Rule Developer’s Guide

Version 39
IBM Tivoli Enterprise Console

Rule Developer’s Guide

Version 39
Contents

About this guide ............... ix
Who should read this guide .......... ix
What this guide contains .......... ix
Publications .................. x
IBM Tivoli Enterprise Console library ........ x
Related publications .......... xi
Accessing publications online .......... xi
Ordering publications .......... xii
Contacting software support .......... xii
Participating in newsgroups .......... xiii
Conventions used in this guide ......... xiii
Typeface conventions .......... xiii
Operating system-dependent variables and
paths .......... xiv
IBM Tivoli Enterprise Console icons .......... xiv

Part 1. Correlating events at the
event server .............. 1

Chapter 1. Rule development
fundamentals .............. 3
Designing rules .................. 3
Events ....................... 4
  Event adapters ................. 4
  Event attributes supplied by an event adapter  5
  Events sent from other products or the command
  line .......... 6
Event filtering .................. 6
Special characters in events .......... 7
Event attribute values supplied by the event server . 7
Event cache ...................... 9
Rules ................. 9
  Rule types ............... 10
  General rule structure .......... 11
  What causes a rule to run .......... 12
  Example actions you can take with rules .......... 12
Rule bases .......... 12
  Rule base targets .......... 14
  Rule sets and rule packs .......... 15
  Rule pack example .......... 16
  Modifying rule sets .......... 17
Event classes ................... 18
Predicates ............... 18
Rule base example .......... 19
  Requirements for rule base example .......... 19
  Assumptions for rule base example .......... 19
  Procedure for rule base example .......... 21
Rule base manipulation procedures using the
rule builder .......... 22
  Creating a new rule base .......... 23
  Editing an existing rule base .......... 24
  Listing all known rule bases .......... 24
  Listing the loaded rule base .......... 25
Listing rule sets and BAROC files in a rule
base .......... 25
Copying a rule base .......... 26
Deleting a rule base .......... 27
Importing a rule set into a rule base .......... 28
Importing BAROC files into a rule base .......... 29
Compiling a rule base .......... 30
Loading and activating a rule base .......... 30
Loaded and active rule bases .......... 31
UNIX directory permissions for rule bases .......... 32
Testing, debugging, and analyzing rules .......... 32
Creating rules in international environments .......... 33
Using non-English text in rules .......... 33
Passing data to tasks and programs .......... 33
  Passing data from exec_program and
  exec_program_local predicates .......... 33
  Data for exec_task and exec_task_local
  predicates .......... 33
  UTF-8 data and local-encoded data
  compatibility .......... 34

Chapter 2. Event class concepts .......... 35
Designing event classes ................. 35
Event class definitions .......... 36
Event attribute definitions .......... 37
  Attribute names .......... 38
  Attribute data types .......... 38
  Attribute facets .......... 39
Inheritance .......... 40
  Inheritance example .......... 42
Enumeration definitions .......... 43
Design recommendation for using enumerations
instead of event class definitions .......... 43
Overview of the root.baroc and tec.baroc files .......... 43
Re-importing modified BAROC files .......... 44
BAROC files in international environments .......... 44

Chapter 3. Rule engine concepts .......... 45
Forced cleaning of the event cache .......... 45
Processing events .......... 45
  Processing timer expiration .......... 47
  Processing the next transaction .......... 47
Performance tips .......... 47

Chapter 4. Rule language reference .......... 49
Naming conventions .......... 49
Variables .......... 50
  Regular variables .......... 51
  Global variables .......... 51
  Manipulating global variables .......... 52
Constants .......... 52
Comments .......... 52
Rule structure .......... 53
  Rule parser grammar .......... 53
  General structure of a plain rule .......... 53

© Copyright IBM Corp. 2003
Chapter 5. Correlation examples .......................... 217
Recording causal relationships between events ........................................... 217
Special cases when recording causal relationships ..................................... 218
Multiple events of the same class but different meanings ......................... 218
An event is not part of a sequence .............................................................. 219
Using causal relationship information ....................................................... 219
Cause events .............................................................................................. 219
Effect events .............................................................................................. 220
Clearing events ......................................................................................... 220
Processing events based on causal relationships ........................................ 221
Processing cause events ........................................................................... 221
Processing effect events ........................................................................... 221
Processing clearing events ....................................................................... 222
Event correlation rule language predicates .................................................. 222
Defining event relationship information ...................................................... 223
How event criteria is evaluated .................................................................. 223
Using event relationship information to find related events ..................... 224
Putting it all together ................................................................................. 225
Creating the event relationship information .............................................. 225
Rules for processing the events .................................................................. 226
process_problem_events rule .................................................................. 226
process_clearing_event rule .................................................................... 228
Alternatives ............................................................................................... 228

