page 306 |

## Table 9-2 Summary of ECDL Built-in Functions

| Operator | Description | Section and Page |
| :--- | :--- | :--- |
| split | Splits a string into a list of substrings | "split" on page 307 |
| stringOf | Returns a string representation of the <br> supplied value | "stringOf" on page 308 |
| System. audit_log | Prints any specified data to the <br> engine log | "System.audit_log" on <br> page 309 |
| System. circuit_dump | Prints a snapshot of the engine to the <br> trace log | "System.circuit_dump" on <br> page 310 |
| System.trace | Prints text to the trace log | "System.trace" on page 311 |
| System.set_trace_mask | Sets the engine trace mask | "System.set_trace_mask" on <br> page 312 |
| Time.now | Returns the current time | "Time.now" on page 314 |
| truncate | Returns the nearest integer value <br> closer to zero | "truncate" on page 315 |
| union | Returns a list consisting of the union <br> of the two lists supplied | "union" on page 316 |
| unionc | Returns a list consisting of the union <br> of the two lists suppled using the <br> supplied function to perform <br> comparisons | "unionc" on page 317 |

## Operators and Built-in Functions

## Built-in Functions

## alter

Syntax event alter dictWhere:event is an event.dict is a dictionary (not an event) with key-value pairs of a type thatconforms to the types of key-value pairs in event.
Description
Example
See also

- "Dictionary Data Type" on page 225
- "Event Data Type" on page 226
- "Dictionary and Event Manipulation" on page 332
- "Create N ode" on page 87
- "M odify N ode" on page 105


## and

$\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { boolean1 and boolean2 } \\ \text { Where: }\end{array} \\ & \text { boolean1 and boolean2 are both booleans. }\end{array}$ Description $\left.\quad \begin{array}{l}\text { The result is the boolean value true if both arguments are true, } \\ \text { otherwise the result is false. } \\ \text { If the first argument is false then the operation short-circuits and the } \\ \text { second argument is not evaluated. This can have implications if you } \\ \text { require the second argument to be evaluated for its side-effects. }\end{array}\right\}$

## Operators and Built-in Functions

## Built-in Functions

## append

| Syntax | append <event $1><$ event $2><$ relation string> |
| :---: | :---: |
|  | Where: |
|  | event 1 and event 2 are both primitive events. |
| Description | The drill information for an event can be appended to an event from any node within the node's condition parameter. |
|  | The function takes the correlated event and correlating events and rel ation string as parameters.From above event 1 is the correlated event and event 2 is the correlating event. |
| Examples | append in an unless node may look like, |
|  | append input_event inhibitor_event "relation_suppress" |
| See also | - "flush" on page 273 |

## bitand

| Syntax | bitand Integer1 Integer2 <br> Where: <br>  <br>  <br> Integer1 and Integer2 are both integers. |
| :--- | :--- |
| Description | The result is the integer value of a bitwise and operation between the <br> two arguments. The arguments are treated as 32 bit unsigned bit <br> patterns. |
| Examples | bitand 7 o returns the integer 0. <br> bitand 71 returns the integer 1. |
| See also | - "bitinv" on page 256 |
|  | - "bitleft" on page 257 |
|  | - "bitor" on page 258 |
| - "bitright" on page 259 |  |
| - "bitxor" on page 260 |  |

## Operators and Built-in Functions

## Built-in Functions

## bitinv

| Syntax | bitinv integer |
| :---: | :---: |
|  | Where: |
|  | integer is an integer. |

Description The result is the integer value of a bitwise inversion of the argument. The argument is treated as a 32 bit unsigned bit pattern.

Examples bitinv 1 returns the integer - 2 .
See also - "bitand" on page 255

- "bitleft" on page 257
- "bitor" on page 258
- "bitright" on page 259
- "bitxor" on page 260


## bitleft

Syntax bitleft integer1 integer2Where:integer1 is an integer specifying the number of bits to shift.integer2 is the integer whose bits are to be shifted.
DescriptionExamplesbitleft 31 returns the integer 8.
See also - "bitand" on page 255

- "bitinv" on page 256- "bitor" on page 258- "bitright" on page 259
- "bitxor" on page 260


## Operators and Built-in Functions

## Built-in Functions

## bitor

Syntax $\quad$| bitor integer1 integer2 |
| :--- |
| Where: |
| integer1 and integer2 are both integers. |

Description The result is the integer value of a bitwise or operation between the two arguments. The arguments are treated as 32 bit unsigned bit patterns.

## Examples

bitor 70 returns the integer 7 .
bitor 71 returns the integer 7 .
bitor 81 returns the integer 9 .

## See also

- "bitand" on page 255
- "bitinv" on page 256
- "bitleft" on page 257
- "bitright" on page 259
- "bitxor" on page 260


## bitright

| Syntax | bitright integer1 integer2 |
| :---: | :---: |
|  | Where: |
|  | integer1 is an integer specifying the number of bits to shift. |
|  | integer2 is the integer whose bits are to be shifted. |
| Description | The result is the integer value of a bitwise right shift of integer2 by integerl bits. Both arguments are treated as 32 bit unsigned bit patterns. |
| Examples | bitright 38 returns the integer 1. |
| See also | - "bitand" on page 255 |
|  | - "bitinv" on page 256 |
|  | - "bitleft" on page 257 |
|  | - "bitor" on page 258 |
|  | - "bitxor" on page 260 |

## Operators and Built-in Functions

## Built-in Functions

## bitxor

| Syntax | bitxor integer1 integer2 |
| :---: | :---: |
|  | Where: |
|  | integer1 and integer2 are both integers. |
| Description | The result is the integer value of a bitwise exclusive or operation between the two arguments. The arguments are treated as 32 bit unsigned bit patterns. |
| Examples | bitxor 70 returns the integer 7 . |
|  | bitxor 71 returns the integer 6 . |
|  | bitxor 81 returns the integer 9 . |
| See also | - "bitand" on page 255 |
|  | - "bitinv" on page 256 |
|  | - "bitleft" on page 257 |
|  | - "bitor" on page 258 |
|  | - "bitright" on page 259 |

chr
Syntax chr integer
Where:
integer is the ASCII code of a character.
Description chr returns a string consisting of a single character. The character hasthe code of the integer passed. integer must be in the range $0<=$Integer $<=65536$ decimal. The translation is sensitive to the currentlocale settings.
Example For U.S. English, chr 97 returns the single character string "a".
See also - "String Data Type" on page 219

- "ord" on page 304


## Operators and Built-in Functions

## Built-in Functions

## datastore

| Syntax | datastore keyValue |
| :---: | :---: |
|  | Where: |
|  | keyValue can be any data of any data type. |
| Description | The only function that provides access to Data Store values is the datastore function. The datastore function takes a single parameter, the keyvalue, and returns the associated returnvalue from the Data Store. |
| Example | For example, if the following entry has been added to the Data Store: ADD DATA( "AlarmDelay", 30s ) |
|  | then the expression |
|  | datastore "AlarmDelay" |
|  | will return the duration 30s. The return type of the datastore function is always the same type as the value in the Data Store. |
| See also | - "Data Store" on page 186 |

```
div
\begin{tabular}{ll} 
Syntax & \begin{tabular}{l} 
int1 div int2 \\
Where:
\end{tabular} \\
int 1 is the integer dividend. \\
int2 is the integer divisor.
\end{tabular}
Description The div function divides int1 by int2 to produce an integer result.
Example }7\mathrm{ div 3 results in 2
See also • "Integer Data Type" on page 213
- "/ (real divide)" on page 241
```


## Operators and Built-in Functions

## Built-in Functions

## explode

| Syntax | explode string |
| :---: | :---: |
|  | Where: |
|  | string is a string to be exploded |
| Description | explode returns a list of one-character strings consisting of the letters in the argument string. |
| Examples | explode "hedley" returns the list of strings ["h","e","d","1,"e","y"]. |
|  | explode "" returns the empty list []. |
| See also | - "String Data Type" on page 219 |
|  | - "List Data Type" on page 223 |
|  | - "split" on page 307 |

## explode_time

```
Syntax explode_time time
Where:
time is a time data type to be exploded
explode_time returns a dictionary with the following predefined structure:
```

```
type TimeKey = token ( year,
```

type TimeKey = token ( year,
month,
month,
day,
day,
hours,
hours,
minutes,
minutes,
seconds,
seconds,
micro_seconds,
micro_seconds,
timezone,
timezone,
daylight
daylight
)
)
dict (Time.TimeKey, Integer|Real)

```
dict (Time.TimeKey, Integer|Real)
```

Description

Example

The time is represented in UTC. All values in the dictionary are integers except for:

- seconds, which is a Real value that can contain fractions of a second
- micro_seconds, which is always zero because it is no longer used (it has been kept for compatibility with ECS 2)
- timezone, which is always zero because the time is represented in UTC
- daylight, which is a Boolean value that is always false because the time is represented in UTC

The following example illustrates a function that returns true if an input event's creation time is between 9am and 5pm UTC.

```
let
    val etime = explode_time (input_event create_time)
    val hour = etime (Time.hours)
in
    (hour >= 9) and (hour <= 17)
end
```

The TimeKey component is prefaced by the module name "Time" in the form Time.hours.

Operators and Built-in Functions Built-in Functions

See also

- "Time Data Type" on page 217
- "implode_time" on page 278


## fact_exists

## Syntax

fact_exists fact
Where:
fact is a 3-tuple conforming to the type definition:
type Fact $=$ (any data, integer, any data)
By convention the data values are referred to as leftValue and rightValue, and the integer is referred to as the relation.

## Description

## Example

## See also

The fact_exists function checks if fact exists in the Fact Store. It returns the boolean data type true if the fact exists, otherwise it returns false.

When you use the fact_exists function, you must give the entire relationship as the argument, that is, leftValue, relation, and rightValue. For example, given the Fact Store:

```
("aaa", 1, "bbb")
("aaa", 1, "ccc")
("ddd", 1, "ccc")
("eee", 2, "fff")
("eee", 3, "fff")
```

we can write the fol lowing:

- fact_exists ("aaa", 1, "bbb") returns true
- fact_exists ("aaa", 3, "bbb") returns false
- fact_exists ("eee", 1, "bbb") returns false
- "Fact Store" on page 193
- "fact_find_left" on page 268
- "fact_find_right" on page 269

Operators and Built-in Functions

## Built-in Functions

## fact_find_left

## Syntax

Description

## Example

See also
fact_find_left factQuery
Where:
factQuery is a tuple conforming to the type definition:
type FactQuery = (any data, integer)
For the fact_find_left function the data value is the rightValue and the integer is the relation.

The fact_find_left function returns a list containing a copy of all leftValues that match the supplied rightValue and relation. An empty list is returned if there are no matching facts.

When you use the fact_find_left function, you provide the rightValue and the relation as arguments. For example, given the Fact Store:

```
("aaa", 1, "bbb")
("aaa", 1, "ccc")
("ddd", 1, "ccc")
("eee", 2, "fff")
("eee", 3, "fff")
```

we can write:

- fact_find_left ("ccc", 3) returns []
- fact_find_left ("fff", 2) returns ["eee"]
- fact_find_left ("ccc", 1) returns ["ddd", "aaa"]
- "Fact Store" on page 193
- "fact_exists" on page 267
- "fact_find_right" on page 269


## fact_find_right

## Syntax

fact_find_right factQuery
Where:
factQuery is a tuple conforming to the type definition:
type FactQuery $=$ (any data, integer)
For the fact_find_right function the data value is theleftValueand the integer is the relation.

## Description

The fact_find_right function returns a list containing a copy of all rightValues that match the supplied leftValue and relation. An empty list is returned if there are no matching facts.

## Example

When you use the fact_find_right function, you provide the leftValue and the relation as arguments. For example, given the Fact Store:

```
("aaa", 1, "bbb")
("aaa", 1, "ccc")
("ddd", 1, "ccc")
("eee", 2, "fff")
("eee", 3, "fff")
```

we can write:

- fact_find_right ("aaa", 2) returns []
- fact_find_right ("ddd", 1) returns ["ccc"]
- fact_find_right ("aaa", 1) returns ["bbb", "ccc"]

See also

- "Fact Store" on page 193
- "fact_exists" on page 267
- "fact_find_left" on page 268


## Operators and Built-in Functions

## Built-in Functions

## feed

feed <event> <flagToSrcCircuit>

Where: $\quad$\begin{tabular}{l}

- <event> is the event that is fed back into the engine <br>
Description <br>

| The feed () function is provided to identify events that will be sent to |
| :--- |
| other circuits. The feed ( ) API is used while configuring other nodes. | <br>


| The second argument to the feed () function is flagToSrcCircuit. If |
| :--- |
| the argument is false, the event is fed back to all circuits except the |
| source circuit. If the argument is "true" then the event is fed back to all |
| the circuits including the source circuit. |

\end{tabular}

WARNING When the parameter flagToSrcCiruit is set to "true", ensure that proper conditions are specified to avoid infinite feedback of events into the source circuit.
Example feed input_event false
See Also - "feedall" on page 271

- "feedothers" on page 272


## feedall

| Syntax | feedall <event> <br> Where: <br> <event> is the event that is fed back to the engine. |
| :--- | :--- |
| Description | The feedall function is used to send back events to all the circuits <br> including the source circuit in the same engine. The feedall function <br> works in the similar manner that the feed function works with the <br> flagToSrcCircuit set to "true". |

WARNING Proper conditions have to be specified to avoid infinite looping of events into the source cicuit.

Example feedall input_event
See Also • "feed" on page 270

- "feedothers" on page 272


# Operators and Built-in Functions 

## Built-in Functions

## feedothers

| Syntax | feedothers <event> <br> Where: |
| :--- | :--- |
| <event> is the event that is to be fed back into the engine. |  | Description $\quad$| The feedothers function is used to feed back events to all circuits |
| :--- |
| except the source circuit in the same engine.The feedothers function |
| works in the same manner that the feed function works with the |
| flagToSrcCircuit set to "false". |

flush
Syntax flush <event>
Where:
event is the event that is to be logged.append happening then the event maybe flushed.
Examples flush in a table node may look like,
flush current_event
See also - "append" on page 254Description If an event is going to be stored in the circuit forever or there can be no
WARNING In case of drill info being added after flush, duplicate record for the event could be created.

## Operators and Built-in Functions

## Built-in Functions

## fold

Syntax foldl func initial list

Where:
func is the function to iterate over the list elements and takes the form:
func arg1 arg2
arg1 will receive successive elements from the list, and arg2 is initially set to initial, and then the accumulated result returned from the previous iteration of func.
initial is the initial value applied to the accumulated result.
list is the list whose values are to be iterated over.
Description

Example You could define foldl (fn $x y=>(x+10 * y)) 0[3,5,7]$. This would proceed in steps like this:

1. 0 is assigned to the accumulated result for Step 2.
2. $(3+10 * 0)$ results in 3 , which is assigned to the intermediate result for Step 3.
3. $(5+10 * 3)$ results in 35 , which is assigned to the intermediate result for Step 4.
4. $(7+10 * 35)$ results in 357 , which is the final result, and the iteration ends.

- "Advanced Function Writing Features" on page 347
- "foldr" on page 275


## foldr

| Syntax | foldr func initial list |
| :---: | :---: |
|  | Where: |
|  | func is the function to iterate over the list elements and has the form: |
|  | func arg1 arg2 |
|  | where arg1 will recei ve successive elements from the list, and arg2 is initially set to initial, and then the accumulated result returned from the previous iteration. |
|  | initial is the initial value applied to the accumulated result. |
|  | list is the list whose values are to be iterated over. |
| Description | foldr makes it easy to scan a function over a list and accumulate a result. Contrast foldr with foldl which iterates list elements in the opposite direction. |
|  | The function func must be defined assuming that its first argument is an el ement of list and the second is the accumulated result, initially set to initial. Each iteration of func accumulates the result, until all elements of list are processed. The final result is the value returned by the last call of func. |
| Example | The expression foldr (fn $x$ y $=>(x+10 * y)) 0$ [3, 5, 7] would proceed in steps like this: |
|  | 1. 0 is assigned to the accumulated result for Step 2. |
|  | 2. $(7+10 * 0)$ results in 7 , which is assigned to the intermediate result for Step 3. |
|  | 3. $(5+10 * 7)$ results in 75 , which is assigned to the intermediate result for Step 4. |
|  | 4. $(3+10 * 75)$ results in 753 , which is the final result, and the iteration ends. |
| See also | - "List Manipulation" on page 327 |
|  | - "foldl" on page 274 |

## Operators and Built-in Functions

## Built-in Functions

## gen_ccall

| Syntax | gen_ccall ("libName, functionName, list) |
| :---: | :---: |
|  | Where: |
|  | libName is the name of the shared library in which the function resides of type string |
|  | functionName is the name of the function to call of type string |
|  | list is a list of parameters to be passed to the perl function |
| Description | The result is a tuple that contains the status of the call and the return value(s) of the function in the form of a list. |
| Examples | val (status, returnList) = gen_ccall("libraryName", "C_FunctionName", $[1,2,3])$. |
|  | if the function call succeeded then the status is zero and the return value(s) of the 'C'function is in list, returnList |
|  | gen_perlcall |
| Syntax | gen_perlcall (functionName, list) |
|  | Where: |
|  | functionName is the name of the function to call - it a string |
|  | list is the list of parameters to be passed to the perl function |
| Description | The result is a tuple that contains the status of the call and the return value(s) of the perl function in the form of a list. |
| Examples | ```val (status, returnList) = gen_perlcall("PerlFunctionName", [1,2,3])``` |
|  | if the function call succeeded then the status is zeroand the return value(s) of the perl function is in the list, returnList. |
|  | implies |
| Syntax | boolean1 implies boolean2 |

Where:
boolean1 and boolean2 are both booleans.

```
Description The result is the boolean value true unless boolean1 is true and
boolean2 is false, in which case the result is false. The following identity is always true:
boolean1 implies boolean2 \(=\) (not boolean1) or boolean2
If boolean 1 is false then the expression "short-circuits" returning true, and boolean2 is not evaluated.
Examples true implies false returns the boolean false. false implies false returns the boolean true. false implies true returns the boolean true.
To check if all events coming from Sydney are going to M elbourne:
```

```
forall ev in ev_list :-
```

forall ev in ev_list :-
(ev("source") = "Sydney")
(ev("source") = "Sydney")
implies
implies
(ev("dest") = "Melbourne")
(ev("dest") = "Melbourne")
See also

- "List Manipulation" on page 327
- "and" on page 253
- "or" on page 304

```

\section*{Operators and Built-in Functions}

\section*{Built-in Functions}

\section*{implode_time}

\section*{Syntax}

\section*{Description}
implode_time dict
Where dict is a time value dictionary with the structure defined below.
implode_time returns a value of type Time, based on the contents of the dictionary dict, which has the following predefined structure:
```

type TimeKey = token ( year,
month,
day,
hours,
minutes,
seconds,
micro_seconds
timezone,
daylight
)
dict (Time.TimeKey, Integer|Real)

```

All values in the dictionary are integers except for:
- seconds, which is a Real value that can contain fractions of a second.
- micro_seconds, which is always zero because it is no longer used (it has been kept for compatibility with ECS 2)
- timezone, which is a duration that is positive for time zones west of GMT. A timezone specified as an integer or real is interpreted as seconds.
- daylight, which is a Boolean value that is true if daylight saving is in effect.

All members of the dictionary are required, but timezone and/or daylight values can be omitted with the following effect:
\begin{tabular}{|l|l|l|}
\hline timezone & daylight & Effect \\
\hline absent & absent & \begin{tabular}{l} 
Assume local time and obtain the daylight \\
savings mode from the operating system.
\end{tabular} \\
\hline present & absent & \begin{tabular}{l} 
Use the specified \(t\) imezone value, and assume \\
a value of false for daylight.
\end{tabular} \\
\hline
\end{tabular}
\begin{tabular}{|l|l|l|}
\hline timezone & daylight & Effect \\
\hline absent & present & \begin{tabular}{l} 
Use local time and the specified daylight \\
value.
\end{tabular} \\
\hline
\end{tabular}
```

Example The following example converts the time 11:13:03.5 AM, 15 April 1990
EST-10 to a Time value:

```
```

val birthday = implode_time (

```
val birthday = implode_time (
    Time.year => 1990,
    Time.year => 1990,
    Time.month => 4,
    Time.month => 4,
    Time.day => 15,
    Time.day => 15,
    Time.hours => 11,
    Time.hours => 11,
    Time.minutes => 13,
    Time.minutes => 13,
    Time.seconds => 3.5,
    Time.seconds => 3.5,
    Time.timezone => -10h
    Time.timezone => -10h
    Time.daylight => true
    Time.daylight => true
    )
    )
See also
- "Time Data Type" on page 217
- "explode_time" on page 265
```


## Operators and Built-in Functions

## Built-in Functions

## integerOf

| Syntax | integerOf <string> |
| :---: | :---: |
|  | Where: |
|  | string can be any string containing a decimal number. |
| Description | The integerOf function converts a string containing a decimal number to an integer. |
| Example | integerOf "1234" returns 1234 |
|  | inter |
| Syntax | list1 inter list2 |
|  | Where: |
|  | list1 and list2 arelists containing elements of any data type. |
| Description | Returns a new list consisting of the intersection between list 1 and list2. Each of the elements of list1 and list2 are compared using the = (equality) operator to produce the list of elements common to both. Duplicate elements are removed. A TypeM ismatch exception is raised if $=$ is not defined for the data types of the elements in the lists. |
| Example | $[1,2,3]$ inter [ $2,3,4]$ returns the list [2,3]. |
|  | $[1,2,3]$ inter [4,5,6] returns the empty list []. |
|  | $[1,1,2,2,3]$ inter $[1,1,2,2,3]$ eliminates duplicates to return the list $[1,2,3]$. |
|  | [1, 1, 2, 2, 3] inter [] returns the empty list []. |
| See also | - "List Data Type" on page 223 |
|  | - "interc" on page 281 |
|  | - "union" on page 316 |
|  | - "unionc" on page 317 |

## interc

Syntax $\quad$ interc func list1 list2
Where:
list1 and list2 are lists containing elements of any data type. func is a function that returns a boolean data type and has the form:

```
func arg1 arg2
```

The return value must be true if the arg1 and arg2 are "the same", and false otherwise.

Description

Example The following example compares elements from two lists using the anonymous function $f n \mathrm{x} y=>\mathrm{x}=\mathrm{y}$ that simply uses the $=$ operator to compare each of the elements in the lists.

```
interc (fn x y => x = y) [1,2,3] [2,3,4]
```

The expression returns the list $[2,3]$. This is equivalent to writing $[1,2,3]$ inter $[2,3,4]$.

The next example uses a named function instead:

```
fun event_equals e1 e2 = e1(unique_Id) = e2(unique_Id)
val evlist_intersect = interc event_equals ev1 ev2
```

See also • "List Data Type" on page 223

- "interc" on page 281
- "union" on page 316
- "unionc" on page 317

Operators and Built-in Functions

## Built-in Functions

## Ip.fromOctet

|  | See also: Ip.toOctet |
| :--- | :--- |
| Syntax | Ip. fromOctet string <br> Where: |
|  | string is the IP address in 4 byte network address form. For |
|  | example, the address 122.48 .33 .46 would be $\backslash 172 \backslash 060 \backslash 041 \backslash 056$. |

## Ip.toOctet

SyntaxDescription
Example
See also

- "I p.fromOctet" on page 282


## Operators and Built-in Functions

## Built-in Functions

## join

\(\left.$$
\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { join listofstrings string } \\
\text { Where: }\end{array}
$$ <br>
\& listofStrings is a list of source strings to be joined <br>

\& string is a string of separator characters\end{array}\right]\)| join takes the strings in listofstrings and concatenates them into a |
| :--- |
| singlestring, separating each string from the next with the characters in |
| string. The returned result is a string. |

## length

Syntax length stringlength list
Where:
string is a (multibyte) string whose characters are to be counted.list is a list whose elements are to be counted.
Description length returns an integer representing the number of characters in thestring passed, or elements in the list, as appropriate.
Examples length "jabber" returns the integer 6.
length "" returns the integer 0.
length [1, 2, 3] returns the integer 3.
See also - "String Data Type" on page 219

- "List Data Type" on page ..... 223


## Operators and Built-in Functions

## Built-in Functions

## Match.make

SyntaxDescription Match.make returns an opaque data type that contains the compiledform of the search pattern. The compiled search is usually passeddirectly to the Match. test or Match.testVars function.
A two-stage compile and search makes the pattern matching mechanismmore efficient, particularly if the pattern compilation is done in a GlobalDefinition. Ultimately, the compiled opaque data type can only be passedto one of the test functions Match.test or Match.testVars.
Examples Match.make("cat", "", IgnoreCase) returns an opaque data type
See alsocontaining the compiled search expression.

```
Match.test( Match.make("cat", "", IgnoreCase)) "concatenate"
returns the boolean true.
Match.test( Match.make("cat", "", IgnoreCase)) "fred"
returns the boolean false.
```

- "=and !=(equality operators)" on page 243
- " $\gg=\ll=$ (relational operators)" on page 244
- "Match.test" on page 293
- "Match.testVars" on page 294


## Pattern-matching

ECS provides a powerful text pattern-matching language that allows logical testing for the existence of substrings and patterns. Parts of a text string can be extracted and assigned to tags, which may be reused within the same scope. This section describes the operators and syntax of the pattern-matching language.

The pattern-matching language used in the match functions is the same as that used in HP OpenView IT Operations.

Frequently, pattern-matching means simply scanning for a specific substring in the target string. For example, to search for the substring ERROR anywhere in the target string you search for the pattern:

```
"ERROR"
```

Similarly, should you wish to match text not containing a specific substring (for example, WARNING), you type:
" < ! [WARNING] >"
This uses the not operator "!", together with the chevrons "< >" that must enclose all operators, and the square brackets " [ ]" that isolate sub-patterns.

You control case-sensitivity with a separate argument to the Match.make function.

## Operators and Built-in Functions

## Built-in Functions

## Defining Match Expressions

## - Ordinary Characters

Ordinary characters generally represent themselves. However, if any of the following special characters are used they must be prefaced with a backslash escape character ( $\backslash$ ) to mask their usual function.

```
[ ] < > | ^ $
```

- Expression Anchoring Characters (^ and \$)

If the caret ( $\wedge$ ) is used as the first character of the pattern, only expressions discovered at the beginning of lines are matched. For example, "^ab" matches the string "ab" in the line "abcde", but not in the line "xabcde".

If the dollar sign is used as the last character of a pattern, only expressions at the end of lines are matched. For example, "de\$" matches "de" in the line "abcde", but not in the string "abcdex".

If ^ and \$ are not used as anchoring characters, that is, not as first or last characters, they are considered as ordinary characters without masking.

## - Expressions Matching Multiple Characters

Patterns used to match strings consisting of an arbitrary number of characters require one or more of the following expressions:

- <*> matches any string of zero or more characters (including separators)
$-<n \star>$ matches a string of $n$ arbitrary characters (including separators)
- <\#> matches a sequence of one or more digits
- <n\#> matches a number composed of $n$ digits
- <S> matches a sequence of one or more separator characters
- <nS> matches a string of $n$ separators
- <e> matches any string that contains no separator characters, in other words, a sequence of one or more non-separators; this can be used for matching words.
Separator characters are configurable for each pattern. By default, separators are the space and the tab characters. The separator string
is specified as the second element in the 3-tuple passed to the Match.make function.


## - Bracket ([ and ]) Expressions

The brackets ([ and ]) are used as delimiters to group expressions. To increase performance, brackets should be avoided wherever they are superfluous. In the pattern:

```
"ab[cd[ef]gh]"
```

all brackets are unnecessary-"abcdefgh" is equivalent.
Bracketed expressions are used frequently with the OR operator "|", the NOT operator "!" and when using sub-patterns to assign strings to tags.

- The OR (| ) Operator

Two expressions separated by the vertical bar character "|" matches a string that is matched by either expression. For example, the pattern:

```
"[ab|c]d"
```

matches the string "abd" and the string "cd".

## - The NOT (!) Operator

The not operator "!" must be used with delimiting square brackets, for example:

```
"<![WARNING]>"
```

The pattern above matches all text which does not contain the string "WARNING".

The not operator may al so be used with complex sub-patterns:

```
"LN<*>: R< ![490|[501[a|b]]] >-<*>"
```

The above pattern makes it possible to generate a message for any line connection other than from repeaters 490, 501a or 501b.
Therefore, the following would be matched:

```
"LN270: R300-427"
```

However, this string is not matched, because it refers to repeater 501a:

```
"LN270: R501a-800"
```

If the sub-pattern including the not operator does not find a match,

## Built-in Functions

the not operator behaves like a <*>: it matches zero or more arbitrary characters. For this reason, there is a difference between the UNIX expression " [!123] ", and the corresponding ECS pattern matching expression: "<! [1|2|3]>". The ECS expression matches any character or any number of characters, except 1,2 , or 3 ; the UNIX expression matches any one character, except 1, 2 , or 3.

- The Mask ( $\backslash$ ) Operator

The backslash " $\backslash$ " is used to mask the special meaning of the characters:
[ ] < > | ^ \$
A special character preceded by $\backslash$ results in an expression that matches the special character itself.

Because ^ and \$ only have special meaning when placed at the beginning and end of a pattern respectively, you need not mask them when they are used within the pattern (in other words, not at beginning or end).

The only exception to this rule is the tab character, which is specified by entering " $\backslash t$ " into the pattern string.

## Tags

Search patterns may use tags to identify part(s) of the target string to, for example, compose a new string from selected parts of the target string. To define a tag, add ".tagname" before the closing chevron. The pattern:

```
^errno: <#.number> - <*.error_text>
```

matches a string such as:
errno: 125 - device not in service
and assigns "125" to the tag number and "device not in service" to the tag error_text. The tags may be accessed as members of a dictionary. See "Match.testVars" on page 293.

## Assignment Rules

In matching the pattern "<*. tag1><*. tag2>" against the string "abcdef", it is not immediately clear which substring of the input string is assigned to each tag. For example, it is possible to assign an empty string to tag1 and the whole input string to tag2, as well as assigning

```
"a" to tag1 and "bcdef" to tag2, and so forth.
```

The pattern-matching algorithm always scans both the input line and the pattern definition (including alternative expressions) from left to right. <*> expressions are assigned as few characters as possible. <\#>, <@>, <S> expressions are assigned as many characters as possible. Therefore, tag1 will be assigned an empty string in the above example.

To match an input string such as:

```
"this is error 100: big problem"
```

use a pattern such as:

```
error <#.errnumber>:<*.errtext>
```

In which:

- " 100 " is assigned to the tag errnumber
- "big problem" is assigned to the tag errtext

For performance and pattern readability purposes, you can specify a delimiting substring between two expressions. In the above example, ":" is used to delimit <\#> and <*>.

Matching <@.word><\#.num> against "abc123" assigns "abc12" to word and "3" to num, as digits are permitted for both <\#> and <@>, and the left expression takes as many characters as possible.

Patterns without expression anchoring can match any substring within the input line. Therefore, patterns such as:
"this is number<\#.num>"
are treated in the same way as:

```
"<*>this is number<#.num><*>"
```


## Sub-Patterns Assignment

In addition to being able to use a single operator, such as * or \#, to assign a string to a tag, you can also build up a complex sub-pattern composed of a number of operators, according to the following pattern:
<[sub-pattern].tag>
For instance: < [rack<\#>.brd<\#>] .hware>
In the example above, the period ( . ) between rack<\#> and brd<\#> matches a similar dot character, while the dot between ] and hware is necessary syntax. This pattern would match a string such as

## Operators and Built-in Functions

## Built-in Functions

"rack123.brd47" and assigns the complete string to hware.
Other examples of sub-patterns are:
<[Error|Warning].sev>
and
<[Error[<\#.n><*.msg>]]. complete>
In the first example above, any line with either the word "Error" or the word "Warning" is assigned to the tag, sev. In the second example, any line containing the word "Error" has the error number assigned to the tag, n, and any further text assigned to msg. Finally, both number and text are assigned to complete.

## Examples of Pattern-matching Conditions

The following examples show some of the many ways in which the ECS pattern-matching language can be used.

- "Error"

Recognizes any message containing the keyword Error at any place in the message, when ExactCase is specified.

- "panic"

Matches all messages containing panic, Panic, PANIC at any place in the text, when I gnoreCase is specified.

- "logon|logoff"

Uses the or operator to recognize any message containing the keyword logon or logoff.

- "^switch:<*.msg> errno<*><\#.errnum>\$"

Recognizes any message such as:
"switch: lost service errno : 6"
or
"switch: service unavailable; errno 16"
In the first example, the string "lost service errno" is assigned to the tag msg. The digit 6 is assigned to the tag errnum. N ote the way that the anchoring symbol is used to specify that the digit 6 will only be matched if it is at the end of the line.

- "^errno[ |=]<\#.errnum> <*.errtext>"

Matches strings such as:

```
"errno 6 - no such device or address "
or
"errno=12 not enough capacity. "
```

N ote the space before the or operator. The expression in square brackets matches either this blank space, or the equals sign. The space between <\# . errnum> and <*. errtext> is used as a delimiter. Although not strictly required for assignments to the tags shown here, this space serves to increase performance.

```
- "^system:<*>:<*.id>:"
```

Matches a line delimited by colons such as:
"system:abc123:\#103.79a:270295114730:"
and returns the third field in tag id. The colon ": "in the middle of the pattern is used to delimit the string passed to id from the preceding string. The colon at the end of the pattern delimits the string passed to id from the succeeding field in the input pattern. Here the colon is necessary to separate the strings, not merely to enhance efficiency.

- ^Warning:<*.text>on circuit<@.circuit>\$

Matches any message such as:
"Warning: too many errors on circuit 473-186"
and assigns "too many errors" to text, and "473-186" to circuit.

## Match.test

```
Match.test opaque string
```

Where:
opaque is a compiled search pattern, as provided by the Match.make function.
string is the string to be searched for the pattern.
Match functions do not support strings that contain the NUL character ( $\backslash 000$ ). If any of the string arguments to a match function contain NUL then the result is undefined.

## Operators and Built-in Functions

## Built-in Functions

| Description | Match.test returns a boolean value indicating whether the pattern was found in the string string. The value true is returned if the pattern represented by the opaque argument is present in string. Otherwise, the value false is returned. The pattern being searched for must first be compiled by the Match. make function before being passed to Match.test as the opaque argument. Any tags in the pattern are ignored. |
| :---: | :---: |
| Examples | For example, the following ECDL fragment returns true because the pattern "cat" appears in the string "concatenate": |
|  | let |
|  | Val pat = Match.make ( "cat","",IgnoreCase ) |
|  | in |
|  | Match.test pat "concatenate" |
|  | end |
| See also | - "String Data Type" on page 219 |
|  | - "=and != (equality operators)" on page 243 |
|  | - " $\gg=\ll=$ (relational operators)" on page 244 |
|  | - "Match.test" on page 293 |
|  | - "Match.testVars" on page 294 |
|  | Match.testVars |
| Syntax | Match.testVars opaque string |
|  | Where: |
|  | opaque is a compiled search pattern, as provided by the Match.make function. |
|  | string is the string to be searched for the pattern. |
|  | Match functions do not support strings that contain the NUL character ( $\backslash 000$ ). If any of the string arguments to a match function contain NUL then the result is undefined. |
| Description | Match.testVars returns a tuple containing a bool ean and dictionary. The boolean value indicates whether the pattern was found in the string |

string. The dictionary contains tags contained in the pattern, together with the value of each, extracted from the string. The dictionary is empty if no match is found. The pattern being searched for must first be compiled by the Match. make function into an opaque data type before being passed to Match. test as the opaque argument. The data type of the return value is defined as:

```
(Boolean * Dict(String, String))
```


## Examples <br> The following example extracts digits from the end of strings matching

See also the pattern in pat:

```
fun test1 (result) =
let
    val pat = Match.make("xyz<*.text>D<#.digits>$", "",
                ExactCase)
    val (succ, vars) = Match.testVars pat "xyzjunkD6"
    val digits =
        if succ
        then vars "digits"
        else raise NotFound
        end
in
    digits :: result
end
```

- " $\gg=\ll=$ (relational operators)" on page 244
- "Match.test" on page 293
- "Match.testVars" on page 294


## Operators and Built-in Functions

## Built-in Functions

## mod

| Syntax | integer1 mod integer2 |
| :---: | :---: |
|  | Where: |
|  | integer1 and integer2 are both integers. |
| Description | The result is the integer value of the remainder after dividing Integer1 by Integer2. |
| Examples | 7 mod 3 returns the integer 1. |
|  | $1 \bmod 1$ returns the integer 0 . |
|  | $7 \mathrm{mod}(-3)$ returns the integer 1. |
|  | $(-7) \bmod 3$ returns the integer -1 |
| See also | - "/ (real divide)" on page 241 |
|  | - "div" on page 263 |
|  | not |
| Syntax | not boolean |
| Description | The result is the boolean value which is the logical inverse of the argument. |
| Examples | not true returns the boolean false. |
|  | not false returns the boolean true. |
| See also | - "Boolean Data Type" on page 215 |

## nth

\(\left.$$
\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { nth integer list } \\
\text { Where: }\end{array}
$$ <br>
<br>
integer is the number of the element to be returned. <br>

\& list is the list from which the element is to be extracted.\end{array}\right\}\)| nth returns a single element from the list. The first element in the list is |
| :--- |
| element number one. The data type of the result is the data type of the |
| returned element. |
| If there is no such element because integer is less than 1 or greater |
| than the number of elements in the list, then a NotFound exception is |
| raised. |
| Example |
| nth $1 \quad[1,2,3]$ returns the integer 1. |

## Operators and Built-in Functions

## Built-in Functions

## oid.append

Syntax Oid.append oid number
Description
Where:
oid is an oid data type.
number is an integer to be appended to the end of the oid.
ExamplesOid. append appends the integer number to the end of the objectidentifier oid and returns the resulting oid.
See also - "oid.join" on page 299

- "oid.last" on page 300
- "oid.split" on page 301
- "oid.split_at" on page 302
- "oid.split_nnm" on page 303


## oid.join

## Syntax <br> Description

Examples

Oid.join listOfIntegers
Where:
listOfIntegers is a list data type containing the integers to be assembled into an oid.

Oid.join returns an oid formed from appending all the integers together. This is the inverse of the Oid. split function.

Oid.join $[1,3,6,1,4,1,11,2,17,1,58916865]$ evaluates to the oid 1.3.6.1.4.1.11.2.17.1.58916865.

Oid.join ( Oid.split myOid ) evaluates to myOid. Oid. join is the inverse of Oid.split.

Oid. join [5] evaluates to the oid 5 (note that this is an oid data type and not an integer). This is a useful way to obtain oids with two or less components. Normally, you cannot write a literal oid with less than three parts because the ECDL compiler interprets it as a real or an integer. For example, " 1.2 " is interpreted as a real and " 1 " is interpreted as an integer.

Oid. join [] evaluates to an oid with no numbers. This is probably only useful as an argument to other Oid. functions.

See also

- "oid.append" on page 298
- "oid.last" on page 300
- "oid.split" on page 301
- "oid.split_at" on page 302
- "oid.split_nnm" on page 303


# Operators and Built-in Functions 

## Built-in Functions

## oid.last

| Syntax | Oid.last oid |
| :---: | :---: |
|  | Where: |
|  | oid is an oid data type. |
| Description | Oid.last returns the last part of the oid as an integer. |
| Examples | Oid.last (1.3.6.1.4.1.11.2.17.1.58916865) evaluates to the integer 58916865. |
| See also | - "oid.append" on page 298 |
|  | - "oid.join" on page 299 |
|  | - "oid.split" on page 301 |
|  | - "oid.split_at" on page 302 |
|  | - "oid.split_nnm" on page 303 |

## oid.split

Syntax Oid.split oid
Where:
oid is an oid data type.
Description Oid.split returns a list of integers representing all the elements in theoid.
Examples Oid.split 1.3.6.1.4.1.11.2.17.1.58916865 evaluates to the list [1, 3, 6, 1, 4, 1, 11, 2, 17, 1, 58916865]Oid.split ( Oid.join myListOfIntegers ) evaluates tomyListOfIntegers. Oid.split is the inverse of Oid. join.
See also - "oid.append" on page 298- "oid.join" on page 299- "oid.last" on page 300- "oid.split_at" on page 302- "oid.split_nnm" on page 303

Operators and Built-in Functions

## Built-in Functions

## oid.split_at

| Syntax | Oid.split_at index oid |
| :---: | :---: |
|  | Where: |
|  | index is an integer that determines where the oid is split. index must be positive. An InvalidArgs exception is raised if it is less than 0. |
|  | oid is an oid data type. |
| Description | Oid.split_at splits the object identifier oid into two oids after the indexth element. The two oids are returned as a tuple. |
| Examples | ```Oid.split_at 4 1.3.6.1.4.1.11.2.17.1 returns (1.3.6.1, 4.1.11.2.17.1)``` |
|  | Oid.split_at 0 1.3.6.1.4.1.11.2.17.1 returns ( ?, 1.3.6.1.4.1.11.2.17.1) where ? represents the "unwritable" Oid with no parts |
|  | Oid.split_at 10 1.3.6.1.4.1.11.2.17.1 returns (1.3.6.1.4.1.11.2.17.1, ?) where ? represents the "unwritable" Oid with no parts |
| See also | - "oid.append" on page 298 |
|  | - "oid.join" on page 299 |
|  | - "oid.last" on page 300 |
|  | - "oid.split" on page 301 |
|  | - "oid.split_nnm" on page 303 |

## oid.split_nnm

Syntax Oid.split_nnm oidWhere:
oid is an oid data type.

Description

Examples

See also

Oid.split_nnm removes the last number from the oid and returns a tuple comprising the truncated oid and the number. It will raise the InvalidArgs exception if the oid has no parts.

The following contrived example for a Filter node Condition parameter splits the supplied NNM oid and prints the enterprise object identifier and specific trap number to the trace log:

```
let
    val (enterprise, specificTrap) = Oid.split_nnm
        1.3.6.1.4.1.11.2.17.1.58916865
in
        system.trace (StringOf enterprise) ;
        system.trace (StringOf specificTrap) ;
        true
end
results in the following lines in the trace log:
TRACE [interpreter]: 1.3.6.1.4.1.11.2.17.1
TRACE [interpreter]: 58916865
```


## Operators and Built-in Functions

## Built-in Functions

## Or

\(\left.$$
\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { boolean1 or boolean2 } \\
\text { Where: }\end{array}
$$ <br>

boolean1 and boolean2 are both booleans.\end{array}\right]\)| The result is the bool ean value true if either or both arguments are true, |
| :--- |
| otherwise the result is false. |
| If boolean1 is true then the function "short-circuits" and boolean2 is not |
| evaluated. This may have implications if boolean2has side effects. |

## perl_interp

```
Syntax perl_interp function-name
Where:
    function-name is the user defined function written in Perl. The
perl_interp function is used to access external data and perform
correlation based on the data obtained.
Description The perl_interp function calls a user defined Perl function.
Example
```

```
val perl_ret =
```

val perl_ret =
perl_interp("perl_func_name",arg1,arg2,arg3....,argn)
perl_interp("perl_func_name",arg1,arg2,arg3....,argn)
perl_ret is a list of EV_AttrValues, where individual perl_ret values can be accessed by

```

\title{
Operators and Built-in Functions
}

Built-in Functions

\section*{reverse}
\begin{tabular}{|c|c|}
\hline Syntax & reverse list \\
\hline & Where: \\
\hline & list is a list of any data types. \\
\hline Description & reverse returns a new list with same elements in reverse order. \\
\hline Example & reverse [1, 2, 3] returns the list [3,2,1]. \\
\hline See also & - "List Data Type" on page 223 \\
\hline & round \\
\hline Syntax & round real \\
\hline & Where: \\
\hline & real is the real number to be rounded. \\
\hline Description & The result is the nearest integer value, where .5 rounds away from 0 The return data type is integer. \\
\hline Examples & \begin{tabular}{l}
round 1.5 returns the integer 2 . \\
round (-1.5) returns the integer -2 . \\
round 10.3 returns the integer 10 .
\end{tabular} \\
\hline See also & \begin{tabular}{l}
- "Real Data Type" on page 214 \\
- "truncate" on page 315
\end{tabular} \\
\hline
\end{tabular}

\section*{split}
Syntax split string1 string2Where:
string1 is the source (multibyte) string to be splitstring2 is the (multibyte) substring to search for
Description
Examplessplit "h-a-t" "-"returns the list ["h", "a", "t"].split "splatter" "plat" returns ["s", "ter"]split "jabber" "z" returns ["jabber"] because no split pointsubstring was found.
See also - "String Data Type" on page 219
- "List Data Type" on page ..... 223
- "explode" on page 264
- "join" on page 284

\section*{Operators and Built-in Functions}

\section*{Built-in Functions}

\section*{stringOf}
\begin{tabular}{ll} 
Syntax & \begin{tabular}{l} 
stringof anydata \\
Where:
\end{tabular} \\
anydata can be any of the types listed in the table bel ow.
\end{tabular}
\begin{tabular}{|l|l|}
\hline Type & Example of string returned by stringOf function \\
\hline Void & () \\
\hline Integer & 123 \\
\hline Boolean & true \\
\hline Real & 4.7 \\
\hline String & "abc" \\
\hline Time & \begin{tabular}{l} 
yyyymmddhhmmss.usecsZ e.g. 19960101000000.0Z for 1 Jan \\
1996
\end{tabular} \\
\hline Duration & 2000 s for 33m 20s \\
\hline Oid & 4.6 .8 .0 \\
\hline Event & event(0x...) \\
\hline Opaque & Opaque(0x...) \\
\hline Tuple & (3, 5, false) \\
\hline List & [3, 5, false], lists of lists are properly represented. \\
\hline Token & token (2) (3) \\
\hline
\end{tabular}

\section*{System.audit_log}
\(\left.\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { System.audit_log anydata } \\
\text { Where: }\end{array} \\
\text { anydata is an ECDL expression that evaluates to any data type } \\
\text { (anything except a function type). }\end{array}\right]\)\begin{tabular}{l} 
The System. audit_log function converts anydata to printable data and \\
writes it to the engine log at a log mask level of LoG_INFo. The return \\
value is Void.
\end{tabular}

\section*{Operators and Built-in Functions}

\section*{Built-in Functions}

\section*{System.circuit_dump}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Syntax} & System.circuit_dump () \\
\hline & Where: \\
\hline & () is a Void dummy argument that must be passed. \\
\hline \multirow[t]{3}{*}{Description} & The system. circuit_dump function is equival ent to the ecsmgr -snapshot command except that the engine snapshot is written to the trace log file. \\
\hline & The system. circuit_dump function returns Void. \\
\hline & A snapshot is a static dump of an engine's state and should only be performed at the request of support personnel. Depending upon the nature of the circuit and the number of events detained or stored in the circuit, a snapshot may produce a very large quantity of data. This function may be used in an appropriately configured sequence expression, in any correlation node parameter. \\
\hline \multirow[t]{2}{*}{Example} & In a Filter node condition parameter a snapshot may be taken before the condition is evaluated with: \\
\hline & (System.circuit_dump () ; input_event "deviceId" = "thor" ) \\
\hline \multirow[t]{2}{*}{See also} & - "Sequence" on page 339 \\
\hline & - ecsmgr \((1 \mathrm{M})\) command referenc \\
\hline
\end{tabular}

\section*{System.trace}
\(\left.\left.\begin{array}{ll}\text { Syntax } & \begin{array}{l}\text { System.trace string } \\
\text { Where: }\end{array} \\
& \text { string is any expression that evaluates to a String. }\end{array}\right] \begin{array}{l}\text { The system.trace function writes string to the trace log and returns } \\
\text { Void. The string may be literal text or an expression that evaluates to a } \\
\text { String data type. This function may be use in an appropriately } \\
\text { configured sequence expression, in any correlation node parameter. }\end{array}\right\}\)\begin{tabular}{l} 
The trace mask must be set to enable the System. trace log level using \\
either System. set_trace_mask or ecsmgr -trace.
\end{tabular}

\section*{Operators and Built-in Functions}

\section*{Built-in Functions}

\section*{System.set_trace_mask}
\begin{tabular}{ll} 
Syntax & System.set_trace_mask integer \\
Where:
\end{tabular}
integer is the trace log mask value.
Description The system. set_trace_mask function sets the trace log mask to the value of the integer argument expression and returns the previous trace mask as an Integer value. This function may be used in an appropriately configured sequence expression, in any correlation node parameter.
The trace mask values shown in Table 8-3 should be summed to enabled multiple log levels. Note that trace mask values must be expressed in decimal notation; ECDL does not support hexadecimal.
Table 9-3 Trace Mask Values
\begin{tabular}{|l|l|}
\hline \begin{tabular}{l} 
Decimal \\
value
\end{tabular} & Description \\
\hline 1 & Event transfer \\
\hline 2 & Event discard \\
\hline 4 & Event create \\
\hline 8 & Internal debugging (user usage limited) \\
\hline 16 & Create/delete of nodes \\
\hline 64 & Entry or exit of events to nodes \\
\hline 128 & (unused) \\
\hline 256 & Memory allocation failure (user usage limited) \\
\hline 512 & Processing information, OPI debugging \\
\hline 1024 & Circuit load, transit delays \\
\hline 2048 & Invariant fail (user usage limited) \\
\hline 4096 & Interpreter tracing, exception messages \\
\hline 8192 & Interpreter detail (user usage limited) \\
\hline
\end{tabular}

\section*{Table 9-3 Trace Mask Values}
\begin{tabular}{|l|l|}
\hline \begin{tabular}{l} 
Decimal \\
value
\end{tabular} & Description \\
\hline 16384 & Event delete \\
\hline 32768 & Management, ecsmgr and engine \\
\hline 65536 & system.trace \\
\hline
\end{tabular}
```

Example
In a Filter node Condition parameter, to set the trace mask to enable event transfer, event discard and event create (using a sequence expression):
(
System.set_trace_mask (1 + 2 + 4) ;
input_event "severity" = critical
)

```
See also

To set the trace mask as above, and to enable subsequent use of the system.trace function:
```