Chapter 6. Testing, tracing, and profiling rules ............................................ 231
Testing rules ............................................................................................... 231
Sending events with commands .................................................................. 231
Sending events with the tec_agent_demo program ..................................... 231
Configuration prerequisites for using the tec_agent_demo program ........... 231
Using the tec_agent_demo program ........................................................... 232
Sending events with the SendEvents program ............................................ 232
Creating event files from events in the reception log .................................. 233
Tracing rules ............................................................................................... 234
Trace granularity ....................................................................................... 235
Tracing a rule set ....................................................................................... 235
Tracing a rule ............................................................................................. 235
Trace information ...................................................................................... 236
Rule tracing example ................................................................................ 236
Tracing a rule event filter .......................................................................... 236
Tracing rule actions .................................................................................... 237
Tracing multiple solution event filters in predicates .................................. 238
Profiling rules ............................................................................................. 239
Profile granularity ...................................................................................... 240
Profiling a rule set ..................................................................................... 240
Profiling a rule ........................................................................................... 240
Measuring event processing performance ................................................ 240

Chapter 7. Creating rules with the graphical rule builder ......................... 243
Overview of the rule builder ....................................................................... 243
Graphical rule builder considerations for international environments ....... 243
Using the rule builder ................................................................................. 244
Starting the rule builder ............................................................................. 244
Editing an existing rule base ...................................................................... 247
Creating a new rule set .............................................................................. 247
Editing a rule set ....................................................................................... 248
Creating a simple rule .............................................................................. 249
Creating a compound rule ........................................................................ 258
Saving a rule set ....................................................................................... 262
Compiling a rule base ................................................................................ 263
Stopping and restarting the event server .................................................... 266
Rule builder examples ............................................................................... 267
Defining an escalation policy ...................................................................... 267
Problem ..................................................................................................... 267
Solution ..................................................................................................... 267
Get event information ............................................................................... 267
Build the rules .......................................................................................... 268
Monitoring for potential security breaches on UNIX endpoints .................. 271
Problem ..................................................................................................... 271
Solution ..................................................................................................... 271
Modify default severity ............................................................................ 271
Create the rules ......................................................................................... 271
Using Distributed Monitoring as a passive monitor ................................... 274
Problem ..................................................................................................... 274
Solution ..................................................................................................... 274
Configure the distributed monitoring monitors .......................................... 274
Create the rules ........................................................................................ 275
Linking related events ................................................................................ 278
Part 2. Correlating events at the gateway or adapter ........................................... 281

Chapter 8. Writing rules for the state correlation engine ........................................... 283

State correlation rules ................................................................. 283
Predicates .................................................................................. 284
Actions ....................................................................................... 284
Discard ....................................................................................... 285
TECSummary ............................................................................. 285
Forward ...................................................................................... 286
SendTECEvent .......................................................................... 287
Attributes common to all rules .................................................... 288
Matching rules ......................................................................... 288
Duplicates rules ....................................................................... 289
Threshold rules ....................................................................... 289
Collector rules ......................................................................... 292
Passthrough rules .................................................................... 292
Reset on match rules ................................................................ 294
Cloning state machines ............................................................. 296

Chapter 9. State correlation reference ................................................................. 305

Rule syntax .............................................................................. 305
Basics of a rule ......................................................................... 305
Match rule example .................................................................. 306
Collector rule example .............................................................. 307
Duplicate rule example .............................................................. 308
Threshold rule example .............................................................. 309
Passthrough rule example .......................................................... 309
Reset on match rule example ...................................................... 310
Declaration of variables .............................................................. 311
Predicate library ....................................................................... 312
Value types .............................................................................. 312
Attributes ................................................................................ 313
Predicates and functions for state correlation .............................. 313

Integer predicates .................................................................. 314
  == .................................................................................... 314
  != .................................................................................... 314
  > ................................................................................... 314
  < ................................................................................... 315
  >= .................................................................................. 315
  <= .................................................................................. 315

Integer functions .................................................................... 316
  + ................................................................................... 316
  * .................................................................................. 316
  / .................................................................................... 317

IP address predicates ................................................................. 334
  == .................................................................................... 334
  != .................................................................................... 335