(
System.set_trace_mask (1 + 2 + 4 + 65536) ;
input_event "severity" = critical
)

```

See also
- "Sequence" on page 339
- "System.trace" on page 311
- ecsmgr(1M) command reference

Operators and Built-in Functions

\section*{Built-in Functions}

\section*{Time.now}
Syntax Time.now ()
Where:
() is a void value (dummy parameter) that must be passed.
Description The result is a Time data type value containing the current time.
ExamplesThe following example sets the "createTime" attribute of a newly createdevent to the current time. Note that explode_time is used to deconstructthe Time data type value into its component parts:
```

let
val t = explode_time (Time.now ())
in
created_event alter
(
"createTime.date.year" => t Time.year,
"createTime.date.month" => t Time.month,
"createTime.date.day" => t Time.day,
"createTime.time.hour" => t Time.hours,
"createTime.time.minute" => t Time.minutes,
"createTime.time.second" => t Time.seconds,
"deviceId" => "Ephor",
"messageType" => "LNKUP",
"severity" => 2,
"text" => ""
)
end

```
See also
- "Time Data Type" on page 217
- "explode_time" on page 265

\section*{truncate}
\begin{tabular}{ll} 
Syntax & \begin{tabular}{l} 
truncate real \\
Where:
\end{tabular} \\
& real is the real number to be truncated.
\end{tabular}\(\quad\)\begin{tabular}{l} 
The result is the nearest integer value closer to 0. \\
Description \\
The return data type is integer.
\end{tabular}

\section*{Operators and Built-in Functions}

\section*{Built-in Functions}

\section*{union}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Syntax} & list1 union list2 \\
\hline & Where: \\
\hline & list1 and list2 are lists containing elements of any data type for which the equality operator \(=\) is defined. \\
\hline Description & Returns a new list consisting of the union of list1 and list2. The result is a list of all elements from both lists, except that duplicate elements are eliminated. \\
\hline \multirow[t]{4}{*}{Example} & \([1,2,3]\) union \([2,3,4]\) returns the list [1,2,3,4]. Duplicates are eliminated from the result. \\
\hline & \([1,2,3]\) union [4,5,6] returns the list [ \(1,2,3,4,5,6]\) \\
\hline & \([1,1,2,2,3]\) union \([1,1,2,2,3]\) eliminates duplicates to return the list \([1,2,3]\). \\
\hline & [1, 1, 2, 2, 3] union [] returns thelist [1, 2, 3]. \\
\hline \multirow[t]{4}{*}{See also} & - "List Data Type" on page 223 \\
\hline & - "inter" on page 280 \\
\hline & - "interc" on page 281 \\
\hline & - "unionc" on page 317 \\
\hline
\end{tabular}

\section*{unionc}
\begin{tabular}{|c|c|}
\hline \multirow[t]{6}{*}{Syntax} & unionc func list1 list2 \\
\hline & Where: \\
\hline & list1 and list2 arelists containing elements of any data type. \\
\hline & func is a function that returns a boolean data type and has the form: \\
\hline & func arg1 arg2 \\
\hline & The return value must be true if the arg1 and arg2 are "the same", and false otherwise. \\
\hline \multirow[t]{3}{*}{Description} & Returns a new list consisting of the union of list1 and list2. Each of the elements of list1 and list2 are compared, using the function func supplied as the first argument. The result is a list of all elements from both lists, except that duplicate elements are eliminated. \\
\hline & The unionc function is similar to the union function except that union is in prefix form instead of infix, and the comparison between el ements is performed using a user supplied function instead of the = operator. \\
\hline & The unionc function is most useful when manipulating lists of elements whose data types do not have an equality operation defined, or when the required comparison is something other than a simple test for equality. \\
\hline \multirow[t]{3}{*}{Example} & The following example compares elements from the two lists using the anonymous function \(f n \mathrm{x} y=>\mathrm{x}=\mathrm{y}\) that simply uses the \(=\) operator to compare each of the elements in the lists. \\
\hline & unionc ( \(f\) n x y \(=>\mathrm{x}=\mathrm{y}\) ) [1,2,3] [2,3,4] \\
\hline & The expression returns the list \([1,2,3,4]\). This is equivalent to writing [1,2,3] union [2,3,4]. \\
\hline \multirow[t]{4}{*}{See also} & - "List Data Type" on page 223 \\
\hline & - "inter" on page 280 \\
\hline & - "interc" on page 281 \\
\hline & - "union" on page 316 \\
\hline
\end{tabular}

Operators and Built-in Functions
Built-in Functions

\section*{10 \\ Writing ECDL Expressions}

This chapter contains a detailed description of the Event Correlation Description Language (ECDL)—the language used to configure nodes in a correlation circuit. Typically, elements of the language are used:
- when filling in parameters of nodes (condition parameter in particular)
- in the "Global Definitions" of a circuit
- in the Filter Condition expression on the External tab

\section*{NOTE \\ You cannot use ECDL expressions in Data Store or Fact Store entries because no expression evaluation is done when these stores are loaded, updated or accessed.}

The main sections in this chapter are:
- "Binding and Comparisons" on page 321
- "Comments in ECDL" on page 325
- "Data Type Handling" on page 326
- "Flow-control Expressions" on page 334
- "Exceptions" on page 341
- "F unctions and Language Layout" on page 344
- "M odules and Name Search" on page 356

If you have a good working knowledge of the C language then the last section is specifically for you:
- "An Introduction to ECDL for C Programmers" on page 370

\section*{Binding and Comparisons}

ECDL binding patterns assign names to values. Without the use of binding patterns a lot of event data manipulations and comparisons would be difficult to perform.

\section*{Named Values}

ECDL provides a mechanism for assigning names to values. This is similar to the concept of variables in some languages but in ECDL, once a name is assigned to a value, that value cannot be changed. These are known as named values.

For example,
```

val myValue = 23

```

Here, the keyword val represents a value definition, myValue is the name, and 23 is the value that myValue represents. The value that myValue represents is now unchangeable. The named value myValue can now be used anywhere where an integer number is expected.
For example:
```

val myOtherValue = 2 * myValue + 3

```

Neither of these two examples specifies the data types of the named values. ECDL infers that the data type is Integer in both cases. However, it is sometimes desirable to specify the data type explicitly, particularly in function application.
To specify a data type, simply follow the named value with a colon (:), followed by the type expression. For example:
```

func plus2 (num : Integer) : Integer = num + 2

```

\section*{Binding Patterns}

The previous chapter described complex data types associated with the manipulation of events. To help manage these complex expressions, ECDL has a mechanism known as binding patterns. Binding patterns enable named values to be used for all or part of a complex ECDL data value. This mechanism allows for more than one name to be bound simultaneously to parts of a complex data value.

\section*{Writing ECDL Expressions \\ Binding and Comparisons}

Where an ECDL value is a Tuple (as with the ASN. 1 CHOICE name and value) for example:
("number", 5)
Typically, you need to have access to the value ( 5 in this case), as a simple ECDL value. By using binding patterns we can write:
```

val (choiceName, myValue) = ("number", 5)

```

This now provides the named values choiceName (representing "number") and myValue (representing 5).

The wildcard pattern _ (underscore) can be used for elements in the binding pattern whose value is of no interest. In the previous example, if you were not interested in the CHOICE name, say, you could write:
```

val (_, myValue) = ("number", 5)

```

In this case, we have only myValue as a named value, which can now be used in subsequent expressions such as:
myValue > 3
Table 10-1 Binding Pattern Forms
\begin{tabular}{|l|l|}
\hline Pattern & Description \\
\hline identifier & \begin{tabular}{l} 
An identifier matches any value and binds the identifier to the value. All of the \\
identifiers in a binding pattern must be different.
\end{tabular} \\
\hline identifier:type & \begin{tabular}{l} 
This binds the identifier as above and also says the type of the value must \\
conform to the given type (compile time check only).
\end{tabular} \\
\hline- & \begin{tabular}{l} 
The underscore character is the wildcard pattern. It will match with any value but \\
does not bind a name to the value.
\end{tabular} \\
\hline constant & \begin{tabular}{l} 
A pattern can be any constant including the void value () and token names. The \\
constant pattern matches if the value is equal to the constant. This form is mainly \\
used in case expressions. For example: \\
case severity of low 1 | medium 2 | high 3 \\
returns the value 1, 2, or 3 depending on whether the severity is low, medium, or \\
high.
\end{tabular} \\
\hline
\end{tabular}

\section*{Table 10-1 Binding Pattern Forms}
\begin{tabular}{|c|c|}
\hline Pattern & Description \\
\hline (pat1, pat2, ...) & \begin{tabular}{l}
A tuple of patterns matches a tuple value if the number of inner patterns pat 1 , pat 2 , etc. is the same as the number of values in the tuple and each inner pattern matches the corresponding value in the tuple. pat 1 is bound to the first element in the tuple, pat 2 is bound to the second, and so on. For example: \\
(part1,part2) = input_event "choiceAttribute" \\
binds part1 to the first value in the choice tuple, and part2 to the second value.
\end{tabular} \\
\hline [] & This matches an empty list. \\
\hline [pat1, pat2, ...] & \begin{tabular}{l}
This matches a list with the same number of elements as there are inner patterns pat 1, pat 2 etc. Each inner pattern is bound to the corresponding list element. For example:
\[
[a, b, c]=[1,2,3]
\] \\
binds a to \(1, \mathrm{~b}\) to 2 , and c to 3 .
\end{tabular} \\
\hline pat1::pat2 & \begin{tabular}{l}
This matches any list that is not empty. The pattern pat 1 is bound to the first element in the list and the second pattern pat 2 is bound to the rest of the list, from the second element on. For example: \\
val first::rest \(=[1,2,3,4]\) \\
binds first to the value 1 and rest to the value \([2,3,4]\).
\end{tabular} \\
\hline pat1 as pat2 & This allows two patterns to bind to the same value. The first pattern must be an identifier, optionally with a type constraint. At the same time, the value is matched with pat 2 . The result is that you can use the name in pat 1 to refer to the whole value and the names in pat 2 to refer to its parts. This can be more efficient. For example:
val whole as (part1, part2) = input_event "choiceAttribute" \\
\hline
\end{tabular}

Binding patterns can be used in more detailed data type manipulations as we will see later in this chapter.

\section*{Writing ECDL Expressions \\ Binding and Comparisons}

\section*{"Let" Expressions in Node Parameters}

To take advantage of named values and binding patterns in the parameters of circuit nodes we need to introduce the let expression.

To test if an event's severity is either 5 or 7 , and pass it through a Filter node if it has, you could use an expression such as:
```

input_event "severity" = 5 or input_event "severity" = 7

```

This can be written using a let expression as:
```

let
val severity = input_event "severity"
val badProblem = 5
val reallyBadProblem = 7
in
severity = badProblem or severity = reallyBadProblem
end

```

The named values badProblem and reallyBadProblem are introduced only to show simple named values being defined, and to help express the meaning of the subsequent condition. The values 5 and 7 could have been used in the condition directly without affecting the result.

The let expression allows you to define named values that are visible only within the scope of the let expression. You define named values between the let and in keywords, and place any valid expression between the in and the end. The expression between the in and the end can refer to any of the named values defined earlier. The named values are only visible between the let and the end, and are invisible outside that scope. The value returned by the whole expression is the value of the expression between the in and end.

No identifier (the name of the named value) may be defined more than once within the same let expression, but let expressions may be nested and, if the same identifier is defined in a nested let expression, the inner identifier hides the one defined at the outer level.

A let expression can be used anywhere an expression can be used (such as in a node condition parameter).

\section*{Comments in ECDL}

ECDL uses the same language comment style as ASN. 1 and GDM O.
That is, comments begin with a double-hyphen "--" and terminate with either the end of the line, or another double-hyphen. For example:
```

-- This entire line is a comment
val myValue = 23 -- This is a comment after my value
-- definition
val myValue = --23-- 24 -- This is a comment after my value
-- definition, where the value 23 has
-- been "commented out", and the
-- value 24 is used instead.

```

\section*{Data Type Handling}

The basics of data types supported in ECDL are described in Chapter 8, "Data Types," on page 211. In summary, the ECDL types include:
- simple types like Integer, Real, Boolean, Oid, and String
- complex types such as Tuple, List, and Dictionary
- special types such as Function, and Void.
- type collections such as dynamic types and unions

This section describes how complex types, such as Tuples, Lists and Dictionaries, are handled.

\section*{Tuple Manipulation}
"Tuple Data Type" on page 222 introduced the tuple type. This section shows more advanced ways of manipulating tuples.

A tuple is a fixed structure containing two or more members. Each member of the tuple has a defined data type.

Tuples arise in ECDL expressions as node parameters, such as:
- the Time Alignment parameter to the Clock and Rate nodes
- event attributes in events that rely on ASN. 1 or MDL (ITO messages do not contain Tuples)

In ASN. 1 the SEQUENCE/SET and CHOICE types are represented using the Tuple type. In MDL subtypes are represented as Tuples.

Tuples appear in ECDL as a comma-separated member set enclosed in parentheses:
```

("number", 42)

```

Tuples may be nested to any level.

Tuple type specification

The type specification for a tuple consists of a list of member data type specifications, separated by asterisks "*", and enclosed in parentheses. The following example defines a tuple holding a string as the first member, and an integer as the second:
```

val myTuple: (String * Integer) = ("number", 42)

```

Tuple binding patterns

Binding patterns are frequently useful when accessing MDL and ASN. 1 event attributes. The form for the binding pattern is a comma separated member identifier set, enclosed in parentheses. For example:
```

val (firstMember, secondMember) = ("number", 42)

```

This results in firstMember \(=\) "number" and secondMember \(=42\).
To access the members of an ASN. 1 SEQUENCE type, you could write:
```

val (identifier, significance, _) =
input_event("eventInfo.additionalInformation[0]")

```

This might result in identifier \(=2.9 .3 .2 .7 .29\) and significance \(=\) true. Note also that the wildcard character "_" is used for the third member of the tuple to indicate that you are not interested in using that value. The wildcard character " " can be used more than once in a binding pattern. Identifiers must, however, be unique.

The length of a binding pattern is important: (id, sig, _ ) is not the same as (id, sig ).

\section*{List Manipulation}

A list is an ordered set of values and is of variable length. All the elements in a list must be of the same nominal data type. However, by making use of dynamic data types, lists can be defined to hold a variety of data. If a data type for the list is not is not defined then it is set to List Any Type.

Lists are represented in ECDL as a comma separated list of elements enclosed in square brackets. Some simple examples are:
\begin{tabular}{ll}
{\([5,4,8,10]\)} & -- List of Integers \\
{\([" a b c ", ~ " z y z ", ~ " d e f "]\)} & -- List of Strings \\
{\([1, ~ " g h i ", ~ 3.4]\)} & \(--~ L i s t ~ o f ~ m i x e d ~ t y p e s ~(L i s t ~ A n y ~ T y p e) ~\)
\end{tabular}

A more complex example is a list of tuples:
```

[ ("number", 1), ("number", 5), ("number", 7) ]

```

The empty list has the form:
[]

\section*{Writing ECDL Expressions}

\section*{Data Type Handling}

\section*{List type specification}

The type specification for lists consists of the keyword list, followed by the type specification for the data type of all elements in the list.

For example to define a list of I nteger values:
```

val myIntegerList : List Integer = [5, 4, 8, 10]

```

Alternatively, to define a list that can contain elements of any type:
```

val myAnyList : List (Any Type) = ["abc", 5, (2, "def")]

```

The parentheses around (Any Type) are simply for clarity and are not required.

List construction Lists may be constructed using the infix construction operator "::". For example:
```

5 :: 4 :: 8 :: 10 :: []

```
produces a list with the value [5, 4, 8, 10].
This operator takes its left operand as a data element and its right operand as a list, and is right associative. Which means that if you read the example starting with the empty list [] on the right side, then you prepend element value 10 to that list, which produces a list [10], which in turn has element value 8 prepended to it, producing [8,10], etc., until the list is constructed.

\section*{Searching Lists}

Some list operations are performed using functions documented in Chapter 9, "Operators and Built-in Functions," on page 233. The operations discussed here are not functions but are based on expressions whose syntax is part of the ECDL language.

Exists The exists operation is used to determine if a particular element is present in a list. The basic form of the exists expressions is:
```

exists element in someList where condition

```

This reads "does there exist an element in someList, such that a given condition is satisfied." The result type of this expression is boolean.

For example
exists element in \([2,4,1,7]\) where element \(>3\)
produces the bool ean result of true, since the list contains at least one element greater than 3 . The elements in the list are iterated over from
first to last (or until the condition evaluates to true), with element representing each element in the iteration for the evaluation of the condition.

An example is to check for any events in a Table node's contents attribute that are of a given event type:
```

exists table_event in table1.contents where
table_event "eventType" = 2.9.3.2.10.4

```

An older form of this expression also exists: (v1 form).

\section*{ForAll}

Find

The forall operation is similar to the exists operation except that where exists returns true if any element in a list satisfies a given condition, forall must satisfy the condition for every element in the list.
The basic form of the forall expression is:
forall element in someList where condition
This reads "is it true that, for all elements in someList, the condition is true." The result type of this expression is boolean.

For example
forall element in \([2,4,1,7]\) where element \(>3\)
produces the bool ean result of false, since at least one element in the list is not greater than 3 . The elements in the list are iterated over from first to last (or until the condition evaluates to false), with element representing each element in the iteration for the evaluation of the condition.

An older form of the forall expression also exists: (v1 form).
The find operation searches a list for the first element that matches a given condition.

The basic form of the find expression is:
```

find element in someList where condition

```

This reads "Find the first element in someList, such that a given condition is satisfied." The result type of this expression is a tuple containing the matched element as the first member, and the remaining list after the found element as the second member.

For example
```

find element in [2, 4, 1, 7] where element = 4

```
produces the result
(4, [1, 7])
The elements in the list are iterated over from first to last, with element representing each element in the iteration for the evaluation of condition.

If condition cannot be satisfied for any element in the list, the NotFound exception is raised.

An older form of the find expression also exists: (v1 form).

\section*{Deriving Lists from Lists}

Some complex decisions on lists of events are more simply expressed as a combination of simpler decisions.

Suppose we have a Table node hol ding al arms from a wide area and wish to determine if there have been more than three alarms from Sydney of severity greater than 5 . This can be broken down into two steps.
1. Select the events in the table that satisfy the alarm condition.
2. Count them.

The selection step can be done with a select expression, such as this:
```

select event
from event in alarm_table.contents
where event "severity" > 5 and event "source" = "Sydney"

```

The expression has three parts. The from part contains an iterator just like the search expressions. It passes over all events in the contents list and makes each one available under the name event. For each event in the list the condition in the where part is evaluated to decide if the event is to be selected. Then for each selected event the expression after the select keyword (called the map expression) is evaluated. The results of all of these evaluations are combined into a new list. Here the map expression is just event which means "put the selected event into the new list".

The result of this select expression is a list of just those events in the table that satisfy our condition. We can then count them using the length built-in function, which determines how many members are in a list.

Putting this all together we get a condition that looks like this:
```

let
val alarms =
select event
from event in alarm_table.contents
where event "severity" > 5 and event "source" =
"Sydney"
in
length alarms > 3
end

```

The expression after the select keyword can be used to put more than just the event into the new list. For example we could build a list of the devices that produced the alarms, from a "device" attribute, by writing
```

select (event "device")
from event in alarm_table.contents
where event "severity" > 5 and event "source" = "Sydney"

```

There will be one device value in the list for each selected alarm. This will probably mean that there are duplicates in the list. You can use the union built-in function to discard duplicates.
```

let
val devices =
select (event "device")
from event in alarm_table.contents
where event "severity" > 5 and event "source" =
"Sydney"
val unique_devices = devices union []
in
... -- use unique_devices
end

```

Where part The where part is optional. If omitted then the default is as if you wrote where true.

Iterator patterns Just as with the search expressions, the iterator can include any binding pattern. Only those list members that match the pattern are considered. In the variable bindings of an SNMP event, we could find all string values and their names in a variable binding list by writing:
```

select (name, value)
from (name, ("simple", ("string", value))) in var_bindings

```

Here we use the default where part, which has a value of true. The only members that will be selected are those that match the pattern. For each
selected member we construct a Tuple of the name and value.

\author{
List builder
}

There is an older more compact form of syntax for select expressions, called the list builder syntax. The list builder equivalent of the first select expression example would be:
```

[event | event in alarm_table.contents :-
event "severity" > 5 and event "source" = "Sydney"]

```

The select syntax is the recommended form for readability.

\section*{Dictionary and Event Manipulation}

A dictionary is a structure that holds key-value pairs. The event data type is a specialization of the dictionary type, and is used to access the attributes of events in ECDL. We have seen many ECDL examples that access members of an event using the form:
```

input_event "eventType"

```

In this case, the string "eventType", which is the name of the event attribute, forms the key for a given event. The value returned by this expression is the value of the event attribute.

General dictionaries are used as the right-hand parameter to the alter function, and are required when using certain string pattern matching functions.

A simple example of a dictionary value is:
```

val myDict =
("currentTemp" => 32.1, "minTemp" => 27.0, "maxTemp" => 35.5)

```

Dictionary values are accessed in the same way as event attributes. For example:
```

myDict "minTemp"

```

This results in the real value of 27.0.
The previous examples all used a string key to access values in a dictionary. However, the key can be any simple data type. For example, attributes of temporary events in an ECS circuit are accessed using an integer key.

There is no restriction on the number of key-value pair members in a dictionary.

Dictionary type specification

The type specification for dictionary consists of the keyword dict, followed by the specification for the data types of the key and value, comma separated and enclosed in parentheses. The data type of the key can be any simple type.
The following is an example dictionary with a string key, and a real value:
```

val myDict : dict (String, Real) =
("currentTemp" => 32.1, "minTemp" => 27.0, "maxTemp" => 35.5)

```

\section*{Flow-control Expressions}

This section discusses the ECDL flow-control expressions.
Like many languages, ECDL provides if and choose for alternative expression evaluation. It also supports switching on dynamic types with the typecase expression.

ECDL provides no explicit looping expressions but there are constructs that allow equivalent processing to be done. Iteration is typically handled by built in functions or by writing recursive functions (a function that either directly or indirectly calls itself).

The flow-control expressions in ECDL are:
- Choose
- Case
- Typecase
- If
- Sequence

\section*{Making Decisions: If and Choose Expressions}

An if expression can be used to select between two values, depending on a Bool ean value. In the following example the severity threshold depends on the source of the event.
```

let
val source = input_event "source"
val severity = input_event "severity"
in
if source = "Sydney"
then
severity > 5
else
severity > 3
end
end

```

An if expression must have both a then and else part containing an expression each (and an end afterwards). The value of the if expression is either the value of the then expression or the value of the else
expression depending on whether the value after the if is true or false.
When we examine the if expression we see that only the threshold varies with the source. We could reduce the if to the smallest size by rearranging the expression:
```

let

```
    val source = input_event "source"
    val severity = input_event "severity"
    val threshold \(=\) if source \(=\) "Sydney" then 5 else 3 end
in
    severity > threshold
end

This example illustrates an important facet of ECDL. An expression evaluates to a value so any expression can be used where any value is expected, provided that the expression evaluates to the expected type. A literal value is the simplest form of an expression. The right hand side of a val declaration must be an expression so we can put an if expression there. It selects between the two integer literals which are the then and else expressions.

The above example also shows a name being declared in one val declaration and used in another.

If we wanted to choose the threshold depending on several sources we could use nested if expressions. For example:
```

if input_event "Source" = "Sydney"
then
5
else
if input_event "Source" = "Canberra"
then
3
else
4
end
end

```

Here the else expression is itself another if expression. This can be written more simply using a choose expression.
```

choose (input_event "Source") of
"Sydney" => 5
"Canberra" => 3
| _ => 4
end

```

\section*{Flow-control Expressions}

The expression after the choose keyword is the test expression. The body of the choose expression is a sequence of choice parts separated by a vertical bar character (|). E ach part consists of a choice expression and a result expression. The value of the test expression is compared with each of the choice expressions in turn. When one is found that is equal to the test value then the corresponding result expression is evaluated. The result value becomes the value of the choose expression.

The last choice part in the above example is a catch-all that uses the wildcard symbol (an underscore _ ). The wildcard symbol matches with any test value so the last choice part is always chosen if none of the previous parts was chosen.
A wildcard part is not necessary, but if you omit it and none of the choice parts is sel ected then a NotF ound exception is raised. The wildcard must always be the last part, if present.
The next section describes a similar expression, the case expression, which makes choices depending on value structures.

\section*{Case}

A case expression is similar to a choose expression but it makes its decision based on the structure of the test value, by matching binding patterns rather than by testing for equality.

A (simplistic) demonstration is to find the first member of a tuple no matter whether it has two or three members.
```

case myTuple of
(f, _) => f
| (f, _, _) => f
end

```

Here we use the name f to match the first member of the tuple in each case and wildcards to ignore the other members.

The case parts are examined from top to bottom in order. The first one that matches is selected and the expression after the arrow is evaluated. If no case part matches then the BindingM ismatch exception is raised. You can add a catch-all using a single wildcard. For example:
```

case myTuple of
(f, _) => f
(f, _, _) => f
_ => "rubbish"
end

```

This works because a wildcard matches any value, no matter what its structure.

Here is an example of a case expression to extract the "TimeTicks" value from an ObjectSyntax value in an SNMP variable binding, if it is a "TimeTicks" value, otherwise return 0.
```

case object_value of
("simple", ("ticks", n)) => n
|_ => 0
end

```

Literal strings are used to match the SNMP choice tags for the choices that we want. (The nested pattern structure follows the structure of the ObjectSyntax ASN. 1 declaration.) The name \(n\) is bound to the "TimeTicks" value if the rest of the pattern matches.
A very common use of a case expression arises when writing your own functions to iterate over lists. At each step of the iteration, examine the first member of the remainder of the list. The code for a complex list search might include something like this:
```

case rest_of_list of
[] => raise NotFound
first :: rest =>
if first = ... -- test first, use rest
end

```

You must consider both the empty and non-empty cases otherwise a BindingMismatch will occur.

\section*{Typecase}

Typecase expressions provide a mechanism to do alternative processing, depending upon the data type of something (such as an event attribute). Typecase is useful for dealing with dynamic or union types in ECDL.

The basic form of the typecase expression is:
```

typecase match oftype_expression_1 => result_1
|type_expression_2 => result_2
| ...
|type_expression_n => result_n
end

```

\section*{Flow-control Expressions}

The match expression is evaluated first. Then, each type_expression_ is evaluated in order from first to last. The evaluation stops when a type_expression_ matches match. The corresponding result_is then evaluated. The return value of the entire typecase expression is the return value of result_ for the matched type_expression_.

The possible type values for type_expression_ are listed in Table 10-2.
Table 10-2 Valid Typecase Types
\begin{tabular}{|l|l|l|}
\hline Integer & Real & Boolean \\
\hline String & Oid & Duration \\
\hline Time & Void & Opaque \\
\hline Any list & Any Dict & Any Event \\
\hline Any Data & Any Type & \\
\hline
\end{tabular}

You can also use any Union of these types (Integer | Real) for example, or user defined names for these types (including names of token types).

The Tuple data type is not supported for typecase expressions.

The following example is a typecase expression that results in the boolean value true if the data type of the value being tested is integer or real, or false otherwise.
```

typecase myValue of
Integer=> true
Real=> true
Any Type=> false
end

```

There is no specific construct for a default typecase part. However, the type Any Type matches on any data type, and acts as a catch-all. If no match is made for the typecase, a TypeMismatch exception is raised.

\section*{Sequence}

The sequence expression is a mechanism for separating multiple ECDL expressions, sequentially. This construct is sometimes useful to configure M odify or Create nodes (though its use is not necessary). It is rarely used otherwise.

The basic form of the Sequence expression is:
```

expression_1 ; expression_2 ; ... expression_n
expression_1 is evaluated first, followed by expression_2, and so on.
E ach expression is separated from the next by a semicol on ";". For
example, in a Modify node:
(
input_event alter ("generic-trap" => 1);
input_event alter ("specific-trap" => 3)
)

```

In this example, the first alter is performed, then the second. The return value of a sequence expression is the return value of the last expression in the sequence.

The parentheses surrounding the sequence expression are optional in most places. The ECS Designer always puts parentheses around the text of any node parameter so you can write the parameter as a sequence expression without the parentheses.
The return value of sequence expressions is ignored by Create and M odify nodes (the event is modified rather than the return value being used). Consequently, the return value from the sequence does not matter.
On the other hand, if a sequence expression is used where the return value is to be used, you must ensure that the last expression in the sequence returns the value you want. The return value(s) of previous expressions in the sequence are ignored. The fol lowing example illustrates the point for the condition parameter of a Filter node, where a boolean result is required:
```

(
system.trace "Filtering on severity = ";
system.trace (stringOf (input_event "severity"));
input_event "severity" = 3
)

```

\section*{Flow-control Expressions}

The overall result is true if the event's severity is 3, or false otherwise. The system.trace expressions are evaluated for their side-effects (printing a value to the trace log), and return values (Void in this case) are ignored and lost.

\section*{Exceptions}

Exceptions are a convenient way of dealing with errors without having to write code to deal with every possible error condition. Exceptions let you alter the flow of control in a controlled and predictable way.

When an ECDL operation cannot be completed, an appropriate exception is raised. An exception disrupts the normal flow of control and prevents the expression from returning a value. When an exception is raised you can explicitly handle the exception by taking some action within ECDL or you can allow the exception to be passed back to the ECS circuit node in which the expression is being evaluated. Here the exceptioth will be logged, unless the error output port of that node has been connected.

\section*{Handling Exceptions}

Typically, an exception is raised because of some error in the ECDL expression being evaluated. Allowing the exception to be passed back to the ECS node makes sense most of the time. However, there are some instances where you may decide to explicitly handle the exception and return an appropriate value. For instance, a NotFound exception may be raised by a find operation on a list, where no element in the list matched a given condition. Under some circumstances this would not be considered an error, in which case you would like to handle the NotFound exception rather than allowing the exception to be passed back to the ECS node.

The following example shows how to return the number of elements in a list, after a certain element (that satisfies a given condition), or -1 if that element could not be found in the list:
```

let val (_, theRemainingList) = find table_event in
tablel.contents where table_event "eventType" =
2.9.3.2.10.4
in
length theRemainingList
end
handle NotFound => -1 end

```

The handle is attached to the smallest previous expression (if necessary, use parentheses to define the expression to be handled). In the above example, it is attached to the let expression. Any expression can have a

\section*{Exceptions}
handle expression defined for it. For instance, a handle could be placed in the declarations section, or expression section of a let expression, if needed to handle exceptions at that level.

In the previous example, the handle shown is a simple, one exception handler. However, the handle expression is very similar to a case expression, enabling different expressions to be evaluated depending on the exception raised.

The basic form of the handle expression is:
```

handlehandle_pattern_1 => expression_1
|handle_pattern_2 => expression_2
...
|handle_pattern_n => expression_n
end

```

Where handle_pattern_x is an exception name, or the wildcard handle pattern "_" may be used as a default handle pattern to match any exception.

\section*{Standard Exceptions}

The standard predeclared exceptions are listed in Table 10-3 (names are case-insensitive). The Code represents the Integer value code that may appear in engine logs.
Table 10-3 Standard Exceptions
\begin{tabular}{|l|l|l|}
\hline Code & Name & Description \\
\hline 1 & BindingMismatch & \begin{tabular}{l} 
Raised when an attempt is made to bind a value to a \\
binding pattern and the binding cannot be completed.
\end{tabular} \\
\hline 2 & TypeMismatch & \begin{tabular}{l} 
Raised when the type of a data value does not \\
conform to the data type expected for the operation \\
being attempted.
\end{tabular} \\
\hline 3 & DivideByZero & Raised when an attempt is made to divide by zero. \\
\hline 4 & RloatingOverflow & \begin{tabular}{l} 
Raised by an operation performed with real \\
arithmetic that overflows or underflows the floating \\
point representation.
\end{tabular} \\
\hline 5 & RangeError & \begin{tabular}{l} 
Raised in cases where a value is of the right type but \\
its value exceeds some allowed range.
\end{tabular} \\
\hline
\end{tabular}

\section*{Table 10-3 Standard Exceptions}
\begin{tabular}{|l|l|l|}
\hline Code & Name & Description \\
\hline 6 & NotFound & \begin{tabular}{l} 
A general-purpose exception that is raised by some of \\
the built-in search and dictionary operations.
\end{tabular} \\
\hline 7 & InvalidArgs & \begin{tabular}{l} 
Raised when a built-in function is applied with \\
invalid arguments. Declared functions could use this \\
as well. For example: applying ord to a string with \\
length \(>1\).
\end{tabular} \\
\hline 8 & InternalError & \begin{tabular}{l} 
Raised when the correlation engine detects a fault \\
while evaluating the predicate. The evaluation may \\
be unable to continue.
\end{tabular} \\
\hline
\end{tabular}

\section*{Raising Exceptions}

Both user defined and standard exceptions may be raised from ECDL expressions. Generally, you define a new exception as a Global Definition using the basic form:
exception exception_identifier
For example, in
exception myNewException
myNewException is now an exception name that can be used in handle expressions.

To raise an exception, the basic form is:
raise exception_identifier
For example
```

raise NotFound
raises the standard NotFound exception. Likewise
raise myNewException

```
raises the user defined exception myNewException.

\section*{Functions and Language Layout}

The function mechanism in ECDL includes a large part of the mechanisms of functional languages. The power of these advanced features can be used to simplify the writing of sophisticated conditions.

\section*{Calling Function Syntax}

The standard layout for calling ECDL functions is:
```

function_name parameter_1 parameter_2 ... parameter_n

```

A simple example is:
```

split "abc:def:ghi" ":"

```

Here, the function_name is split, and it has two parameters, the string "abc: def:ghi" and the string ": ". Parameters in ECDL are whitespace-separated, not comma-separated as in some other languages. Parentheses may sometimes be necessary to delimit the parameters. For example
```

split ("abc:def" + ":ghi") ":"

```

In this case, the string "abc: def" is concatenated with the string " : ghi", and that resulting string is the first parameter to the split function and the string ": " is the second parameter.

A subtle problem in ECDL is the action of the - (unary minus) operator. This is simply a function that takes one parameter, a number, as in -1.23 . However, where a literal value is being passed to a function, as in the following expression:
```

round -1.23 -- wrong!

```
a type error is raised unless you use parentheses:
```

round (-1.23) -- correct

```

This forces the unary minus operation to be evaluated before the round. Without the parentheses the default order of evaluation is (round -) 1.23, thus leading to a type error when an attempt is made to pass the unary minus operator to the function round.

\title{
Writing ECDL Expressions \\ Functions and Language Layout
}

Extra parentheses can be freely used to ensure the appropriate parameters are passed. For example:
```

split ("abc:def" + ":ghi") (":")

```

In this case, the parentheses around the second parameter, although completely redundant, do no harm.

The following examples show common errors with parentheses:
```

split ("abc:def:ghi" ":") -- Error, trying to pass
-- a single parameter, which
-- is itself incorrect
split ("abc:def:ghi", ":") -- Error, trying to pass a
-- a single parameter, which in
-- this case is a tuple
-- containing two strings.

```

\section*{Prefix and Infix Functions}

There are two types of functions that can be called in ECDL: prefix and infix. Prefix functions arelike the split function in the previous example. These functions have the function name followed by its parameters:
```

function_name parameter_1 parameter_2 ... parameter_n

```

Infix functions are slightly different, in that the first parameter appears before the function name and only two parameters are allowed:
```

parameter_1 function_name parameter_2

```

For example, in a Modify node the Modify Spec parameter:
input_event alter ("severity" => 3)
the function alter takes two dictionary parameters: input_event and ("severity" => 3). The first parameter is to the left of the function name and the second parameter is to the right. Infix functions improve the readability of ECDL for certain functions and simplify the construction of multiple expressions.

\section*{User Defined Functions}

By writing ECDL functions, similar expressions can be shared as reusable functions, and complex conditions can be made more readable by calling functions to solve the problem.

Typically, user defined functions are stored in the Global Definitions for your ECS circuit then called from within node parameters. However, functions can also be declared locally, within a node parameter expression, or nested within let expressions. Functions can even be declared without naming them (anonymous functions).

When you type an expression in the condition parameter of a node you are providing the body of a function. The Filter node condition parameter is a good example. When an event enters a Filter node, the expression in the condition parameter is called as a function. This function must return a boolean value and takes one parameter: the input_event.

The basic form for writing functions is:
fun function_name param_1_spec param_2_spec .. . param_n_spec :return_type_expression \(=\) function_body_expression

Where function_name is the name of the function, param_x_spec is the specification for each of the parameters to the function,
: return_type_expression is the optional data type to be returned by the function and function_body_expression is the body of the function.

The format for the param_x_spec has the form:
param_binding_pattern : param_type_specification
Where the param_binding_pattern is the binding pattern which the calling parameters will be bound to, with the identifiers in the binding pattern available for use in the function body, and : param_type_specification is the optional data type specification for the parameter.

It is recommended that you include type definitions for all parameters. Likewise, a type definition for the function result is also recommended. For example:
func myfunc (param1 : Integer) (param2 String) : Boolean =...

Fol lowing is a simple a function that takes two integer parameters, adds them, doubles them, and returns the value:
```

fun addAndDouble (x:Integer) (y:Integer) :Integer = (x + y) * 2

```

NOTE
The parentheses around the parameter type constraints in the previous
example are required.

To call this function you write:
addAndDouble 25
Which returns the integer value 14.
The type specifications for the parameters and function result are optional, so the function could look like:
```

fun addAndDouble x y = (x + y) * 2

```

However, without the type constraints the compiler cannot check that valid types are passed to the function, so type errors may be raised at runtime rather than during compilation.

\section*{Advanced Function Writing Features}

Iteration and Recursive Functions

Using a select expression

ECDL provides no explicit looping expressions but there are constructs that allow equivalent processing to be done. Iteration is typically handled by built in functions or by writing recursive functions (a function that either directly or indirectly calls itself).

The following examples, show several different ways to count the number of negative integers in a list of integers. The first example uses a select expression, the second uses the alternative list builder expression, the third uses the foldl built-in ECDL function, and the fourth uses a recursive function. For each example, assume the following input list:
val myList \(=[1,-4,0,-1,5,9]\)
E ach example should produce the integer value 2.
length (select e from e in myList where \(e<0\) )
Here, the select expression has been used to first construct a new list that contains only those elements from the input list that are negative. The built-in length function is then used to return the number of elements in that new list.

\section*{Using a list builder expression \\ Using the built-in foldl function}
length [ e | e in myList where e < 0 ]
This is just an alternative form of the select expression and has exactly the same effect.

The built-in fold functions are often the most desirable way to iterate through a list. In the following example, increment I fNegat ive function returns an updated count (countSoFar) for each element in the list:
```

fun incrementIfNegative element countSoFar =
if element < 0
then
countSoFar + 1
else
countSoFar
end
foldl incrementIfNegative 0 myList

```

The foldl function takes a function as its first parameter, and calls the function for each element in the list. The parameters to the called function are the current element of the iteration followed by the return value from the last call to that function in the iteration (or the second parameter to the foldl function if it is the first step in the iteration).

Foldr could equally be used here, just the direction of iteration would be reversed.

The first step in the iteration calls incrementIfNegative with the following parameter values:
incrementIfNegative 10
to return the value 0 . The second step is
incrementIfNegative -4 0
returns the value 1. The third step is
incrementIfNegative 01
and so on.
The final value for the foldl function is that returned from the last call to increment IfNegative, which is 2 in this case.
Typically, the function definition is stored in the Global Definitions of the circuit, and the foldl call is done from a node parameter. The following example shows a Filter condition parameter, which passes an event only if there are less than 3 negative values in the given list:
```

(foldl incrementIfNegative 0 myList) < 3

```

Often, simple functions like the one passed to the foldl function here, can be done with an anonymous (un-named) function, written inline with the foldl call. The following is equivalent to the previous example:
```

foldl (fn element countSoFar =>
if element < 0
then
countSoFar + 1
else
countSoFar
end) 0 myList

```

N ote the use of the ECDL keyword fn for the anonymous function.
The definition for increment IfNegative did not use any type constraints. However, it is good practice to use type constraints, wherever it makes sense, to help detect type-mismatches at compile-time rather than run-time, and to provide self-documenting code. Thefunction definition would then have started as:
```

fun incrementIfNegative (element : Integer)
(countSoFar : Integer ) : Integer =

```

In this case we are restricting the function to only accept Integers, and only result in an Integer. Without the type constraints, the function would have been able to work directly (without change) on real numbers also. You can take advantage of this to write functions that support multiple types. The use of a Union type may be helpful for these cases also.

Writing a recursive The fourth and final way to solve our example problem is to design a function function to recursively call itself until it has counted all the negative numbers in the list.
```

fun countNegative (theList : List Integer) : Integer =
let
fun countRemaining (remainingList : List Integer)
(countSoFar : Integer) : Integer =
case remainingList of
[] => countSoFar-- At end of List
| element::restOfList =>

```
```

    if element < 0
    then
        countRemaining -- recursive
        restOfList
    (countSoFar + 1)
    else
-- Is not a negative number
countRemaining -- recursive
restOfList
countSoFar
end
end
in
countRemaining theList 0
end

```

To call the above function:
```

countNegative myList

```

Here, we call the countNegative function, with our input list as a parameter. This function acts as a wrapper function to the local recursive function (count Remaining), which does the actual work. The only reason for using the wrapper function, is that the actual recursive function requires an extra parameter, which is the count of negative numbers so far (0 initially). The following expression would produce the same result:
```

countRemaining myList 0

```

The count Remaining function takes two parameters, the remaining list that is yet to be checked for negative numbers, and the count of negative numbers so far. The first call to this function will receive our entire list, and 0 as the initial count.

This function first checks to see if it has reached the end of the list (using the case expression to check for empty list). If the empty list is found, the result is simply the countSoFar. If the list is not empty, the list is separated into the head el ement, and the remaining list (using the :: binding pattern in the case expression). If the element is negative, the countRemaining function is called recursively, with the remainingList and the countSoFar incremented by one. If the element is not negative, the same call is made, but the countSoFar is not incremented. The following shows the sequence of recursive calls:
```

countRemaining [1, -4, 0, -1, 5, 9] 0
countRemaining [-4, 0, -1, 5, 9] 0

```
```

countRemaining [0, -1, 5, 9] 1
countRemaining [-1, 5, 9] 1
countRemaining [5, 9] 2
countRemaining [9] 2
countRemaining [] 2

```

The final result is 2.
The piece of ECDL that checks whether the element is negative can be simplified, since the then and else parts are very similar:
```

if element < 0
then
-- Is a negative number
countRemaining -- recursive
restOfList
(countSoFar + 1)
else
-- Is not a negative number
countRemaining -- recursive
restOfList
countSoFar
end

```

This could be replaced with the more elegant:
```

countRemaining restOfList (if element < 0 then
(countSoFar + 1) else countSoFar end)

```

\section*{Fixity Declarations, Associativity and Precedence}

A fixity declaration changes the fixity, precedence and associativity properties of a name. These properties control how a function name is bound to its arguments and the order of evaluation. This section describes how you can set the fixity, associativity, and precedence of functions that you name yourself. You can also override an infix function at the point where it is called, thus temporarily converting it to prefix form. Also in this section are tables that describe the default associativity and precedence of the operators and built-in functions.

Fixity Declarations Fixity declarations must appear in the same declaration group as the name that it affects. The best place to put it is straight after the declaration of the name. The syntax is:
fixity name precedence associativity
where
- fixity is one of the reserved names infix or prefix
- name is the unqualified name that is being changed
- precedence is an integer precedence level
- associativity is one of the (non-reserved) names left, right or nonassoc.

Precedence controls the order of evaluation when two different operators conflict. For example in the expression \((1+3 * 4)\) either the + or the* could be done first. The * is done first since it has higher precedence.

Associativity controls the order of evaluation when two or more infix operations of the same precedence conflict. For example the expression (1 \(+2+3)\) could be evaluated with left-associativity:
\((1+2)+3\)
or with right-associativity:
\(1+(2+3)\)
If an operator is declared to be nonassoc then it is illegal to have this sort of conflict. F or example the =operator is not associative. You cannot write \((1=2=3)\). This is because neither \(((1=2)=3)\) nor \((1=(2=3))\) makes sense. In either case you would be comparing a Boolean with an Integer.
There is a restriction that all names in an expression at the same precedence level have the same associativity. So if you define your own infix operators you must take into account the precedence and associativity of the built-in operators.

Removing Infix It is sometimes useful to be able to call an infix operator or function as a prefix function. To convert an infix expression to prefix, place parentheses around the function name. For example, to sum the values in a list called myList you can write:
```

val total = foldl (+) 0 myList

```
\begin{tabular}{|c|c|c|}
\hline \multirow[t]{7}{*}{Defaults} & \multicolumn{2}{|l|}{The default fixity information for user-defined functions is prefix, a precedence level of 150 and non-associative.} \\
\hline & \multicolumn{2}{|l|}{The standard operators and symbols have the associativity shown in Table 10-4.} \\
\hline & Table 10-4 & Associativity of Operators and Symbols \\
\hline & Associativity & Operator/Symbol \\
\hline & left & or xor and (infix + ) (infix -) * / mod rem \\
\hline & right & \(\Rightarrow \wedge(\) prefix + ) (prefix -) implies \\
\hline & none & \(=!=\langle<=\gg=\) \\
\hline
\end{tabular}

By appropriate use of parentheses, you can always ensure the operation you want to perform is performed the way you require it to.

Table 10-5 shows the precedence of the built-in operators and relevant syntax elements. The higher the precedence number then the tighter the operator binds.

\section*{Table 10-5 Precedence and Fixity of Operators and Built-in Functions}
\begin{tabular}{|l|l|l|}
\hline Precedence & Infix / prefix & Operator/Symbol \\
\hline 130 & prefix & +- \\
\hline 120 & prefix & built in functions \\
\hline 100 & infix & \(::\), available for infix functions \\
\hline 90 & infix & \(\wedge\) \\
\hline 80 & infix & \(* / \bmod\) \\
\hline 70 & infix & +- \\
\hline 60 & infix & \(=!=\langle\langle=\gg=\) \\
\hline 50 & prefix & not \\
\hline 40 & infix & and \\
\hline 30 & infix & or \\
\hline
\end{tabular}

\section*{Table 10-5 Precedence and Fixity of Operators and Built-in Functions}
\begin{tabular}{|l|l|l|}
\hline Precedence & Infix / prefix & Operator/Symbol \\
\hline 20 & infix & implies \\
\hline 10 & infix & => \\
\hline
\end{tabular}

\section*{Modules}

A module is a group of declarations collected together under a module name. M odules may be nested directly within other modules and only other modules. All compound nodes, primitive nodes and other material that you include in your circuit appear within a module. Normally the ECS Designer takes care of the module structure automatically but it is useful to know about modules for the following reasons:
- Diagnostic messages such as trace and statistics information use fully qualified node names that show the module structure. Familiarity with the module structure makes it easier to read this information.
- Some built-in functions reside in their own module (for example, Time, Match and System). You must use the module name when accessing these functions.
- Many facilities in ECDL are packaged into modules. This includes declarations to do with particular endecoders. You can access the declarations in a module using qualified names.
- Everything is enclosed in a module called Std. This includes all of the built-in functions. It may sometimes be useful to qualify a built-in name with std to ensure that you are getting the right one.

In addition, you can add new modules:
- The global definitions section is inside a module (created by the Designer) that encloses the entire circuit.
- You can define your own modules as global definitions.

\section*{Modules}

\section*{Module Files}

Module files are special forms of correlation circuit source files. These files can be modified or created using the ECS Designer.

The purpose of module files is to define ECDL definitions or functions for inclusion into all circuits (like an ECDL equivalent of C "include" files). Some standard module files are included with the ECS Designer:
ecs_defaults.ecs
cmip_defaults.ecs
nnm_defaults.ecs

Standard definitions and default settings. Required.

Useful in relation to CMIP event access only

Required when creating circuits for the NNM correlation engine.

To include a "site-specific" module file, create a correlation circuit source file containing no nodes (this should compile with an error, no nodes type message). Place the source file (.ecs) into the modules directory. The default location of the modules directory is \$OV_CONF/ecs/modules/, though this can be overridden.

Inclusion of non-essential modules files in the modules directory will not normally affect circuit development. M odule scoping prevents name clashes.

\section*{Modules and Name Search}

The fully qualified name of a declaration in a module m is a sequence of module names from std to the modulem, then the declared name, all separated by periods, with no white space. Some examples of fully qualified names are:
```

Std.foldl

```

Std.Match.test
The first is the built-in foldl function and the second is the test function in the built-in string matching module Match. Module names on the left may be omitted if there is no ambiguity. For example Match. test is sufficient to locate the test function unless you have defi ned another visible module called Match.
\begin{tabular}{|c|c|}
\hline Terminology & \begin{tabular}{l}
In the following explanation we use this terminol ogy. \\
- The phrase "the local module" means the one that the declaration in question is immediately declared in. The declaration is said to be "local to the module". \\
- The phrase "an enclosing module" means any module that contains the declaration in question, directly or indirectly, including the local module. \\
- A "qualified name" has zero or more module names. A "fully qualified name" always starts with the module Std.
\end{tabular} \\
\hline Module syntax & A module is declared with the syntax
```

module identifier is
declaration
end

```
Declarations within the module are separated by white space. \\
\hline Open declaration & \begin{tabular}{l}
The open declaration has the syntax \\
open qualified-name \\
The qualified name must be the name of a module. This declaration causes the declarations in the module to be directly visible within the local module without the need to qualify them with module names.
\end{tabular} \\
\hline Name search & \begin{tabular}{l}
When you use a name in ECDL, as opposed to declaring one, the compiler must search for a declaration with that name. Any use of a name may be a qualified name. \\
The name search is conducted first in the local module and then in enclosing modules until std is reached. If this first search does not find the name then the declarations in opened modules are consulted. The open ded arations that are consulted are those in the local module and then in enclosing modules. \\
The search only looks at names of the appropriate kind. For example if the context shows that the name must be the name of a value then only val and fun declarations will be examined. The kinds of names are: value, type, exception and scope.The scope kind includes all modules, compound node names and instances of nodes, compound or primitive. \\
If a name is qualified then the first module name is searched for first.
\end{tabular} \\
\hline
\end{tabular}

\section*{Modules}

\section*{Scope of declarations}

Then if found, the rest of the qualified name must lead to a declaration in the found module and inner modules.

If there are multiple open declarations in a module then they don't interact. The qualified name in one declaration is not looked up in another opened module in the local module. It is, however, possible for the name in an open declaration to be found in a module opened by an outer open declaration.

The scope of a declared name is the region of ECDL text where the name may be used. The scope of a name declared in a module is all of the module. This means that a name may be used anywhere in the module, either before or after the place that it is declared. (This is in contrast to many other programming languages, which only allow a name to be referred to in places after it has been declared.)

\section*{Invoking Perl Functions from ECDL}

ECS provides a means to execute Perl functions from ECDL code. A built-in function, gen_perlcall, is used to invoke the perl script.

\section*{Pre-requisites}

This document assumes the following:
- Perl version 5.6 is installed on the machine
- The user has a working knowledge of Perl and is aware of the different data types
- The user has a working knowledge of ECDL and ECDL data types

The interprocess communication between the engine and the Perl script is transparent to the user. The perl function receives the paramters passed to it as an array/list

The gen_perlcall has two sets of parameters:
- name of the user supplied Perl function
- list of arguments to be passed as paramerters to the Perl function

The argument list can contain nested Lists, Tuples and other ECDL data types.

The gen_perlcall function calls the user supplied function, passes the paramters specified to that function, process it and returns a list of ordered set of values. The List can consist of any number of values in any combination of the supported data types.

\section*{Figure 10-1 Perl to ECDL}


\section*{Perl to ECDL Mapping}

The gen_perlcall uses ECDL data types to be passed into the called Perl function. The return type passed in the form of EV_AttrValues is also an ECDL data type. This is a recursive structure capable of representing a (nested) list of values of various data types. It is a direct representation of the ECDL List and supports most ECDL data types. Refer to Table 10-6 for a mapping of Perl data types to ECDL.

Perl functions are provided to enable exchange of data types from ECDL to Perl
- ecdlEncap () function is included in the customer supplied Perl file. This function can also be called from another file provided the file containing the ecdlEncap () function is sourced appropriately. For a sample of the ecdlEncap () function
```

\#begin
sub ecdlEncap
{
my $x=$_[0];
my $z=$_[1];
if( "\$z" eq "ECS_TUPLE" )

```
```

    { push @$x,"ECS_TUPLE"; }
    if( "$z" eq "ECS_OID" )
    { push @$x,"ECS_OID"; }
    if( "$z" eq "ECS_TIME" )
    { push @$x,"ECS_TIME"; }
    if( "$z" eq "ECS_DURATION" )
    { push @$x,"ECS_DURATION"; }
if( "$z" eq "ECS_BOOLEAN" )
    { push @$x,"ECS_BOOLEAN"; }
if( "$z" eq "ECS_BOOLEAN" )
    { push @$x,"ECS_BOOLEAN"; }
}
\#end

```

Ensure that the file containing the ecdlencap ( ) function defintion and the user defined Perl file are placed under the same directory.

The user defined Perl functions should be placed at the locations given below.

\section*{UNIX}
```

\$OV_CONTRIB/ecs/external/perl

```

\section*{Windows NT}
```

%OV_CONTRIB%ecs\external\perl

```

Sample Perl functions are placed at

\section*{UNIX}
```

\$OV_DB/ecs/examples/circuits/mdl/scen3

```

\section*{Windows NT}
\%OV_DB\%ecs \examples \circuits \(\backslash m d l \backslash s c e n 3\)

\section*{Example 10-1 ecdlEncap usage}
```

\#BEGIN perl
...
@ret1 = ("1.2.3.4")

```

\title{
Writing ECDL Expressions Invoking Perl Functions from ECDL
}
```

ecdlEncap(\@ret1,"ECS_OID");
return \@ret1;

```
\#END

\section*{Perl to ECDL Data Type Mapping}

The table bel ow maps the Perl data types to their ECDL equivalents for the various Perl data types that could be used:

\section*{Table 10-6 Perl to ECDL Data Type Mapping}
\begin{tabular}{|c|c|c|}
\hline ECDL Data Type & Perl Type & Example ECDL Value \\
\hline Object Identifier & String & "1.43.67.52" \\
\hline Tuple & Referance to Array & \[
\begin{aligned}
& (1.3 .6 .1 .4 .11 .2 .17 .2 .1 .0,(" \\
& \text { simple", ("number",108))) }
\end{aligned}
\] \\
\hline List & Referance to Array & ```
[
(1.3.6.1.4.11.2.17.2.1.0,("
simple", ("number",108))),
(1.3.6.1.4.11.2.17.2.1.0,
("simple", ("string",
"SYD")))
]
``` \\
\hline Duration & Double & 13.5h \\
\hline Time & Double & 19970103135959.123456z \\
\hline Integer & Integer & 123 \\
\hline REAL & Double & 5.6 \\
\hline String & String & "My String" \\
\hline Boolean & Integer & true and false \\
\hline
\end{tabular}

Additionally a new data format called ECS_FORMAT has been introduced. The ECS_FORMAT maps a complex list of values to the corresponding values in ECDL data types.

\section*{Invoking 'C' Functions}

ECS provides a means to execute ' C ' functions from ECDL code. A built-in function, gen_ccall, is used to invoke the 'C' function.

The interprocess communication between the ECS engine and the ' C ' funcion is transparent to the user.

\section*{Writing functions in C}

This section describes how to write the \(C\) function such that it is accessible from within ECS. The procedure consists of two parts
1. Writing the \(C\) function using the guidelines as described in the next section.
2. Create a shared library of the C function(s) and store the shared library in the following location

For UNIX
\$OV_CONTRIB/ecs/external
For NT
\%OV_CONTRIB\%/ecs/external

NOTE
The shared library is loaded by the runtime the first time a function within the shared library is invoked(as part of a correlation rule). When a shared library is loaded, the function _Init will be invoked if it exists. The signature for _Init is exactly the same as that of other \(C\) functions.

\section*{Writing ECDL Expressions Invoking 'C' Functions}

\section*{Guidelines for writting a C function}

\section*{Given below is the skeleton code for a function in 'C'}
```

\#include <stdio.h>
\#include <ECS/GC_Values.h>
int testFunction(int argc,void ** argv,int reqId,int cmdId,
genc_callback * callback)
{
int i = 0;
char * str = NULL;
char * oid = NULL;
char myStr[] = "some string ";
char myOid[] = "1.2.3.4.5.6.7.8.9";
GC_Values ** retValue = NULL;
GC_Values * intVal = NULL;
GC_Values * strVal = NULL;
GC_Values * oidVal = NULL;
/* Do your own checking here - this example checks if \# of
arguments is 3 */
if( argc != 3 )
{
/* Improper No. for argumnets */
/* Allocate the space for returing the err string back
to ECS */
retValue = (GC_Values **)calloc(1,sizeof(GC_Values *));
GC_MAKEVALUE(GC_ERRSTR, "Improper Argumnets", strVal);
if(!strVal)
{ /* hosed !!! just return */
callback(reqId, cmdId, 0, NULL);
return ;
}
retValue[0] = strVal;
/* Do callback to notify the error */
callback(reqId, cmdId, 1, \&retValue);
return 0;
}
/* Get the arguments passed to the function. They type of the
arguments needs to be defined by the function writer and its
the resposibility of the Orchestartor user to pass in the
correct number and type.*/
/* Do not free these values, will be freed by caller when
callback is called */

```
```

    i = *(int *)argv[0];
    str = (char *)argv[1];
    oid = (char *)argv[2];
    /* OID will be passed as string to the function */
/* Do your processing here */
/* processing is done-time to return back to ECS*/
/* THIS is the second half */
/* Allocate space for 3 return values - one can return any
number of retrun values - the example return 3 */
retValue = (GC_Values **) calloc(3,sizeof(GC_Values *));
/* Now create the wrapper to pass back the values to ECS*/
GC_MAKEVALUE (GC_INTEGER, \&i, intVal); /* Integer*/
if(!intVal)
{
/* Do Error handling */
}
GC_MAKEVALUE (GC_STRING, myStr, strVal); /* String */
if(!strVal)
{
/* Do Error handling */
GC_FREEVALUE (intVal);
}
GC_MAKEVALUE (GC_OID, myOid, oidVal); /* OID */
if(!oidVal)
{
/* Do Error handling */
GC_FREEVALUE (intVal);
GC_FREEVALUE(strVal);
}
/* Set the 3 return values in the wrapper */
retValue[0] = intVal;
retValue[1] = strVal;
retValue[2] = oidVal;
/* Call the callback to give the value back to ECS

1. ReqId
2. cmdId passed as the argument to this function
3. Number of returnvalues i.e. number of elements in the
GC_Values array
4. Address of the GC_Values array */
```

\title{
Writing ECDL Expressions Invoking 'C’ Functions
}
```

callback(reqId, cmdId, 3, \&retValue);
return 0;

```
\}

\section*{Basic Structure}

The signature of all functions that need to be invoked from ECS is as given below.
```

int func(int argc, void ** argv, int reqId, int cmdId,
genc_callback *callback)

```
where,
argc is the number of arguments passed in
argv is an array of pointers to the arguments
reqId and cmdId are opaque parameters and are used while calling the Callback function.
callback is a pointer to a function that needs to be invoked on completion and to pass back any return values to ECS.

\section*{Writing the function}

Writing a function consists of three distinct parts
1. the first part is getting the arguments passed from ECS
2. the second part is the processing
3. the third part is to return a value or a set of values back into ECS

The first part is getting the arguments passed from ECS. The function can take any number of arguments of any type. It is the responsibility of the user of the function to configure ECS correctly to ensure that the correct parameters are passed in. In the example above the function expects three parameters, an integer, a string and an OID. The pointers to these parameters are in argv and are accessed as argv [0], \(\operatorname{argv}[1], \operatorname{argv}[2]\). While accessing these pointers cast them to the right type.

Parameters passed in MUST NOT be freed by the function. The freeing of the space will be done when the Callback function is invoked.

The table below lists the data type as received by ECS.
\begin{tabular}{|l|l|}
\hline \begin{tabular}{l} 
Type passed in from \\
ECS
\end{tabular} & \begin{tabular}{l} 
Type as received by the C \\
function
\end{tabular} \\
\hline Integer & Integer \\
\hline Float & Float \\
\hline String & Char * \\
\hline OID & Char * \\
\hline Time & Integer \\
\hline
\end{tabular}

Once the parameters passed in have been accessed, the function needs to process it. The function indicates completion of processing by invoking the Callback function. This mechanism all ows the freedom to the function writer to process either synchronously or asynchronously. In other words the function returning does not indicate function completion. The user can choose to extract the arguments, queue them up for some other thread to process it and return immediately. Post processing the Callback function can be invoked to return the values to ECS and simultaneously indicates completion(obviously the call to the Callback will be made from a function other than the one that was originally called). This is useful when(for example) the function needs to go over the network to access data or when databases need to be accessed, both of which take time. If the function being invoked takes very little time, it is suggested that the Callback is called and then return.

If the function encounters an error at any point during it's processing, the error is indicated by calling the Callback function with the error code as shown in the example.

\section*{NOTE The Callback function must be called whether or not the function succeeds or fails.}

Returning values back - Returning values consists of four steps
1. allocating space for the return value
2. wrapping the return value

\section*{Writing ECDL Expressions Invoking 'C' Functions}
3. marshalling the return values
4. calling the Call back function
allocating space for the return value - Make the following calls to allocate space
```

retValue = (GC_Values **)calloc(X,sizeof(GC_Values *));

```
where x is the number of return values that needs to be returned.
retValue is defined like-GC_Values ** retValue = NULL;
wrapping the return value by calling a provided macro(present in the file Gc_Value.h) - as shown below

GC_MAKEVALUE (GC_STRING, valToReturn, strVal);
GC_Values * strVal = NULL
where,
the first parameter is one of
- GC_INTEGER to return an integer
- GC_STRING to return a string
- GC_OID to return an OID(to return an OID, the val ToReturn should be a string in the dot notation).

Example "1.2.3.4"
- GC_FLOAT to return a float
the second parameter valToReturn holds the value to be returned
the third parameter strval is a pointer to the macro GC_Values (Example, Gc_Values * strVal;)
marshalling the return values - in Step 1 we allocated space. In Step 2 we wrapped the return values. In this step we will tie the two.
Assuming there are two values to be returned, do the following
```

retValue[0] = valToReturn1;
retValue[1] = valToReturn2;

```
where retValue is gotten in step 1 and valToReturn1 and valToReturn2 are gotten is step2.
calling the Callback function- the last step is to indicate completion of processing and returning the values to be returned. This is done as

\section*{follows}
```

callback(reqId, cmdId, 3, \&retValue);

```
where
callback is the pointer to a function which was passed in as the fifth parameter and reqId and cmdId were the third and fourth parameters passed in.

\section*{An Introduction to ECDL for C Programmers}

Internally, ECS circuits are completely described as ECDL programs. The compiled ECDL program runs on the ECS Engine to correlate a stream of events. ECDL has two sides: a structural side and a functional side. The structural side describes how a circuit is assembled from nodes and other components. It is read and written by the ECS Designer and the only way to interact with it is graphically, through the circuit displayed on the ECS Designer canvas. The functional side of ECDL is used to implement circuit node parameters, such as the condition parameter of a Filter node.

Concepts Languages like C and Pascal fall into the category of imperative languages. In the imperative style there is some mutable global state (the computer's memory) and all computation is done by making changes to the state. A variable in an imperative language names a piece of the machine state. An assignment statement changes the value associated with a variable name and data structures are updated in place. The imperative programmer must be careful to keep all state changes in the right order or the result will be incorrect.
ECDL falls into the category of functional languages. In a (pure) functional language computation is done by applying pure functions to immutable values to compute new values. There is no state anywhere to be changed. A simple example of a pure function expression is
```

1 + 2

```

This demonstrates the three basic concepts:
- The values 1 and 2 are immutable. The value 2 is always two.
- The +operator is a pure function which means that its result depends only on its arguments, here 1 and 2 . Its behavior cannot be affected by anything in its environment.
- The effect of evaluating the expression is to produce a new value 3. Neither of the arguments 1 or 2 is affected.

This simple expression would be quite a bit harder to understand if you could not be sure what the value of its arguments really were or what the +operator was going to do with them. You would certainly be surprised if you got different results from the addition depending on some global state somewhere.

The goal of the functional style is to preserve these concepts throughout the language so that all code in the language becomes a lot easier to understand.

The problems that an imperative programmer has to deal with include:
- Making sure that variable and data structure updates occur in the right order especially over long sequences of function calls with side-effects.
- Copy-by-reference versus copy-by-value.
- "Sneak" paths due to side-effects.

The copy-by-reference problem arises when you want to make a copy of a data structure. (J ust passing a data structure as an argument to a function is an act of copying.) You could do this by reference, meaning that one instance of the data structure is shared by each copy. Alternatively you could duplicate the data, (copy-by-value) ensuring that there is no sharing. If you do the former then when the first copy is updated the second copy appears to change as well. If you do the latter then updating the first will not result in an update of the second. If you are not very careful about which is the correct way to copy in each circumstance then bugs will arise that can be quite difficult to track down.

Sneak paths are the general problem of side-effects, where one part of the program can have obscure effects on other parts through global variables, shared data structures, and lots more. Sneak paths make functions impure.

By starting with the premise that there will be no mutable state anywhere, functional languages avoid these problems.
- Since there are no side effects or mutations, the programmer no longer has to worry about the order of operations. The compiler is free to choose an execution order that is convenient and/or efficient.
- Copy-by-reference becomes indistinguishable from copy-by-value as far as the programmer is concerned. The compiler is free to choose whichever way is appropriate for each copy.
- The absence of sneak paths means that all data flows are explicit in the code. This makes the code easier to understand. It also makes it more modular since each piece can be guaranteed to function without interference from other pieces.

\section*{An Introduction to ECDL for C Programmers}

Notable Differences

A functional language transfers much more of the burden of programming to the compiler compared with an imperative language. This is an important advantage of ECDL since the goal is to minimize the need for programming when designing a circuit.

ECDL also contains only a rather small subset of the sophisticated features found in general purpose functional languages.

Here is a summary of notable differences between ECDL and C:
- There are no variables in the C sense of the word in ECDL. You can create a value, assign it a name, and manipulate values in sophisticated ways-but once created, a value is immutable. There is no place, for example, for the concept of incrementing a counter (as in the \(C\) expression \(n++\) ).
- Complex data types such as the List, Tuple and Dictionary are built into the language. They are supported by a range of operators that understand the semantics of these data types and make it easy to perform tasks that are complex and error-prone in languages such as C.
- Binding patterns provide a declarative way to build and extract component values from complex types. Binding patterns are a powerful and intuitive mechanism, fundamental to ECDL dedarations.
- Functions are first class values meaning that they can be passed around, stored in data structures and be the result of computations. This makes it easy to build complex functions by composing simpler functions without having to write code. For example complex filtering conditions can be composed from a library of simple conditions.
- ECDL is dynamically typed. This means that the types of values are not comprehensively checked until run-time, although many obvious type errors can be reported at compile time. This is needed because the types of event attributes is often not knowable at compile-time.
- A function must return exactly one value. However, the return value can be a complex data type containing several component values, or the special Void value if no other value is sensible.
- Since there are no mutable variables there can be no iterative-style loops like the for, while, or do...until loops of C. M ost tasks that involve iteration can be achieved with the rich collection of operations on lists. Custom iterative tasks can be solved using recursive

-- alternate form to above

\section*{Writing ECDL Expressions}

\section*{An Introduction to ECDL for C Programmers}
foldl and foldr
Lists are central to ECDL in several ways:
- Lists occur as event attributes, particularly in CMIP. For example, the CMIP "eventInfo.specificP roblems" attribute is a List of Tuples.
- Lists are a natural construct to use for any ordered collection of data of variable size.
- In annotation, Lists are used to exchange request/response data with an outside process. In this case, lists provide a flexible way to exchange data of type Any Data. This means that each element in the List can be of any data type.

\section*{Binding patterns}

Much of the expressive power of ECDL is due to the concept of binding patterns. Binding patterns provide a conceptually simple mechanism that allows you to associate names with values. However, unlike a simple assignment mechanism, binding patterns can assign names to parts of complex data structures, such as Tuples and Lists. In most other languages extracting a component value from a complex data structure is like peeling an onion-you peel back each layer in the structure until you reach the value you want. Binding patterns replace this procedural approach with a simple dedarative style where you effectively say "match up these names to the parts of an object that looks like this." After that you simply access the parts using the names you supplied.

Binding patterns are an indispensable mechanism when dealing with CMIP and SNMP events because of the extensive use of Tuple data types and the ability of binding patterns to easily extract multiple component values from a Tuple. ECDL Tuples represent ASN. 1 SEQUENCE, SET and CHOICE constructs.

Summary
ECDL is a functional Ianguage, well-suited to the event correlation domain. Specific features of ECDL that make it such a good match include:
- Statelessness which helps to make reasoning about circuits easier. M ost circuit nodes are stateless or nearly so and ECDL preserves this property in the expressions attached to them.
- Easy access to components of complex data types built from Tuples, Lists and Dictionaries.
- It minimizes programming by providing powerful features as well as mechanisms for building up complex operations from a library of
simpler operations.

Writing ECDL Expressions
An Introduction to ECDL for C Programmers

\section*{11 Event Log File Format}

This chapter contains a detailed description of the structure of the ECS event log files generated by the ECS Engine and used by the ECS Designer in Simulation mode.

The event log file format is actually in two parts. The first part describes the overall format of log files and the format of meta-information carried in the file. This is the subject of the present chapter.
- "Log File Syntax" on page 379
- "Special Characters" on page 385

The second part is the format used to represent events themselves. Binary encoded events such as SNMP and CMIP are represented in a modified ASN. 1 format which is documented in the appropriate module guide:
- HP OpenView Event Correlation Services SNMP Module
- HP OpenView Event Correlation Services CMIP M odule
- HP OpenView Event Correlation Services ASCII Module
- Appendix A, "Event Correlation in OVO," on page 401

\section*{Log File Syntax}

ECS event log files consist of a series of events. Each event is defined by a sequence of line types that must occur in a defined order. The line types and the order in which they must occur are listed in Table 11-1. The default lead-in character(s) for the control lines must appear at the very start of the line.
Table 11-1 Event Log Line Types
\begin{tabular}{|c|c|c|}
\hline Lead-in & Type & Details \\
\hline \# & Comments (optional) & One or more contiguous comment lines starting with a \# character and ending with the first line that does not start with a \#. Thereafter, any line in the event starting with \# is not treated as a comment. This allows one block of comment lines per event. \\
\hline + & Time control (optional) & A time control line commences with a + character and is used to advance the engine time by the number of seconds specified. \\
\hline ! & Repeat (optional) & A repeat control line commences with a! character and is used by the ecsevgen utility to send the specified number of copies of the event to the engine. It is ignored by the ECS Designer. If a repeat control line is not present then a single copy of the event is generated. \\
\hline & Event & One or more lines comprising the event itself, terminated by and interpreted according to the following event syntax control line. \\
\hline \[
\begin{aligned}
& \%[\text { space }] \\
& \text { (can be } \\
& \text { changed) }
\end{aligned}
\] & Event Syntax & The event syntax line commences with the specified syntax start control sequence. By default this is a percent character followed by a single space character(\% ...). The event syntax control line specifies which endecode module to use and the syntax that the endecode module should use to decode and encode the event. See below for details \\
\hline
\end{tabular}

\section*{Log File Syntax}

The characters \# + ! and \% are only treated specially if they occur as the first character in a line.

For example, an event log with one event is illustrated in Figure 11-1.
Figure 11-1 A Simple Event Log Entry


In the example shown in Figure 11-1 the event occupies only oneline and is terminated by the syntax line starting with a \(\%\). This line specifies that the preceding event is to be decoded using the MDL endecoder and the syntax SimpleEvent, and that a network delay of 3 seconds should be simulated.

The order of events in the log determines the order in which events are input to the engine during simulation. See "Simulating Time in the ECS Designer" on page 173 for details.

\section*{The E vent Syntax Line}

The event syntax line serves several purposes:
- it terminates the event
- it specifies the endecoder module to be used to encode this event
- it specifies the syntax of the event
- and optionally, it specifies the number of seconds that this event is deemed to have been delayed on the network relative to its creation time.

The event syntax line consists of three fields separated by col ons:
\% endecoder:syntax:delay
where the fields have the following meanings:
- endecoder is a mandatory field that specifies the name of the endecoder module (mdl, ber, OpC_Msg, or anno).
- syntax is a mandatory field that specifies the syntax of the event (for example, Trap-PDU for SNMP traps, request for annotation requests, or response for annotation responses).
- delay is an optional integer value specifying the number of seconds that this event is deemed to have been delayed in the delivery network, relative to its creation time. See Chapter 5, "Timing Considerations," on page 165 for details on simulating network delays.

\section*{Logs Generated by the ECS Engine}

The ECS event log file format is flexible and contains several optional line types. However, the files generated by the ECS Engine are fixed format and consist of the following line types:
- A single comment line containing the event's unique ID in the form:
```


# eventid(0:1)

```

The unique ID is assigned to the event by the ECS Engine and may be used in conjunction with the audit trail information to extract events of interest from the event log files.
- The comment is followed by the time control line. This line indicates the number of seconds that elapsed between logging the previous event in the file and this one:
+0
E ngine event output logs always record an increment of 0 seconds.
- The next line specifies a repeat count of 1 . Engine event output logs always specify a repeat count of 1 :
\(!1\)
- The repeat line is followed by the event itself in the appropriate format. The event may extend over several lines:

1997/01/03 12:03:00,Thor, LNKDN, 5, "Link Down:Vili"
- The event is terminated by an event syntax line defining the endecoder and syntax:
\% mdl:SimpleEvent:

\section*{Event Log File Format}

\section*{Log File Syntax}

Engine output event logs do not include the transit delay. Input event logs may specify the actual transit delay.
A typical example event log file, generated by the ECS Engine output event log, with just two events is:
```


# eventid(0:1)

+0
!1
1997/01/03 12:03:00,Thor,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:

# eventid(0:2)

+0
!1
1997/01/03 12:03:01,Oden,LNKDN,5,"Link Down:Vili"
% mdl:SimpleEvent:

```

\section*{Annotation Logs}

Annotation requests and responses are read and written in the form of event log entries. Each request or response appears like an event, with one attribute (responseList) containing the request or response information, as appropriate, in the form of a List.
Annotation requests are written by both the ECS Designer and the ECS Engine. The ECS Designer writes annotation requests to the Output Events window during simulation whenever an Annotate node is triggered. The contents of the Output Events window can then be saved to a file.

The ECS Engine logs annotation requests and responses like any other event, with the one exception that it logs requests in the output event log and responses in the input event log.

The ECS Designer is selective about which events are loaded from a log file. The selection of events depends on the function chosen from the Simulate menu:
- Simulate:Load Input Events. Annotation responses are ignored; other events are input to the engine.
- Simulate:Load Annotation Events. All events except annotation responses are ignored; annotation responses are read and used to simulate responses received from an annotation server.

This selective behavior means that you can use the same ECS Engine input log (containing normal input events as well as annotation
responses) for both inputs to the ECS Designer.

\section*{The Annotation E vent Format}

The annotation event format consists of six attributes, each on a separate line. The six attributes are:
\begin{tabular}{ll} 
requestId & \begin{tabular}{l} 
The unique annotation request ID generated by the \\
ECS Engine when a request is initiated.
\end{tabular} \\
circuitName & \begin{tabular}{l} 
The name of the loaded circuit which generated the \\
annotation request.
\end{tabular} \\
nodeName & \begin{tabular}{l} 
A fully qualified name for the annotation node which \\
generated the annotation request.
\end{tabular} \\
sequenceId & \begin{tabular}{l} 
The uniquelD generated by the annotation node which \\
initiated the request. This ID is unique for the \\
annotation node which generated it only.
\end{tabular} \\
expiryTime & \begin{tabular}{l} 
The time at which the annotation request will expire.
\end{tabular} \\
responseList & \begin{tabular}{l} 
The annotation response values encoded as an ECS \\
List.
\end{tabular}
\end{tabular}

In addition, the event syntax line for an annotation request is:
\% anno:request:
And for an annotation response it is:
```

% anno:response:transitDelay

```

Where transitDelay is the number of seconds by which the response is to be delayed during a simulation (expiryTime is ignored during a simulated annotation response). For example, a complete annotation response would appear as follows:
```


# My annotation event

0
simulate
anno_reg_001_module.tmpcompound.tmpnode.annonode
2
19970103010400.000000Z
["Link Down:Vili", (5, "Thor"), 1.230000, 1]
% anno:response:61

```

This event would appear in the ECS Designer (with View: ECDL-Like Events selected) like this:

\section*{Event Log File Format}

\section*{Log File Syntax}
```

-- response -- (
-- requestId -- 0,
-- circuitName -- simulate,
-- nodeName
anno_reg_001_module.tmpcompound.tmpnode.annonode,
-- sequenceId -- 2,
-- expiryTime -- 19970103010400.000000Z,
-- responseList -- ["Link Down:Vili", (5, "Thor"), 1.230000,
1]

```
    )

\section*{Special Characters}

There are five special characters in event log files:
- The four characters used to define the event log line types (\% + ! and \#)
- and the reserved character @

If it is possible that your event text could start with one of these characters then you need to take special action as described here.

The @ character is reserved for future use. Where an event may start with this character you must insert a line consisting of a single e character after the comment lines or as the first line if there are no comments.

Where an event may start with a \#precede the event with a time control line:
```

+0

```
\#1,"LNKDN","Vili", 23/2/97
\% mdl:SimpleEvent:

Where an event may start with a + precede the event with a repeat control line:
```

!1
+1,"LNKDN","Vili", 23/2/97

```
\% mdl:SimpleEvent:

Where an event may start with a ! precede the event with a repeat control line:
! 1
!1,"LNKDN","Vili", 23/2/97
\% mdl:SimpleEvent:
The first line starting with! is interpreted as the repeat control line. Any following lines that start with a ! are not treated specially as only one repeat control line is allowed.

Where any line of an event may start with a \% followed by a space, a slightly more complex solution is required which redefines the lead-in string for the event syntax control line. This is done using a special comment, as in the following example:

\section*{Event Log File Format \\ Special Characters}
```


# A comment

# SYNTAX START:MyEventSyntax:

+0
!1
%1,"LNKDN","Vili", 23/2/97
MyEventSyntax:mdl:SimpleEvent:

```

This example redefines the default syntax control line lead-in sequence from "\% ". This is done with the special comment \# SYNTAX START: which defines the remainder of the comment line to be the replacement lead-in. In this case the lead-in string is defined to be MyEventSyntax: (including the terminating character : ).

This example would allow events to start with any character except the string sequence MyEventSyntax: .

\section*{12 Audit Logging}

This chapter describes the basis of the ECS audit logging facility. This is an advanced chapter that assumes that you are already familiar with ECS, ECDL and operation of the ECS Engine.

The first section introduces the key concepts:
- "Verifying the Operation of a Circuit" on page 389

This is followed by a description of the function used to write information to the audit log:
- "The audit_log Function" on page 390

Finally, some helpful design hints:
- "Designing an Audit" on page 392

You may also need to refer to details on the operation of the ECS Engine in order to enable the appropriate logging facilities and log levels:
- HP OpenView Event Correlation Services Administrator's Guide

\section*{Verifying the Operation of a Circuit}

Sometimes the operation of an ECS circuit needs to be verified. Two specific verification requirements are:
- prove that an ECS circuit is operating as expected
- identify the input events that caused a particular event to be output (or that caused a particular event not to be output).

Auditing is the process of enhancing an ECS circuit by inserting extra ECDL statements to record critical data in a log, and then analyzing the log file. The audit expressions that you add are inserted into ECDL expressions wherever important data is required and wherever decisions are made in the circuit. For example, it may be important to know the time at which a circuit is loaded, and to record every occasion on which an event is suppressed by a Filter node.

Engine log Audit log information is written to the ECS Engine log file:
\$OV_LOG/ecs/instance/ecsd.log0
During a simulation, you can view the enginelog in the ECS Designer by selecting Simulate: Engine Log from the menu.

For audit statements to be recorded in the ECS Engine log, you must enable the ETL_LOGINFORM level of logging. If you are running the pmd-linked version of the ECS Engine then you must also enable the postmaster log:
\$OV_LOG/pmd.log0
Usually, you will also need to enable event logging in the ECS Engine. See the HP OpenView Event Corrdation Services Administrator's Guide for information about setting logging mask values and turning on input event logging.

The principle on which auditing is based is that you, the circuit designer, must have complete control over the information recorded in the log. This allows you to reduce the overheads involved in auditing to the minimum necessary to satisfy operational requirements.

\section*{The audit_log Function}

Audit records are written by the System. audit_log function using an expression such as:
system. audit_log "event forwarded"
This expression prints the text event forwarded to the engine log file.
The system. audit_log built-in function takes one argument, converts it to readable form if necessary, and prints it to the engine log if logging is in effect. The return value is Void, and this creates a potential problem for most node parameter expressions where you want the expression to return an appropriate value.

To ensure that the appropriate value is returned you must use a sequence expression. Sequences allow multiple expressions to be strung together and executed in the sequence in which you write them. The value of the last expression in the sequence is returned as the value of the sequence as a whole. See Chapter 10, "Writing ECDL Expressions," on page 319 for further details about sequence expressions.

To use a sequence expression you simply separate each expression with a semicolon. Generally, it is also a good idea to enclose the whole lot in parentheses to ensure the sequence is evaluated as a block:
( system.audit_log "event forwarded" ; true )
In this case the audit_log function is executed first, followed by the expression true, which simply returns the value true. The Void value returned by the audit_log function is discarded.

A more realistic example that you might place in a Filter node to pass only those events with a severity of 3 is:
```

if (input_event "severity" = 3)
then
(system.audit_log "event forwarded" ; true)
else
(system.audit_log "event deleted" ; false)
end

```

The examples used so far simply place a text message in the log. However, you can alsolog values and even entire events. For example, to audit the action of an Unless node configured to suppress duplicate events (same "deviceId" but different unique_id), you could write:
```

if ( inhibitor_event "deviceId" = input_event "deviceId"
and
inhibitor_event unique_id != input_event unique_id
)
then(
system.audit_log ("DuplicateGroup",
input_event "deviceId",
"suppressing duplicate event",
input_event unique_id)
; true
)
else
false
end

```

Here, the collection of four elements to be logged is turned into a Tuple by placing parentheses around the collection and separating elements with commas. The audit_log function logs the Tuple and decomposes it into simple data items which it then converts to text. The first and third elements are simply text, but the second element extracts the "deviceId" attribute from the input event and the last element logs the unique ID of the input event.

\section*{Using Unique I Ds to Lookup E vents Details}

Frequently, you will want to record an ECS event in the audit log. To do this you must enable ECS Engine input event logging. This records all events that enter the ECS Engine, together with each event's uniqueID (see "Logs Generated by the ECS Engine" on page 381).

When you record an event in the log file, instead of writing out the entire event details, the audit_log function records just the event's unique ID, which you can match with the details recorded in the event log. This is much more efficient than recording complete event details in the audit, possibly repeating the same details many times.

This referencing technique works because an event can never be changed in ECS. Even the M odify node creates a copy of the input event (with a new uniqueID) and modifies that instead of the original event.

\section*{Designing an Audit}

To validate an entire ECS circuit you need to write an audit log entry for each major decision affecting an event:
- on entry to the circuit
- just before it is suppressed
- as each decision is taken
- as an event is created (Create node) or modified
- on output from the circuit

Events that can be ignored (not recorded in the log) include those events that are incidental and play no part in the correlation.
In some cases the log will be analyzed by another program, such as a general purpose utility like grep(1), a database or spreadsheet application, or a specially developed application capable of more sophisticated analysis. In any case, you need to think about what information should be placed in the log and, if necessary, coordinate the design of the log format with the requirements of the external application that will read it.

\section*{Static and Dynamic Evaluation}

Audit log information is written to the log whenever an expression containing an audit_log function is evaluated. When designing an audit you need to consider whether a parameter is eval uated statically or dynamically.

Audit expressions added to static parameters are evaluated just once, when the circuit is loaded (such as the Count node Initial count parameter). On the other hand, audit expressions added to dynamically evaluated parameters are evaluated whenever the parameter is evaluated. For example, the Filter node condition parameter is evaluated whenever an event enters the node. Some parameters (such as the Delete Condition in a Table node) are evaluated very frequently and can generate huge numbers of audit log entries if you place audit log expressions in them.
See the description of each node in Chapter 3, "Circuit Nodes," on
page 43 for information about whether a particular parameter is evaluated statically or dynamically.

\section*{Audit Logging \\ Designing an Audit}

\section*{13 Files and Directories}

\section*{Files and Directories}

This chapter describes the files read and written by the HP OpenView Event Correlation Services Designer.

\section*{Files Used or Generated by the ECS Designer}

The directory paths listed in Table 13-1 assume that you have defined the Universal Pathname environment variables set with the ov. envvars. sh script. The locations used by ECS are listed in Table 13-2 on page 399. See the HP OpenView Event Correlation Services Administrator's Guidefor further details.

Default names and The ECS engine itself does not require particular filename suffixes. locations However, the ECS Designer expects files to have certain suffixes and to be located in certain directories.

Unless the pathname is specified, all ecsmgr(1M) commands that specify file names assume that files are to be loaded from or saved to the present working directory.
Table 13-1 File Names and Default Locations
\begin{tabular}{|l|l|l|}
\hline File & Location & Description \\
\hline *.ecs & \$OV_DB/ecs/ & \begin{tabular}{l} 
Uncompiled ECS circuit. This is \\
just the default location. Files can \\
be read from or written to any \\
location to which the user has \\
access.
\end{tabular} \\
\hline \begin{tabular}{l} 
ecs_defaults.ecs, \\
cmip_defaults.ecs, \\
and nnm_defaults.ecs
\end{tabular} & \$OV_CONF/ecs/modules/ & \begin{tabular}{l} 
Default modules installed when \\
ecsdes is installed
\end{tabular} \\
\hline *.eco & User defined & User defined
\end{tabular} \begin{tabular}{l} 
Compiled ECS circuits \\
\hline *.ds
\end{tabular}

Table 13-1 File Names and Default Locations
\begin{tabular}{|l|l|l|}
\hline File & Location & Description \\
\hline \begin{tabular}{l} 
pmd.log0 \\
and \\
pmd.log1
\end{tabular} & \$OV_LOG & Engine log files (DM only) \\
\hline \begin{tabular}{l} 
pmd.trc0 \\
and \\
pmd.trc1
\end{tabular} & \$OV_LOG & Engine trace files (DM only) \\
\hline \begin{tabular}{l} 
ecsin.evt0 \\
and \\
ecsin.evt1
\end{tabular} & \$OV_LOG/ecs/instance/ & \begin{tabular}{l} 
Engine input event logs generated \\
by the ECS Engine.
\end{tabular} \\
\hline \begin{tabular}{l} 
ecsout.evt0 \\
and \\
ecsout.evt1
\end{tabular} & \$OV_LOG/ecs/instance/ & \begin{tabular}{l} 
Engine output event logs generated \\
by the ECS Engine.
\end{tabular} \\
\hline \begin{tabular}{l} 
circuit_name.evt0 \\
and \\
circuit_name.evt1
\end{tabular} & \$OV_LOG/ecs/instance/ & \begin{tabular}{l} 
Output event log for the specified \\
circuit, generated by the ECS \\
Engine. instance is the instance \\
number of the ECS engine
\end{tabular} \\
\hline ecsd.conf & \$OV_CONF/ecs/instance/ & \begin{tabular}{l} 
Configuration file for the ECS \\
Engine. See manpage for details.
\end{tabular} \\
\hline mdl.md & \$OV_CONF/ecs/md/mdl/ & \begin{tabular}{l} 
Metadata file used by the MDL \\
endecoder.
\end{tabular} \\
\hline ecs.* & \$OV_CONF/ecs/md/ber/ & \begin{tabular}{l} 
Metadata file used by the BER \\
endecoder.
\end{tabular} \\
\hline ed.conf & \$OV_CONF/ecs/ed/ & \begin{tabular}{l} 
Configuration file listing the \\
endecode modules used by the \\
ECS Engine.
\end{tabular} \\
\hline
\end{tabular}

Where instance is a number that identifies the ECS Engine instance.
The ECS Engine always writes to log and trace files with the suffix 0 . These files are renamed with a suffix of 1 , overwriting any existing file with the same name. Renaming happens on startup and whenever the engine log file size limit is reached (see the HP OpenView Event Correlation Services Administrator's Guide). The ECS Designer uses event log files without any suffix, so you must rename log files generated by the ECS Engine before they can be used in the ECS Designer.
The environment variables installed by default vary depending on the operating system used, as described in Table 13-2.

\section*{Table 13-2 Predefined Environment Variables for File Locations}
\begin{tabular}{|c|c|c|c|}
\hline Environment Variable & Windows NT 4.x & HP-UX 10.x, 11.x & Solaris 2.x \\
\hline \$OV_LOG & C: \Openview \(\backslash\) log & ```
/var/opt/OV/share/
log
``` & ```
/var/opt/OV/share/
log
``` \\
\hline \$OV_CONF & C: \Openview \} conf & /etc/opt/OV/share/ conf & /etc/opt/OV/share/ conf \\
\hline \$OV_DB & C: \Openview\} databases & \begin{tabular}{l}
/var/opt/OV/share/ \\
databases
\end{tabular} & \begin{tabular}{l}
/var/opt/OV/share/ \\
databases
\end{tabular} \\
\hline \$OV_MAIN_PATH & \(\mathrm{C}: \backslash\) Openview & /opt/ov & /opt/ov \\
\hline
\end{tabular}

Files and Directories
Files Used or Generated by the ECS Designer

\section*{A \\ Event Correlation in OVO}

This appendix provides detailed, low-level information designed to help the OVO administrator better understand the design and configuration of event-correlation circuits in the context of OVO. It also gives some hints on how to go about investigating problems relating to OVO-specific, event-correlation circuits. It is divided into the following parts:
- Event or Message attributes
- Event logging
- Troubleshooting OVO-specific problems

\section*{OVO Message Attributes}

Event (or "message" in the context of OVO) attributes are referenced in string form in ECS circuit nodes. For example, a Filter node in an ECS circuit would refer to the OVO message attribute "OBJ ECT" in the following way: input_event OBJECT. Similarly, the "CREATION_TIME" message attribute could be accessed as: input_event CREATION_TIME. See Table A-1 for more information on the OVO-specific event attributes that you may use for event correlation in the context of OVO.

See Table 4-1 on page 152 for OVO primitive event header attribute values.
Table A-1 OVO Primitive Event Body Message Attributes
\(\left.\begin{array}{|l|l|l|}\hline \text { OVO Message Attribute } & \text { Type } & \text { Description } \\ \hline \text { AACTION_ACK } & \text { Boolean } & \begin{array}{l}\text { Defines whether or not the message is acknowledged } \\ \text { automatically on the OVO management server after the } \\ \text { corresponding automatic action has finished successfully. }\end{array} \\ \hline \text { AACTION_ANNOTATE } & \text { Boolean } & \begin{array}{l}\text { Defines whether or not OVO creates "start" and "end" } \\ \text { annotations for automatic actions. }\end{array} \\ \hline \text { AACTION_CALL } & \text { String } & \begin{array}{l}\text { The command to use as automatic action for the OVO } \\ \text { message. Default: empty string; max. length: 254 chars. }\end{array} \\ \hline \text { AACTION_NODE } & \text { String } & \begin{array}{l}\text { Defines the system on which the automatic action runs. } \\ \text { Default value: the content 'NODENAME'; max. length: } \\ 254 \text { chars }\end{array} \\ \hline \text { AACTION_STATUS } & \text { Integer } & \begin{array}{l}\text { Defines the status of the automatic action belonging to the } \\ \text { current message. Possible values are: }\end{array} \\ \text { - ACTION_UNDEF (default) } \\ \text { • ACTION_DEF (default if AACTION_CALL is } \\ \text { defined) }\end{array}\right\}\)

\section*{Table A-1 OVO Primitive E vent Body Message Attributes}
\begin{tabular}{|c|c|c|}
\hline OVO Message Attribute & Type & Description \\
\hline APPLICATION & String & Application name to use for the OVO message. Default: empty string; max. length: 32 chars. \\
\hline CREATION_TIME & Time & The time the message was created. The time is in UNIX format (seconds since Epoch). Default: the (local) time when the message was created. \\
\hline GROUP & String & The OVO message group to use for the message. Default: empty string; max. length: 32 chars. \\
\hline INSTR_IF & String & The name of the external, instruction-text interface. The external, instruction-text interface must be configured in OVO. Default: empty string; max. length: 36 chars. \\
\hline INSTR_IF_TYPE & Integer & \begin{tabular}{l}
Defines whether the internal OVO instruction-text interface or an external interface is used to display instructions for the message. Possible values are: \\
- INSTR_NOT_SET (default) \\
- INSTR_FROM_OPC \\
- INSTR_FROM_OTHER
\end{tabular} \\
\hline INSTR_PAR & String & Parameters for the call to the external, instruction-text interface. Default: empty string; max. length: 254 chars. \\
\hline MSGID & String & Read only. Unique identifier of the message. Modified or newly created messages will assume the ID: ' 00000 ....' \\
\hline MSGSRC & String & Read only. This attribute specifies the source of the message, e.g., the name of the encapsulated logfile if the message originated from logfile encapsulation or the interface name if the message was sent via an instance of the Message-Stream Interface. Default: empty string; no max. length. \\
\hline
\end{tabular}

\section*{Table A-1 OVO Primitive Event Body Message Attributes}
\begin{tabular}{|c|c|c|}
\hline OVO Message Attribute & Type & Description \\
\hline MSGSRC_TYPE & Integer & \begin{tabular}{l}
Read only. Specifies the source type of the message. Each source is represented in one bit, e.g. a message that was generated by the logfile encapsulator and then modified at the Agent MSI will have 'bitor LOGFILE_SRC AGTMSI_SRC' set. \\
Possible values are: \\
- OPCMSG_SRC \\
- LOGFILE_SRC \\
- MONITOR_SRC \\
- SNMPTRAP_SRC \\
- SCHEDULE_SRC \\
- CONSOLE_SRC \\
- SVMSI_SRC \\
- AGTMSI_SRC \\
- LEGLINK_SRC
\end{tabular} \\
\hline MSGTEXT & String & Message text. Default: empty string; no max. length. \\
\hline MSGTYPE & String & This attribute is used to group messages into subgroups, e.g., to denote the occurrence of a specific problem. Default: empty string; max. length: 36 ASCII chars, no spaces. \\
\hline MSG_LOG_ONLY & Boolean & Inserts the message immediately into the history-message table when the message is received on the Management Server. The message is not sent to any operator. An Operator will only be able to see the message when using the OVO history message browser \\
\hline
\end{tabular}

\section*{Table A-1 OVO Primitive E vent Body Message Attributes}
\(\left.\left.\begin{array}{|l|l|l|}\hline \text { OVO Message Attribute } & \text { Type } & \text { Description } \\ \hline \text { MSI_OUTPUT } & \text { Integer } & \begin{array}{l}\text { Defines the handling of the message in the OVO Message } \\ \text { Stream Interface. Each value is representing one bit that } \\ \text { can be bitor'ed. Possible values are: } \\ \text { - SV_MSI_NO_OUTPUT (default) }\end{array} \\ \hline \text { VODENAME } & & \begin{array}{l}\text { - SV_MSI_DIVERT } \\ \text { - } \\ \text { - SV_MSI_COPY }\end{array} \\ \hline \text { AGT_MSI_NO_OUTPUT (default) }\end{array}\right\} \begin{array}{l}\text { AGT_MSI_DIVERT }\end{array}\right\}\)

\section*{Table A-1 OVO Primitive Event Body Message Attributes}
\begin{tabular}{|c|c|c|}
\hline OVO Message Attribute & Type & Description \\
\hline OPACTION_STATUS & Integer & \begin{tabular}{l}
Defines the status of the operator-initiated action belonging to the current message. Possible values are: \\
- ACTION_UNDEF (default) \\
- ACTION_DEF (default if OPACTION_CALL is defined) \\
- ACTION_STARTED \\
- ACTION_FINISHED \\
- ACTION_FAILED
\end{tabular} \\
\hline ORIGMSGTEXT & String & The original message text. This attribute allows you to set additional source information for a message. It is useful if the message text was reformatted but the OVO operator needs to have access to the original text. Default: empty string; no max. length. \\
\hline RECEIVE_TIME & Time & Read only. The time the message was received by the management server. The time is in UNIX format (seconds since Epoch). This value is set by the management server. \\
\hline SEVERITY & Integer & \begin{tabular}{l}
The severity of the message. Possible values are: \\
- SEV_UNKNOWN \\
- SEV_NORMAL \\
- SEV_WARNING \\
- SEV_MINOR \\
- SEV_MAJOR \\
- SEV_CRITICAL
\end{tabular} \\
\hline TROUBLETICKET & Boolean & Forwards OVO messages from the OVO management server to the OVO trouble-ticket (TT) interface, if the TT interface is configured. \\
\hline TROUBLETICKET_ACK & Boolean & Defines that the OVO management server acknowledges the message automatically if forwarding of the message to the trouble ticket system was successful. \\
\hline
\end{tabular}

\section*{Table A-1 OVO Primitive E vent Body Message Attributes}
\begin{tabular}{|l|l|l|}
\hline OVO Message Attribute & Type & Description \\
\hline UNMATCHED & Boolean & Defines whether or not the message matched a condition. \\
\hline
\end{tabular}

All attributes can be read and altered (in a Create or Modify node) except for those marked read-only.

\section*{Using E vent Attributes}

OVO event attributes can be addressed as String data types, or as Tokens. This means, for example, that you can address the OBJ ECT message attribute using the String value "ОВЈЕСТ" or the Token value OBJECT. In practise this means you can either use quotes (String form) or not (Token form).

Using the Token address form has the advantage that a misspelled token name is picked up when the circuit is validated (compiled) rather than at run time. Token addresses are also more efficient than Strings. You should al so remember that case is important with Strings, but is not important when Tokens are used. So OBJECT, object and "OBJECT" are acceptable, but "object" will raise a run time error.
All other ECS documentation assumes that the attributes of primitive events are addressed as String values.

\section*{Examples}

To filter events based on their severity you could enter a Filter node Condition parameter such as:
input_event SEVERITY = SEV_UNKNOWN
To change messages with a severity of SEV_UNKNOWN to SEV_NORMAL you could use a Modify node after the Filter node above, with a Modify Spec parameter such as:
```

modified_event alter (SEVERITY => SEV_NORMAL)

```

To retain just the latest event from a particular OBJ ECT in a Table node, you could enter a Delete Condition parameter such as:
```

current_event OBJECT = retained_event OBJECT

```

\section*{Logging E vents in OVO}

The OVO event correlation processes on the management server and the managed node log messages going into and out of the ECS Engine in the files ecevilg and ecevolg respectively. These logfiles are used for testing and debugging and reside in the following directories:

\section*{Management Server}

HP-UX 10.x /var/opt/OV/log/OpC/mgmt_sv

\section*{Managed Node}

HP-UX 10.x /var/opt/OV/log/OpC
HP-UX 9.x /usr/OV/log/OpC
For example, the ECS Designer uses the logfile ecevilg as its simulation input. You need to transfer the file manually from the managed node to the server if you wish to simulate a circuit's operation. N ote al so that the ECS Designer requires a .evt suffix for the event logfiles.
In order to rebuild an original message from the content of the ECS logfile, the logfile must contain as much information as possible relating to the messages that passed through the correlation process.
Consequently, the structure of all OVO message logfiles are designed to ensure that they contain the required message attributes in the appropriate format.

You can switch on the logging of input and output events in the "Options" window, which you access from the "M essage Source Templates" window. Switching on logging for any one of the EC templates you assign and distribute to a managed node means that logging is switched on for all EC templates on that managed node.

\section*{Event Log Format}

In the ECS Designer event display windows, and in the event log file itself, a typical OVO message appears as shown below:

\section*{Logging Events in OVO}
```

> Security ; /usr/bin/su(1) Switch User ; frankv
> ; :
> INSTR_NOT_SET: ;
> AA: ; ; ; Undef
> AA:
> OA: ; ;
> OA:
> Succeeded switch user to oracle by frankv
> SU 03/19 16:43 + ttyp6 frankv-oracle
% OpC_Msg::

# 0858786227

+0

```

The values shown in the message above are matched to OVO message attributes as shown below. The notation [OPACTION_ACK: "АСК" | ""] means: depending on the OPACTION_ACK attribute, the log contains either ACK or nothing; no quotes are present in the log file itself.
```

> [MSGID]
> [CREATION_TIME] ; [CREATION_TIME (in ASCII)]
> [NODENAME] ([net_type: usually IP]) ; [SEVERITY] ;
[MSG_LOG_ONLY: "LOG_ONLY"|""]
> [UNMATCHED: "Matched"|"Suppressed"|"Unmatched"] ; [MSGTYPE] ;
"Console"|"OpC"|"Logfile"|"Monitor"|"SNMP"|"MSI": [MSGSRC]
> [GROUP] ; [APPLICATION] ; [OBJECT]
> [NOTIFICATION: "NOTIFICATION"|""] ; [TROUBLETICKET: "TT"|""]:
[TROUBLETICKET_ACK: "ACK"|""]
> [INSTR_IF_TYPE:
"INSTR_NOT_SET"|"INSTR_NOT_SET"|"INSTR_FROM_OTHER"] :
[INSTR_IF] ; [INSTR_PAR]
> AA: [AACTION_NODE] ; [AACTION_ACK: "ACK"|""] ;
[AACTION_ANNOTATE: "ANN"|""] ; [AACTION_STATUS:
"Undef"|"Def"|"Started"|"Finished"| "Failed"]
> AA: [AACTION_CALL]
> OA: [OPACTION_NODE] ; [OPACTION_ACK: "ACK"|""] ;
[OPACTION_ANNOTATE: "ANN"|""] ; [OPACTION_STATUS:
"Undef"|"Def"|"Started"|"Finished"| "Failed"]
> OA: [OPACTION_CALL]
> [MSGTEXT]
> [ORIGMSGTEXT]
% OpC_Msg::(time difference between log time and CREATION_TIME)

# (log time in seconds since epoch)

(time difference to next message)

```

Where:
> Marks the start of a new line.
\begin{tabular}{ll}
; & Separates attributes on the same line. \\
XXX & Literal text that appears in the message. \\
[ . . ] & \begin{tabular}{l} 
A message attribute value, as described in Table A-1 on \\
page 403. The square brackets may encl ose a list of \\
literal values, but are not present in the message itself.
\end{tabular} \\
"..." & \begin{tabular}{l} 
A named value (the quotes are not present in the \\
message itself - empty quotes "" indicate that a \\
missing value is acceptable.).
\end{tabular} \\
I Separates alternative values. \\
(...) & \begin{tabular}{l} 
Encloses a descriptive comment.
\end{tabular}
\end{tabular}

\section*{Troubleshooting OVO-specific Problems}

There are a number of options you can choose when investigating EC-related problems in the context of IT/Operations. The problems you encounter can concern the malfunction of either the ECS circuits and templates you design and distribute or the ECS engine itself. First of all you need to establish whether the following conditions are true:
- MSI output must be enabled on the management server and the managed node. You enable MSI output by opening the following sequence of windows:

Management Server

Managed Node

Actions -> Server ->Configure: MSI - Enable Output

Actions -> Node->M odify -> Advanced Options: MSI - Enable Output
- Output to the MSI must be enabled and the option "Divert/Copy M essages" set for each of the message conditions that output messages to the event-correlation engine. You do this by opening the following sequence of windows: Window ->M essage SourceTemplates ->Conditions ->Condition Number \#->Advanced Options: MSI Divert/Copy Messages
- The "message-type" attribute must be set correctly for each message condition. The OVO administrator defines message types: a message type is usually the name of the sub-group to which the message belongs. You set the "message-type" attribute by opening the following sequence of windows: Window \(->\) M essage Source Templates->Conditions -> Condition Number \#. Message Type
- The message attribute you specify in the "message-type" attribute field for a given message condition must match the message attribute defined in the event-type field of the appropriate event-correlation, input node. The appropriateECS input node is the one that starts the event-correlation flow which you want to process the message produced by a given message condition.

If all these conditions are true, then the message should pass through the correlation process as expected. However, if it is the case that the expected behavior means that the message is discarded during the correlation process, then verifying whether or not the event-correlation process is running correctly is more difficult.
The easiest way to find out whether the message is being generated by the template in the first place (since if it were indeed being generated by the template but then correctly discarded by the event-correlation process, it would not appear in the "Message Browser" window) is either to disable the output to the MSI (Actions -> Server -> Configure or Actions \(->\) Node \(->\) M odify) so that the message passes directly to the Message Browser, or set the "Divert/Copy" message to MSI switch to "Copy".

\section*{NOTE \\ Make sure that you redistribute the message source template after changes in conditions such as toggling the "Divert/Copy message to MSI" switch.}

The result of this is that either the original message or a copy of it will appear in the "Message Browser window". If the message does appear in the "M essage Browser" window when the output to the MSI is switched off, you can concentrate further investigation on the event-correlation process itself.

\section*{Event Correlation in OVO}

\section*{Troubleshooting OVO-specific Problems}

\section*{Glossary}

\begin{abstract}
Syntax Notation 1
(ASN.1) An OSI standard related to the Presentation Layer where the abstract representation of the data is independent of its physical encoding. It is specified in ISO/IEC 8824, X. 208.
\end{abstract}
agent A program or process running on a remote device or computer system that responds to management requests, performs management operations, and/or sends event notifications.
annotation API A set of application program interface functions and data structures that supports the transfer of data between an external annotaton server and one or more Annotate nodes in an ECS circuit.
annotation server A user supplied server that receives a request from an Annotation node within a correlation circuit, performs some action, and returns a response to the Annotate node. The action performed by the annotation server may involve information extracted from events in the circuit, and the information returned is typically obtained external to the ECS Engine and the annotation server.
arrival time The time an event arrives at the ECS engine in Universal Coordinated Time (UTC).

ASCII American Standard Code for Information Interchange. A standard used by computers for interpreting binary numbers as characters.

ASN. 1 Abstract Syntax Notation 1.
attribute An object characteristic or property that describes the current state of the object and which has a unique identifier by which it is accessed. In ECS, for example, the "eventTime" attribute of a CMIP event, or the "Rate" attribute of a Rate node. See event attribute; identifier; correlation node attribute.
attribute-value pair The combination of an attribute identifier and the value of that attribute for a specific object. In ECS, attribute-value pairs are represented as key-value pairs in an ECDL dictionary. See also key-value pair; dictionary.

\section*{Basic Encoding Rules (BER)}

Defines how ASN. 1 data types are encoded for transport on the network.
breakpoint A point in a program at which execution is halted so that the program's status, contents of variables and other factors can be examined. In the ECS Designer, in simulation mode, breakpoints are locations in a correlation circuit where event processing is halted to allow for manual intervention.
canvas The working area of the ECS Designer screen. This is where you place, connect, and configure correlation nodes to create your correlation circuit.

CCITT The International Telegraph and Telephone Consultative Committee, an international organization concerned with proposing recommendations for international communications. Replaced by the International Telecommunications Union, Telecommunications (ITU-T) in 1992. See I nternational Tel ecommunications Union, Telecommunications (ITU-T).
circuit Secorrelation circuit.

CMIP SeCommon Management Information Protocol (CMIP).

\section*{Common Management Information Protocol (CMIP) A} protocol for exchanging network management information in an OSI environment (ISO/ITU-T X.710). CMIP communicates management information between a manager and an agent. CMIP allows a manager to retrieve (get) management information from, or to alter (set) management information on an agent. CMIP also allows the manager to create and delete instances of an object managed by the agent, or perform an action on an object. An agent can also emit unsolicited messages, called notifications, to alert managers of noteworthy local conditions.
component event An event that is combined with other events to create a new event. In ECS, a composite event is composed of two or more component events. See composite event.
composite event In ECS, a composite event consists of a structured aggregation of addressiblecomponent events each of which may be a primitive event, a temporary event, or a composite
event. A composite event may only exist within a correlation circuit. See also component event; primitive event; temporary event.
compound node A graphical element that represents a container of lower level components. The lower level components will be displayed when the user opens the compound node. In ECS, a correlation circuit fragment may be encapsulated in a compound node, hence creating a new user-defined correlation node. Compound nodes may be added to libraries and re-used by reference or by copy. Compare with primitive node.
condition (parameter) In ECS, a condition is an ECDL expression specified for a correlation node parameter, usually invol ving attribute from an event, that returns a value used to modify the behavior of the correlation node.
correlation A procedure for evaluating the relationship between sets of data or objects to determine the degree to which changes in one are accompanied by changes in the other. In ECS, correlation is a process of analyzing a stream of events by filtering and detecting patterns
and replacing groups of events with single events that have (possibly) higher information content.
correlation circuit In ECS, a collection of interconnected primitive nodes and compound nodes, configured to perform a filtering or correlation activity. Each correlation node is configured appropriately to the correlation requirement. The configuration includes the specification of the event types, and the allowed transit delays for those events, to be accepted from the external event stream. A correlation circuit can be loaded into an ECS Engine.
correlation circuit port The logical connections between a correlation circuit and the containing infrastructure where events enter and leave the circuit. These ports may be configured to select a subset of events in the input event stream, based upon event encoding type and event syntax. A single port may be connected to multiple Source/Sink nodes, and a single Source/Sink node may be connected to multiple circuit ports.
correlation engine The ECS
runtime component that reads an input event stream, decodes the input events, performs the event correlation, encodes the output events and returns the output events to the event stream. The event correlation is as specified by the one or more correlation circuits loaded into the correlation engine.
correlation node A processing element in a correlation circuit. See also compound node; primitive node.

\section*{correlation node attribute \(A\)}
property of a correlation node that can be read from another correlation node. The Count, Rate, and Table nodes have attributes (which may be exported by a containing compound node as attributes of the compound node). Attributes are addressed using a dot notation:
"node_name.attribute_name".
correlation node parameter In the ECS Designer, a correlation node parameter is an ECDL expression used to configure a correlation node.
correlation node port One of possibly many connection points of a correlation node used to
interconnect correlation nodes. Events enter a correlation node through a port and leave a correlation node through a port. Port types include input, output, control, reset, and error ports. In the ECS Designer, ports visually indicate the sense of the associated event flow. Optional ports are not displayed by default.
creation time The time an event was created. Inside the ECS Engine creation time is represented in Universal Coordinated Time (UTC).
daemon A process that "serves" clients. Sometimes referred to as a server.
data store In ECS, a component of the ECS E ngine which holds user-specified named data items of an ECDL data type. The entries in the data store may be referenced from the ECDL expressions configured into the correlation nodes. A correlation circuit may be associated with one of the possibly many data stores loaded into the correlation engine.
data type A particular kind of data; for example integer, alphanumeric, boolean, date. In ECS, data types are ECDL data
types which define the type and range of values to which an identifier may be assigned. Every valuein ECDL has a data type, but the type need not be explicitly stated. The types range from simple types such as integers, to compound types such as dictionaries and lists, and special types such as functions and events.
dictionary (data type) In ECS, a dictionary is an ECDL data type comprised of an unordered list of key-value pairs. Any value is accessed via reference to the key. Within ECS, an event is treated as a dictionary with attribute names being the dictionary keys which provide access to the attribute values.

Distributed Management Platform (DM) HP OpenView Distributed Management Platform, the platform which provides the infrastructure for implementing OSI-based management solutions.

DM SeeDistributed Management Platform (DM)
duration data type In ECS, a duration is an ECDL data type used to represent relative or elapsed time values. Compare with time data type.

\section*{dynamic parameter A}
parameter whose value is determined during program execution. In ECS, an ECDL expression configured for a correlation node parameter which is evaluated each time an event enters the correlation node. Typically, the value returned by a dynamic parameter changes for each event processed.

ECDL SeEvent Correlation Description Language (ECDL).

ECS See Event Correlation Services (ECS).

ECS circuit Secorrelation circuit.

ECS Designer The ECS Designer is the ECS component which you use to create and test correlation circuits. The ECS Designer works in two modes: build mode where you create correl ation circuits, and simulate mode where you test the circuits.

\begin{abstract}
ECS Engine Seecorrelation engine
ecsmgr The command line program used to administer a running ECS Engine.
endecode In ECS, a term used to refer to a combined encoding or decoding function or capability. An endecode module is an architectural entity which provides encoding and decoding for a specific type of event.
evaluation license A license
granted for a specific period of time for the purpose of evaluating ECS.
event An event is an unsolicited notification such as an SNMP trap, a CMIP notification, or a TL1 event, generated by an agent process in a managed object or by a user action. Events usually indicate a change in the state of a managed object or cause an action to occur. In ECS, an event is encoded as a primitive, compound, or temporary event. ECS events contain header attributes added to the input events to assist the processing of the events whilethey are in the ECS correlation circuit. The header attributes are stripped before the events are transmitted from the ECS circuit.
event attribute A characteristic property of an event. In ECS, event attributes are either part of the internally created event header common to all event types, or part of the event body that contains the input event.

\section*{Event Correlation Description} Language (ECDL) The language used to specify correlation circuits (node relationships, parameter expressions, data and fact store values) for the ECS Engine.

\section*{Event Correlation Services} (ECS) The HP OpenView Event Correlation Services product.
event encoding type The first and highest level in the three-tiered ECS event classification system. An event's encoding type determines the endecode module that will be used to translate the event to and from its native format. For example, CMIP notifications and SNMP traps both use the BER encoding type. ASCII events use the MDL encoding type, and ITO messages use the ITO encoding type. See also event syntax; event type
event flow An ECS circuit represented graphically as a circuit schematic consisting of
\end{abstract}
correlation nodes interconnected by lines (connections). See also correlation circuit.
event body The body of an event depends on the event class. The body of a primitive event is the original message, trap or event; the body of a temporary event may be empty; and the body of a composite event consists of other events.
event header Inside ECS and event is augmented with additional information such as the event encoding type, event syntax, event type, and event class. This information is carried in a header that is attached to the event body. See also event body.
event I/O API A set of application program interface functions and data structures that supports the input and output of events to and from the ECS Engine.
event syntax Therules governing the structure and content of an event. In ECS, the event syntax is the second level in the three-tiered ECS event classification system. An event's syntax determines how the event's attributes are read and written. For example, SNMP traps have an event syntax of Trap-PDU
and CMIP notifications have an event syntax that evaluates to an OID identifying the GDMO notification. ASCII events have a syntax determined by the MDL definition used to read and write them. See also event encoding type; event type.
event type A classification of an event into a particular category that further defines the nature of the event. In ECS, the event type is the third and lowest level in the three-tiered event classification system. The event type is represented by the ECS header attribute "event_type". For SNMP traps the event type is the generic trap number (1-6). The CMIP event type is the OID of the notification. ASCII events have an event type determined by the MDL definition used to read and write them. See also event encoding type; event syntax.
expiry time Annotation requests are valid for a limited time, determined by the Annotate node's Time Limit parameter. The expiry time is the time at which the annotation request was generated plus the Time Limit. In other words, it is the time at which the request expires.
expression In general, a set of reserved words, symbols, variables, and functions that is evaluated to provide a result. In ECS, an expression is any collection of valid ECDL statements. Note that ECDL is a functional language that has no concept of variables.
fact store A component of the ECS Engine which stores relationships between objects. Any two objects which may be any ECDL data type, may be related using any user-defined relationship. The facts may be accessed at runtime by the ECDL expressions configured into the correlation node parameters.

FLEXIm A Licensing technology used in stand-alone and DM-integrated ECS products.
floating license A license where there is a single license server for all licensing clients on the network. Any licensing client on the network can access the license server to check out a license.
function A general term for a portion of a program that performs a specific task. In ECS, an ECDL function is one of the built-in functions or operators, or a user
defined function. ECDL functions can be named or anonymous, but must return an ECDL value.

GDMO See Guidelines for the Definition of Managed Objects (GDMO).

\section*{Greenwich Mean Time}

Standard time used throughout the world based on the mean solar time of the meridian of Greenwich. See Universal Coordinated Time (UTC).

\section*{Guidelines for the Definition of Managed Objects (GDMO)}

Describes a formal method for describing the important characteristics and operations of an object class. Specified in ISO 10165-4, X. 722.

HP OpenView A family of network and system management products, and an architecture for those products. HP OpenView includes devel opment environments and a wide variety of management applications.
identifier A name that within a given scope uniquely identifies the object with which it is associated.

IEC International Electrotechnical Commission.

IEEE Institute of Electronic and Electrical Engineers.

\section*{International Telecommunications Union, Telecommunications (ITU-T)}

The ITU is a world-wide organization within which governments and industry coordinate the establishment and operation of telecommunications networks and services. It is responsible for the regulation, standardization, coordination and development of international telecommunications as well as the harmonization of national policies. The ITU is an agency of the United Nations. In 1992 it took over the functions of the CCITT.

ISO International Standards Organization.

ITU-T International
Telecommunications Union, Telecommunications.
key-value pair A data storage item consisting of a search key paired with a value. In ECDL, a key-value pair is written as "key => value". See also dictionary.
library In ECS, a repository for compound nodes. Compound nodes in the library may be referenced from a circuit, or copied from the library and modified.
license The legal right to use a feature in a software program.
license server The server processes that manage access to ECS features by licensed users.
list data type a variable-length ordered set of values all of the same data type. In ECDL, a list data type may contain a set of values of any other ECDL data type including complex types such as lists and tuples.

\section*{Management Information Base}
(MIB) A logical collection of configuration and status values that can be accessed via a network management protocol.

MDL SeMessage Description Language
message description Detailed information about an event or message. In ECS, a description of the attributes and formatting of a text-based event (message), that allows the MDL endecode module to decode and encode events
consistent with that syntax. Message descriptions which are written in Message Description Language (MDL) are translated into metadata before being used by the ECS engine endecode module. See metadata.

\section*{Message Description}

Language A language used to describe a text event's attributes and formatting. Each text event syntax has its own message definition written in MDL. See also message definition; event syntax.
metadata Data about data. In ECS, message descriptions are translated into metadata which is a form which maximizes access performance by the MDL endecode module. See message description. CMIP and SNMP metadata is derived from MIBs.

MIB Management Information Base (MIB).

Network Node Manager (NNM) Definition to come from OVSD.

NNM Se Network Node Manager (NNM).
node 1. A computer system or device (e.g., a printer, router, bridge) in a network. 2. A graphical element in a drawing that acts as a junction or connection point for other graphical elements. 3. In ECS, see correlation node.
nodelock license A license where the license server and license clients must be on the same machine, meaning that the licensed application is "locked" to running on the node that is the license server.
object identifier (OID) A unique sequence of numbers or string of characters used for specifying the identity of an object, that is obtained from an authorized registration authority or an al gorithm designed to generate universally unique values.

OID Seobject identifier (IOD).
oid data type In ECS, an oid is an ECDL data type which contains an Object Identifier in dot-separated notation (e.g., 1.2.3.4.5). Wherethe data item is dynamically interpreted, at least three elements ( 2 dots) are required to avoid interpretation as a real data type.

Open Systems I nterconnection (OSI) A standardization model in which a manager process is responsiblefor executing specific management functions requested by the user through interactions with an agent process. The agent process represents the management services offered by the managed objects.

OSI SeOpen Systems Interconnection (OSI).

OVO HP OpenView Operations, a distributed client/server software solution that helps system administrators detect, solve, and prevent problems occurring in networks, systems, and applications.
parameter In ECS, seecorrelation node parameter.
pmd HP OpenView postmaster daemon.
port 1. A location for passing information into and out of a network device. 2. In ECS, a location for passing events into and out of a correlation node or a correlation circuit. See correlation node port; correlation circuit port.
primitive event An ECS internal event which encapsulates an input event. Several header attributes are added as a header for correlation and control purposes, which are stripped before the primitive event leaves the ECS engine. See also event; temporary event; composite event.
reserved word Words that have special meaning in ECS and cannot be used for any other identifier.

\section*{Simple Network Management Protocol (SNMP) The ARPA network management protocol running above TCP/IP used to communicate network management information between a manager and an agent. SNMPv2 has extended functionality over the original protocol.}
simulate Seesimulation.
simulation In general, the imitation by a program of a process or set of conditions affecting one or more objects such that the results of the program reflect the impact of the process or changes in conditions. In ECS, a simulation is the process of feeding events from an event log file through the correlation circuit to observe the
behavior of the correlation circuit using aids such as breakpoints, tracing, and stepping.

SNMP SeeSimple Network Management Protocol (SNMP).

SNMP trap An unconfirmed event, generated by an SNMP agent in response to some internal state change or fault condition, which conforms to the protocol specified in RFC-1155. See event.
socket stack An interface that supports interprocess communication based on the use of file handles. In ECS a socket stack is used to communicate with the ECS Engine for command, i/o and annotation purposes.

Software Distributor (SD) HP OpenView multi-platform software installation product.
static parameters In general, parameters whose values are determined prior to program execution. In ECS, a statically evaluated parameter is a correlation node parameter where the value is defined when the correlation circuit is loaded. The value does not change when an
event enters the associated node/port. See dynamic parameters.
syntax In general, the rules governing the structure and content of a language or the description of an object. In ECS, see event syntax.

\section*{Telecommunications Management Network (TMN)}

The term used to identify a homogeneous approach to the management of heterogeneous networks. It is defined in the international standards referred to as ITU-TSS M3100. TMN recommendations incorporate OSI NM concepts, principles, protocols and application services.
temporary event In ECS, an event that is created transparently by particular correlation nodes, and which may exist only within a correlation circuit. Temporary events may consist only of header attributes created by the correlation engine, or they may additionally contain user data. Temporary events cannot be transmitted outside the correlation engine. See also event; primitive event; composite event.
time data type An ECDL data type that includes time and date.

TL1 Transaction Language One was developed by Bellcore and is a management system protocol that uses structured text messages to pass information about networks and network element states.

TMN See Telecommunications Management Network (TMN).
transit delay The difference between an event's arrival time and its creation time. Transit delays can be caused by external network delays or by deliberately introduced delays in an ECS circuit.
trap See SNMP trap; event.

\section*{tuple data type An ECDL data} type. A data structure consisting of a fixed collection of elements, where each element is a simple ECDL type or a complex ECDL data type.

\section*{Universal Coordinated Time}
(UTC) Standard time used throughout the world based on the mean solar time of the meridian of Greenwich. Formerly known as Greenwich Mean Time (GMT).
universal pathname \(A\) set of environment variables that describe standard pathnames. Universal pathnames hide variations between pathnames on different versions of Unix.

UTC SeUniversal Coordinated Time (UTC).

\section*{X/Open Management Protocol}
(XMP) An API specified by the X/Open standards body that provides a common access mechanism to both CMIS and SNMP management protocol services.

XMP Sex/ Open Management Protocol (XMP).

Zulu SeUniversal Coordinated Time (UTC).

\section*{Index}
- arithmetic subtraction, 238
- negative sign, 238

\section*{Symbols}
, 244
! not (pattern matching), 289
\#one or more digits (pattern matching), 288
\$ end-of-line anchor (pattern matching), 288
() dict delimiter, 225
() expression sequence delimiter, 339
() tuple delimiter, 326
() void value, 227
* arithmetic multiplication, 240
* tuple data type separator, 326
* zero or morecharacters (pattern matching), 288
*.conf* file suffix, 398
*.ds file suffix
description and default location, 397
required by ECS Designer, 186
*.eco file suffix, 397
*.ecs file suffix, 397
*.evt* file suffix, 397, 398
*.fs file suffix
description and default location, 397
required by ECS Designer, 193
*. \({ }^{\prime}{ }^{*}\) file suffix, 398
*.md* file suffix, 398
*.mdl* file suffix, 398
*.trc* file suffix, 398
+ arithmetic addition, 237
+ positive sign, 237
+string concatenation, 237
, dict key-value pair separator, 225
, list el ement separator, 327
, tuple member separator, 326
. pattern assignment (pattern matching), 290
/ arithmetic division, 241
:: list prepend, 246
; expression separator, 339
=relational equality, 243
\(=\) value binding, 37
=> (map dict key to value), 225
\(>\) relational greater than, 244
\(>=\) relational greater than or equal to, 244
@one or more non-separators (pattern matching), 288
[] list delimiter, 327
[]expression nesting (pattern matching), 289
I character escape (pattern matching), 290
\t tab (pattern matching), 290
^ arithmetic exponentiation, 242
^start-of-line anchor (pattern matching), 288 wildcard (binding pattern), 322
alternative (pattern matching), 289
alternative (typecase expressi ons), 337
alternative (union data types), 231

\section*{A}

AACTION_ACK
ITO message attribute, 403
AACTION_ANNOTATE
ITO message attribute, 403
AACTION_CALL
ITO mess̄age attribute, 403
AACTION_NODE
ITO mess̄age attribute, 403
AACTION_STATUS
ITO mess̄age attribute, 403
addition operator,
arithmetic (+), 237
alter function (dict or Event), 252
alternative operator
pattern matching (| ), 289
typecase expressions (| ), 337
union data types (| ), 231
and function (bool ean), 253, 254
Annotate

\section*{Index}
source of event context, 184
Annotate node
AAAA, 58
allowable output to Sink node, 63
creates composite events, 159
creates temporary events, 157
introduces internal transit delays, 171
introduction to, 58
recursive connection forbidden, 57
transit delays, 56
annotate spec parameter, Annotate node, 61
Annotated node
See also Sink node
annotation
and circuits, 27
Annotate node, 58
event format, 383
logs, 382
requests and responses, logging, 382
selective logging, 382
transit delays, 171
any keyword
dynamic data types, 230
APPLICATION
ITO message attribute, 404
applications and streams, 27
arrival_time
event (header) attribute, 152, 170
ASCII events
create_time, 170, 174
ASN. 1
attribute notation, 155
associativity in ECDL expressions, 351
attributes
circuit vs. Compound node, 35
audit log information
formatting for external applications, 392
when written to the log, 392
audit_log function, 309, 390
auditing
circuit operation, 389
use of unique IDs, 391
autostart parameter
Clock node, 66

\section*{B}
benefits of ECS, 25
BER metadata file name and location, 398
binding patterns
(wildcard), 322
ānd named values, 321
forms of, 322
named values, 321
tuple data type, 322
BindingMismatch exceptions, 342
bitand function (integer), 255
bitinv function (integer), 256, 273
bitleft function (integer), 257
bitor function (integer), 258
bitright function (integer), 259
bitxor function (integer), 260
boolean data type, 215

\section*{C}

C language
introduction to ECDL, 370
case expressions (ECDL), 336
character escape operator, pattern matching ( \(\backslash\) ), 290
choose expressions (ECDL), 334
chr function (string), 261
circuit design and stream policy, 30
circuit designer
skills and experience required, 21
circuit_dump function, 310
circuit \(\bar{N}\) ame

\section*{Index}
annotation attribute, 383
circuits
compiled files
enabling and disabling on streams, 31
entry of events, 31
events entering multiple, 32
features of, 26
global definitions, 39
multiple circuits, 32
multiple streams, sharing, 30
parameters, 35
properties of, 34
representation of events within, 152-163
scope of correlation Iogic, 25
testing new, 31
as top-level Compound node, 35
transit delays accommodate delayed
events, 171
uncompiled files, 397
using with different Data and Fact stores, 31
verification, 389
cleanup phase
engine processing, 179
Clear Input port
Count node, 85
Rate node, 115
Clear Output port
Count node, 85
Rate node, 115
Clock node
AAAA, 64
allowable output to Sink node, 70
creates temporary events, 157
introduction to, 64
See also Sink node
source of event context, 184
CMIP event reports
create_time, 174
stamped with absolute creation times, 167
Combine node
AAAA, 71
allowable output to Sink node, 78
creates composite events, 159
introduces internal transit delays, 171
introduction to, 71
recursive connection forbidden, 57
See also Sink node
source of event context, 184
transit delays, 56
comments
in Data Store files, 188
in ECDL expressions, 325
in Fact Store files, 194
composite events
AAA, 149
AAAA, 159
creating, 159
definition of, 149
exist only within a circuit, 159
introduction to, 159
restructuring, 159, 161
Compound node introduction to, 34
and circuits, 27
attributes referable by other nodes, 40
as a circuit at the top-level, 35
circuit policy, 42
global definitions lost when stored in library, 39
node parameters, 41
properties of, 34
scope of global definitions, 39
concatenation operator,
string (+), 237
concept of correlation, 25
condition parameters
Combine node, 76

\section*{Index}

Filter node, 103
Table node, 133, 134
Unless node, 144
conf file suffix, 398
configuration file
name and location, 398
Configure nodeName dialog
node parameters, 46, 49
node ports, 46, 53
connections, within circuits, 26
constants
binding patterns, 322
contents attribute
Table node, 135
context
Data and Fact Stores for current, 185
introduction to, 183
Coordinated Universal Time definition of, 69
required in event timestamps, 167, 168
correlation engine
incrementing current time, 179
order of correlating events indeterminate, 177
phases in correlation step, 179
correlation, concept of, 25
count attribute
Count node, 84
Table node, 136
Count node
AAAA, 81
attribute referable by other nodes, 40
introduction to, 81
count_dict attribute
Couñt node, 85
countRemaining function, 350
Create node
AAAA, 87
creating primitive events, 30, 156
introduction to, 87
modifying created primitive events, 30
setting create_time attri butes, 156
source of event context, 184
create spec parameter
Create node, 90
create_time
even \(\bar{t}\) (header) attribute, 152, 170, 184
created_event (event name), create spec parāmeter, 88
CREATION_TIME
ITO messāge attribute, 404
current context
See current context
current time
increments in correlation engine, 179
resolution of, 169
current_event (event name)
delete condition parameter, 134
retain condition parameter, 134

\section*{D}
data keyword
dynamic data types, 231
Data Store files
comments in, 188
creating, 186
data types of entries, 188
ECDL expressions forbidden in, 190, 320
format of entries, 187
format of header line, 187
formats of files, 187
name and default location, 397
Data Stores
AAAA, 186
and circuits, 27
and statically evaluated node parameters, 52
dumping, 202

\section*{Index}
in current context, 185
introduction to, 186
limits to size or number of versions, 185
persistence of context, 185
tips for the use of, 200
when deleted, fate of events, 185
data type errors
in dynamically evaluated expressions, 52
data types
binding patterns, 322
bool ean (simple type), 215
dict (structured type), 225
duration (simple type), 168, 216
dynamic (special), 230
integer (simple type), 213
list (structured type), 223, 327
oid (simple type), 218
opaque (special), 229
real (simple type), 214
specifying return types of functions, 346
string (simple type), 219
time (simple type), 168, 217
token (simple type), 221
tuple (structured type), 222, 326
typecase expressions, 337
types permitted in Data Store files, 189
types permitted in Fact Store files, 195, 196
union (structured type), 231
user-defined (special), 231
void (special), 227
data_store attribute
associating context values with, 184
event (header) attribute, 153
dataStore function
AAAA, 262
accessing data store entries, 189
description of, 262
example, 190
references Data Store, 186

Decrement Input port
Count node, 85
Delay node
AAAA, 93
and transit delays, 56, 171
introduction to, 93
recursive connection forbidden, 57
delay until parameter
Delay node, 95
delete condition parameter
Table node, 134
del imiter
dict "()", 225
expression sequences "()", 339
list data type ([]), 327
tuple data type "()", 326
designer
skills and experience required, 21
dict data type
introduction to, 225
key data type, 332
keys access values, 332
manipulating, 332
specifying types, 333
dict keyword
dynamic data type, 230
div function (integer), 263
DivideByZero exceptions, 342
division operator,
arithmetic (/), 241
dot notation, 55
ds file suffix
description and default location, 397
required by ECS Designer, 186
duration data type
AAAA, 216
applies to relative or elapsed time, 168
introduction to, 216
dynamic data types

\section*{Index}

AAAA, 230
groupings of simple and structured types, 231
introduction to, 230
list data type and, 327
dynamically evaluated parameters
Create node, 90
evaluation of, 52
Filter node, 103
M odify node, 107
Table node, 133, 134
when auditing, 392

\section*{E}

ECDL
introduction to, 25
ECDL expressions, 325
binding patterns, 321
comments in, 206, 325
comparison with C language, 370
exception expressions, 343
exists expressions, 328
find expressions, 329
flow-control expressions, 334
forall expressions, 329
forbidden in Data and Fact Store files, 197
forbidden in Data and Fact store files, 190, 320
forcing precedence, common errors, 345
fun expressions, 346
handle expressions, 342
iteration, 347
let expression, 324
named values, 321
raise expressions, 343
recursion, 347
sequence expressions, 339
typecase expressions, 337
user-defined functions, 345
eco file suffix, 397
ECS circuit, see circuits
ECS Designer
introduction to, 25
engine policy for simulation, 33
environment variables for, 399
Fact Store not updatable while engine is running, 193
files loaded or generated by, 397
identifying ports, 54
ECS Designer (Build) window defining transit delay window for circuit input ports, 171
ECS Designer (Simulate) window
displays current time, 169
statistics available in, 56
ecs file suffix, 397
ecs_defaults.ecs file, 397
ecsīn.evt* files, 398
ecsout.evt* files, 398
encoding type parameter, Create node, 89
encoding_type
event (header) attribute, 153
end-of-line anchor, pattern matching (\$), 288
engine log files
auditing information, 389
file names for DM , 398
flexible format, 381
engine policy, effect on unaccepted events, 35
engine trace files, 398
environment variables, subset required by ECS Designer, 399
equality operator,
relational (=), 243
Error Output port
AAAA, 54
AAAB, 54
Annotate node, 62
Combine node, 78

\section*{Index}

\section*{Count node, 85}

Create node, 91
events logged to engine log if not connected, 54
Extract node, 101
Filter node, 104
general characteristics of, 54
M odify node, 107
Rate node, 115
Rearrange node, 120
Table node, 135
Unless node, 144
errors
data type, in dynamically evaluated expressions, 52
EVComposite
event_class attribute value, 159
event àttributes
arrival time set in event header, 170
creation time set in event header, 170
Event data type, 332
temporary attributes prepended in header, 152
event contexts
simulating in ECS Designer, 201
using to access Fact Store, 199
Event data type
AAAA, 226
event attributes, 332
introduction to, 226
subtype of dict data type, 332
event I/O API, 170
event log files
annotation requests and responses, 382
captured by ECS Engine, 173
example entry, 380
format generated by the ECS Engine, 381
interpolating delays between events, 169
line types, 379
name and location, 397
requirements for event format, 385
special characters, 385
syntax, 379
event names
AAAA, 50
created_event, 88
current_event, 134
general characteristics of, 50
inhibitor_event, 144
input_evēnt, 62, 88, 104, 107, 144
outpūt_event, 77
retained_event, 134
event specification parameters
Create node, 89, 90
M odify node, 107
Rearrange node, 118
event syntax parameter
Create node, 89
event types
basis for filtering at circuit inputs, 36
event_dass
event (header) attribute, 152
event_syntax
event (header) attribute, 153
event_type
event (header) attribute, 153
events
AAAA, 149
accommodating events with long transit delays, 180
and engine policy, 35
arrival_time attribute, 170
create_time attribute, 170
creating, 29
creating composite, 159
creating primitive, 156
creating, temporary, 157
definition of, 149

\section*{Index}
entering multiple circuits, 32
entry into circuits, 31
event log files, requirements for, 385
filtered on event types at circuit inputs, 36
filtered on transit delays at circuit inputs, 35
header attributes, 152
internal handling of, 29
manipulating, 150
modifying attributes of, 29, 155
network delays and time stamping, 166
not accepted by any circuit, 35
order of correlation indeterminate, 177
primitive, handling of, 29
restrictions to output, 27
structure of, 149, 152
temporary header prepended within a circuit, 154
temporary, structure of, 157
types transmitted from output ports, 54
unique id, 150
evt file suffix, 397, 398
EVTemporary
event_class attribute value, 157
exception expressions (ECDL), 343
exceptions
(wildcard) handles any, 342
BindingMismatch, 342
DivideByZero, 342
FloatingOverflow, 342
handling, 341
InternalE rror, 343
introduction to, 341
InvalidArgs, 343
NotFound, 343
NotFound exception, 341
raising, 341, 343
RangeE rror, 342
standard, 342

TypeM ismatch, 342
user-defined, 343
exists expressions (ECDL), 328
expiryTime, annotation attribute, 383
explode function (list string), 264
explode_time function (dict), 265
exponentiation operator,
arithmetic ( \(\wedge\) ), 242
expression delimiter
pattern matching (), 288
expression delimiter (pattern matching), 288
expression nesting delimiter
pattern matching ([]), 289
Expressions, 49
external filtering, 36
Extract node
AAAA, 97
creates composite events, 159
creating primitive events, 29
introduction to, 97
modifying created primitive events, 29
source of event context, 184
extract spec parameter
Extract N ode, 100

\section*{F}

Fact Store files
comments in, 194
creating, 193
data types of entries, 195
ECDL expressions forbidden in, 197, 320
format of entries, 194, 195
format of header line, 194
formats of files, 194
name and default location, 397
Fact Stores
AAAA, 193
and circuits, 27

\section*{Index}
and statically evaluated node parameters, 52
dumping, 202
in current context, 185
introduction to, 193
limits to size or number of versions, 185
persistence of context, 185
tips for the use of, 200
when deleted, fate of events, 185
fact_exists function
AĀAA, 267
example, 198
introduction to, 267
testing presence of relationships, 197
fact_find_left function
AĀAA, \(\overline{2} 68\)
example, 198
extracts left-hand values of relationships, 197
introduction to, 268
fact_find_right function
AAAAA, 269
example, 198
extracts right-hand values of relationships, 197
introduction to, 269
fact_store
event (header) attribute, 153
fact_store attribute
associating context values with, 184
Fail Output port
AAAA, 54
Annotate node, 62
Create node, 96
Extract node, 101
general characteristics of, 54
Unless node, 144
False Output port
Filter node, 104
files
*.ds data store files, 186
*.fs Fact Store files, 193
Fact Store file naming, 193
loaded and generated by ECS Designer, 397
filesData Store file naming, 186
Filter node
introduction to, 102
find expressions (ECDL), 329
fixity declarations in ECDL, 351
FloatingOverflow exceptions, 342
flow-control expressions (ECDL)
AAAA, 334
and exceptions, 341
introduction to, 334
foldl function
AAAA, 274
example, 348
introduction to, 274
foldr function
AAAA, 275
example, 348
introduction to, 275
forall expressions (ECDL), 329
fs file suffix
description and default location, 397
required by ECS Designer, 193
fun expressions (ECDL), 346
functions
AAAf, 228
AAAfixity, 345
alter (dict), 252
and (boolean), 253, 254
arguments white-space separated, 344
audit_log, 390
bitand (integer), 255
bitinv (integer), 256
bitleft (integer), 257
bitor (integer), 258

\section*{Index}
bitright (integer), 259
bitxor (integer), 260
calling syntax, 344
chr (integer), 261
countRemaining, 350
datastore (various), 189, 262
div (integer), 263
explode (list string), 264
explode_time (dict), 265
fact_exis̄ts (bool ean), 197, 267
fact_find_left (any list), 197, 268
fact_find_right (any list), 197, 269
fixity of, 345
foldl (various), 274
foldr (various), 275
function types, 228
global, definitions of, 38
implies (boolean), 276
implode_time (time), 278
infix, \(3 \overline{45}\)
inter (list), 280
interc (list), 281
I p.fromOctet (string), 282
Ip.toOctet (string), 283
iteration, 347
join (string), 284
length (integer), 285
match.make (opaque), 286
match.test (bool ean), 293
match.testVars (tuple boolean * dict), 294
mod (integer), 296
not (boolean), 296
nth (various), 297
oid.append (boolean), 298
oid.join (boolean), 299
oid.last (boolean), 300
oid.split (boolean), 301
oid.split_at (boolean), 302
oid.split_nnm (boolean), 303
or (boolean), 304
ord (integer), 304
prefix, 345
recursion, 347
reverse (list), 305, 306
round (integer), 306
specifying return data types, 346
split (list string), 307
stringOf (string), 308
System.audit_log, 309, 390
System.circuit_dump, 310
System.set_trāce_masks, 312
System.trače, \(31 \overline{1}\)
Time.now, 314
truncate (integer), 315
union (list), 316
unionc (list), 317

\section*{G}
global definitions
introduction to, 37
and circuits, 27
binding values to identifiers, 37
circuits vs. Compound nodes, 35
not saved when Compound node stored in
library, 39
scope of in Compound node, 39
stored with a circuit, 39
Global Definitions dialog
accessing, 37
defining exceptions, 343
defining Fact Store relationships, 196
defining union data types, 231
defining user-defined data types, 231
global functions, 38
global types, 38
global values, 37
greater than operator, relational (>), 244

\section*{Index}
greater than or equal to operator, relational (>=), 244
GROUP
ITO message attribute, 404

\section*{H}
handles for expressions (ECDL)
attaching to expressions, 342
specifying scope, 191
handling exceptions, 341

\section*{I}
identifiers
binding global values to, 37
binding patterns, 322
construction of, 205
if expressions (ECDL), 334
implies function (boolean), 276
implode_time function (time), 278
Increment Input port
Count node, 85
I ncrement Output port
Count node, 85
infix and prefix (ECDL), 352
infix functions, 345
Inhibited Output port
Unless node, 144
Inhibitor Input port
Unless node, 144
inhibitor_event (event name), condition
parameter, Unless node, 144
initial count parameter
Count node, 84
input phase
engine processing, 179
overload processing, 180
input policy, 27
Input port

AAAA, 53
Annotate node, 62
Combine node (multiple inputs), 78
Create node, 91
Delay node, 96
Extract node, 101
Filter node, 104
general characteristics of, 53
M odify node, 107
Rate node, 115
Rearrange node, 120
relationship with circuit, 26
Sink node, 124
Table node, 135
Unless node, 144
input ports
AAAA, 53
circuit vs Compound node, 35
configured in Configure nodeName dialog, 53
external filtering, 36
general characteristics of, 53
maximum number of, 36
required and optional ports, 53
input_event (event name)
annōtate spec parameter, 62
condition parameter
Filter node, 104
Unless node, 144
create spec parameter, 88
modify spec parameter, 107
INSTR_IF
ITO message atttribute, 404
INSTR_IF_TYPE
ITO message attribute, 404
INSTR_PAR
ITO message atttribute, 404
integer data type, 213
inter function (list), 280

\section*{Index}
interc function (list), 281
InternalError exceptions, 343
interval parameter
Clock node, 66
Combine node, 75
Rate node, 111
InvalidArgs exceptions, 343
Ip.fromOctet function (string), 282
Ip.toOctet function (string), 283
ITO
create_time, 174
encoding_type, 153
event logs, 409
event_syntax, 153
message attributes, 403
troubleshooting, 412
ITO event attributes
addressing, 408

\section*{J}
join function (string), 284

\section*{K}
key spec parameter
Count node, 84
Rate node, 113

\section*{L}
length function (integer), 285
less than operator, relational (, 244
less than or equal to operator, relational (, 244
let expressions (ECDL), 324
libraries
global definitions lost when Compound node stored, 39
life, the universe, and everything, meaning of, 238, 327
list builder expressions (ECDL)
examples, 347
syntax, 332
list constructor operator (::), 246
list data type
AAAA, 223
binding patterns, 322
constructor (::), 328
deriving sub-lists, 330
dynamic data types and, 327
empty list, 327
exists expressions, 328
find expressions, 329
forall expressions, 329
introduction to, 223
manipulating, 327
select expressions, 330
specifying type, 328
list key word dynamic data type, 230
log file suffix, 398

\section*{M}
map dict key to value ( \(\Rightarrow\) ), 225
mask operator,
pattern matching ( \(\backslash\) ), 290
mask-character operator,
pattern matching ( \(\backslash\) ), 290
match.make (opaque), 286
match.test (bool ean), 293
match.testVars (tuple bool ean * dict), 294
max events parameter
Extract node, 100
Table node, 133
md file suffix, 398
mdl file suffix, 398
MDL message description file
name and location, 398
MDL metadata file

\section*{Index}
name and location, 398
message description file
MDL, name and location, 398
metadata file
BER, name and location, 398
MDL, name and location, 398
mod function (integer), 296
Modify node
AAAA, 105
creating primitive events, 29
introduction to, 105
modifying components of composite events, 160
modifying created primitive events, 29
source of event context, 184
modify spec parameter
M odify node, 107
module
AAAA, 355
files, 356
introduction to, 355
names, 355
names, scope of, 358
moving parameter
Rate node, 112
MSG_LOG_ONLY
ITO- messäge attribute, 405
MSGID
ITO message attribute, 404
MSGSRC
ITO message attribute, 404
MSGSRC_TYPE
ITO message attribute, 405
MSGTEXT
ITO message attribute, 405
MSGTYPE
ITO message attribute, 405
MSI_OUTPUT
ITŌ message attribute, 406
multiple circuits
and events, 32
and stream policy, 32
multiplication operator, arithmetic (*), 240

\section*{N}
n\#number of di gits (pattern matching), 288
\(\mathrm{n}^{*}\) number of characters (pattern matching), 288
named values
AAAA, 321
and binding patterns, 321
binding patterns, 321
introduction to, 321
let expressions in, 324
specifying data type, 321
negative sign operator (-), 238
network delays
time stamping of events, 166
no separators match operator,
pattern matching (), 288
node
changing names attribute references, 47
col ors indi cate status of, 47
common features, 46
configuring, 46
default names of, 46
displaying names of, 47
order of correlating events indeterminate, 177
nodes
and circuits, 26
node attributes
AAAA, 55
changing node names breaks references to, 47
in Compound nodes, 40
displayed in Node Status dialog, 46, 55

\section*{Index}
general characteristics of, 55
referencing in node parameter expressions, 55
value, retrieving from composite events, 51
value, retrieving from primitive events, 51
node parameters
AAAA, 49
and circuits, 26
and Data and Fact Store updates, 52
in Compound nodes, 41
configured in Configure nodeName dialog, 49
dynamic evaluation of, 52
expect values with specific data type, 52
general characteristics of, 49
simple expressions in, 49
static evaluation of, 52
Node Status dialog
displays node statistics, 56
NODENAME
ITO message attribute, 406
nodeN ame,
annotation attribute, 383
nodes
can introduce transit delays, 56
changing names breaks, attribute references, 47
col ors indicate status of, 47
common features, 46
configuring, 46
default names of, 46
displaying names of, 47
order of correlating events indeterminate, 177
nonassoc ECDL keyword, 352
not function (boolean), 296
not operator,
pattern matching (!), 289
NotFound exception, 341, 343

NOTIFICATION
ITO message attribute, 406
now function, 314
nS number of separators (pattern matching), 288
nth function (various types), 297
num_attrs
event (header) attribute, 153
number of characters expression, pattern matching ( n ), 288
number of digits expression, pattern matching ( \(\mathrm{n} \#\) ), 288
number of separators expression, pattern matching (nS), 288

\section*{0}

OBJ ECT
ITO message attribute, 406
oid data type, 218
oid.append function, 298, 299
oid.last function, 300
oid.split function, 301
oid.split_at function, 302
oid.split_nnm function, 303
one or more digits expression, pattern matching (\#), 288
one or more non-separators expression, pattern matching ( \({ }^{\wedge}\) ), 288
one or more separators expression, pattern matching (S), 288
OPACTION_ACK
ITO messäge attribute, 406
OPACTION_ANNOTATE
ITO message attribute, 406
OPACTION_CALL
ITO messagge attribute, 406
OPACTION_NODE
ITO message attribute, 406
OPACTION_STATUS

\section*{Index}

ITO message attributes, 407
opaque data type, 229
operators, 235-246
, 244, 244
- (subtraction, sign), 238
() (dict delimiter), 225
() (expression sequence delimiter), 339
() (tuple delimiter), 326
* (multiplication), 240
* (tuple type separator), 326
+ (plus, concatenation, sign), 237
, (dict key-value pair separator), 225
, (list element separator), 327
, (tuple member separator), 326
/ (division), 241
:: (prepend to list), 246
; (expression separator), 339
\(=\) (equality), 243
\(\Rightarrow\) (map dict key to value), 225
\(>\) (greater than), 244
\(>=\) (greater than or equal), 244
[] (list delimiter), 327
ヘ (exponentiation), 242
| (alternative), 231
(typecase alternative), 337
AAAfixity, 345
fixity of, 345
or function (boolean), 304
or operator, pattern matching (| ), 289
ord function (string), 304
order parameter
Combine node, 76
ORIGMSGTEXT
ITO message attribute, 407
output event log files, 398
output policy, 27
Output port
AAAA, 54
Annotate node, 62

Clock node, 70
Combine node, 78
Create node, 91, 96
Extract node, 101
general characteristics of, 54
M odify node, 107
Rate node, 115
Rearrange node, 120
relationship with circuit, 26
Source node, 127
Table node, 135
Unless node, 144
output ports
AAAA, 53
configured in Configure nodeName dialog, 53
events discarded if not connected, 54
general characteristics of, 53
required and optional ports, 53
types of events transmitted from, 54
output_event (event name)
condition parameter
Combine node, 77
overload processing, 180

\section*{P}
pattern assignment operator, pattern matching (.), 290
pattern matching, 287
assigning subpatterns to variables, 290
examples, 292
operators and expressions, 288
subpatterns, 291
pattern matching expressions, 288, 293
(expression delimiter), 288
! (not), 289
\# (one or more digits match), 288
\$ (end-of-line anchor), 288
* (zero or more characters match), 288

\section*{Index}
. (pattern assignment), 290
@(one or more non-separators match), 288
It (tab), 290
^(start-of-line anchor), 288
| (alternative), 289
assigning patterns to variables rules for, 290
examples, 292
n\#(number of digits match), 288
n* (number of characters match), 288
nS (number of separators match), 288
opaque data type and pattern matching expressions
AAAo, 229
rules for, 290
S (one or more separators match), 288
pmd. Iog* files, 398
pmd.trc* files, 398
ports
identifying in ECS Designer, 54
postmaster log
when required for auditing, 389
precedence in ECDL, 351
prefix and infix (ECDL), 352
prefix functions, 345
prepend operator, list (::), 246
primitive events
AAAA, 149
attribute key data type, 155
creating through Create node, 156
definition of, 149
modifying attributes of, 155
structure of, 152, 154
properties of circuits and compound nodes, 34

\section*{R}
raising exceptions (ECDL), 343
RangeE rror exceptions, 342
rate attribute

Rate node, 114
Rate node
AAAA, 109
attribute referable by other nodes, 40
introduction to, 109
rate_dict attribute
Rate node, 114
real data type, 214
Rearrange node
AAAA, 116
allowable output to Sink node, 121
introduction to, 116
restructures composite events, 159
See also Sink node
source of event context, 184
rearrange spec parameter, Rearrange node, 118
RECEIVE_TIME
ITO message attribute, 407
recursive connections, 57
relational operators, 244
requestld
annotation attribute, 383
reserved words and symbols, 204
Reset Input port
AAAA, 53
Annotate node, 62
Combine node, 78
Count node, 85
Create node, 96
general characteristics of, 53
Rate node, 115
Table node, 135
Unless node, 144
Reset Output port
AAAA, 54
Annotate node, 62
Combine node, 78
Count node, 85

\section*{Index}
general characteristics of, 54
Rate node, 115
Table node, 135
responseList
annotation attribute, 383
restrictions to event output, 27
retain condition parameter, Table node, 133
retained_event (event name)
delete condition parameter, 134
reverse function (list), 305, 306
round function (integer), 306

\section*{S}

S one or moreseparators (pattern matching), 288
save until parameter
Table node, 133
scope of names, 358
select expressions, 330, 347
separator
dict key-value pair (,), 225
expressions (;), 339
list elements (,), 327
tuple data types (*), 326
tuple members (,), 326
sequence expressions (ECDL)
mechanism for separating ECDL expressions, 339
returns value of last expression, 339
sequencel d
annotation atttribute, 383
set trace mask function, 312
SEVERITYY
ITO message attribute, 407
simple keyword
dynamic data types, 230
simulating
time, 173
transit delays, 175

Sink node
introduction to, 122
output to external environment, 124
receiving output from Annotate node, 63
receiving output from Clock node, 70
receiving output from Combine node, 78
receiving output from Rearrange node, 121
receiving output from Unless node, 146
snapshot of engine state, 310
SNMP traps
create_time, 174
transit delay set to zero, 172
variable bindings example, 337
source code samples, 361
Source node
introduction to, 125
split function (list string), 307
Start Input port
Clock node, 70
Start Output port
Clock node, 70
start-of-line anchor
pattern matching ( \(\wedge\) ), 288
statically evaluated parameters
evaluation of, 52
when auditing, 392
statistics
displayed in Node Status dialog, 56
Std, module name, 356
Stop I nput port
Clock node, 70
Stop Output port
Clock node, 70
stream
default policy, 28
policy, 27
policy and circuit design, 30
stream policy and multiple circuits, 32

\section*{Index}
and transit delays, 32
streams
introduction to, 27
and applications, 27
enabling and disabling circuits, 31
entry of events, 31
sharing circuits, 30
string concatenation operator (+), 237
string data type, 219
stringOf function (string), 308
subtraction operator,
arithmetic (-), 238
synchronization of network important, 167
System.audit_log function, 309, 390
System.circuit_dump function, 310
System.set_trāce_masks function, 312
System.trace function, 311

\section*{T}
tab character
pattern matching ( \(\backslash \mathrm{t}\) ), 290
Table node
AAAA
, 128
and event contexts, 185
attributes referable by other nodes, 40
introduction to, 128
recursive connection forbidden, 57
tags, 290
temporary events
AAA, 149
introduction to, 157
creating, 157
definition of, 149
exist only within a circuit, 157
structure of, 157
testing a new circuit, 31
time
setting create_time attributes, 156
synchronization, 167
time zones, 174
time alignment parameter
Clock node, 66
Rate node, 112
time data type
AAAA, 217
applies to absolute times, 168
introduction to, 217
time limit parameter
Annotate node, 62
time window parameters
Clock node, 66
Combine node, 75
Delay node, 95
Rate node, 111
Table node, 133
Unless node, 143
Timenow function, 314
timing
importance of network synchronization, 167
token data type, 221
trace function, 311
transit delays
accommodating long delays, 180
Annotate node, 56
basis for filtering events at circuit inputs, 35
Combine node, 56
Delay node, 56
effect of stream policy, 32
external filtering, 36
how caused, 171
introduced by some nodes, 56, 171
set to zero for SNMP traps, 172
setting in event log, 175
Unless node, 56
trc file suffix, 398
TROUBLETICKET

\section*{Index}

ITO message attribute, 407
TROUBLETICKET_ACK
ITO message attribute, 407
True Output port
Filter node, 104
truncate function (integer), 315
tuple data types
AAAA, 222
binding pattern for, 322
Data Store files, 188
Fact Store files and function arguments, 197
introduction to, 222
manipulating, 326
specifying, 326
typecase expressions not permitted, 338
type
global, definition of, 38
type keyword
dynamic data types, 231
user-defined data types and, 231
typecase expressions (ECDL)
introduction to, 337
valid data type tests in, 338
TypeM ismatch exceptions, 342

\section*{U}
union data types
AAAA, 231
defined in the Global Definitions dialog, 231
introduction to, 231
union function (list), 316
unionc function (list), 317
unique ID
of an event, 150
used for auditing purposes, 391
unique_id
event (header) attribute, 152
universal pathnames
subset required by ECS Designer, 399
Unless node
AAAA, 139
allowable output to Sink node, 146
creates composite events on error, 159
introduces internal transit delays, 171
introduction to, 139
recursive connection forbidden, 57
See also Sink node
transit delays, 56
UNMATCHED
ITO message attribute, 408
user-defined data types
AAAA, 231
defined in the Global Definitions dialog, 231
introduction to, 231
UTC, See Coordinated Universal Time

\section*{V}
value binding operator (=), 37
values
global, definitions of, 37
verification of circuits, 389
void data type, 227
void value "() ", 227

\section*{w}
wildcard binding pattern (_), 322
window parameter
Unless node, 143

\section*{Z}
zero or more characters expression, pattern matching (*), 288```