String set predicates ................................................................. 335
  == .................................................................................... 335
  != .................................................................................... 335
  areDisjoint ....................................................................... 336
  isMemberOf ...................................................................... 336
  isSubsetOf ...................................................................... 336
String set functions ........................................ 337
  intersect ........................................ 337
  join ........................................ 337
  previousValues .................................. 337
Boolean predicates ...................................... 338
  and ........................................ 338
  not ........................................ 338
  or ........................................ 339
Integer set predicate .................................. 339
  isMemberOf .................................. 339
IP address set predicate ............................... 339
  isMemberOf .................................. 339
State correlation messages ............................. 340

Part 3. Appendixes ........................................ 343

Appendix A. Using Prolog in rules .................... 345
Prolog predicate online information .................. 345
Language basics ........................................ 345
  Terms ........................................ 345
  Facts ...................................... 346
  Rules ...................................... 346
  Knowledge base ................................. 347
  Comments ................................... 348
Data Types and Ranges .................................. 348
  Data type mapping between BAROC and Prolog .... 349
Example of using Prolog in a Tivoli Enterprise
  Console rule .................................... 349
Quick reference of Prolog predicates and operators 350
Instantiation and unification .......................... 350
Test data type and variable instantiation ........... 351
Convert data types .................................... 351
Atom manipulation ..................................... 352
  Standard order comparison ........................ 352
  Functions .................................... 352
  Miscellaneous .................................. 352
List manipulation ..................................... 353
Mathematical expressions ............................... 353
  Expression evaluation ........................... 353
  Arithmetic comparison ........................... 354
  Mathematical operators .......................... 354
  Pointer arithmetic ................................ 354
  Miscellaneous .................................. 355
Knowledge base ........................................ 355
I/O ............................................... 355
Alphabetic listing of built-in predicates and
  operators ...................................... 355
  + ........................................... 356
  - (pointer offset subtraction) .................... 357
  - (pointer subtraction) .......................... 359
  - (sign reversal) ................................ 360
  - (subtraction) .................................. 361
  * ........................................... 362
  / ........................................... 363
  // ........................................ 364
  @< ......................................... 364
  @=< ...................................... 365
  @> ......................................... 366
  @>= ...................................... 367
  \= ....................................... 368
  + (pointer offset addition) ....................... 369
  < ........................................ 370
  = ........................................ 371
  ^= ......................................... 372
  ^= ......................................... 373
  ^= ......................................... 374
  =\= ....................................... 375
  <= ........................................ 376
  >= ........................................ 377
  > ......................................... 378
  >= ......................................... 379
  abolish ....................................... 380
  append ....................................... 381
  ascii ....................................... 382
  assert ....................................... 383
  atom ........................................ 385
  atomic ....................................... 386
  atomlength .................................... 387
  atompredicate ................................ 388
  atomconcat ................................... 389
  atomtolist ................................... 390
  compile ...................................... 391
  consult ...................................... 393
  delete ....................................... 394
  disjoint ...................................... 395
  empty_list .................................... 396
  fclose ...................................... 397
  flisting ..................................... 398
  fopen ....................................... 399
  ground ........................................ 400
  int_to_hex ................................... 401
  integer ...................................... 402
  intersect ..................................... 403
  inttolist ................................... 404
  is ........................................... 405
  is_list ...................................... 406
  length ........................................ 407
  lowertoupper .................................. 408
  member ....................................... 409
  name ........................................ 410
  nmember ...................................... 411
  nmembers ..................................... 412
  not .......................................... 413
  number ....................................... 414
  pointer ....................................... 415
  pointertoatom ................................ 416
  pointertoatom ................................ 417
  pointertoint .................................. 418
  printf ....................................... 419
  read ......................................... 421
  readln ....................................... 422
  real (convert integer) ........................... 423
  real (test real type) ............................ 424
  reconsult .................................... 425
  remove_dups ................................... 426
  retract ....................................... 428
  round ........................................ 429
  rremove ..................................... 430
  sort ......................................... 431
  sprintf ...................................... 432
Making your predicates and facts available to the Tivoli Enterprise Console rule engine 442
Loading predicates and facts from a file 442
Using the assert predicate with a TEC_Start event 442
Using the Prolog interpreter 443

Appendix B. BNF grammar for BAROC files 445
Non-terminals 445
Terminals 445

Notices 447
Trademarks 449

Index 451
About this guide

The IBM® Tivoli Enterprise Console® product is a rule-based, event management application that integrates system, network, database, and application management to help ensure the optimal availability of an organization’s IT services. The IBM Tivoli Enterprise Console Rule Developer’s Guide provides information about developing rules for managing events with the Tivoli Enterprise Console product, using the Prolog engine on the event server or the XML-based state correlation engine at the gateway or adapter.

The emphasis of this guide is on developing rules with a text editor, although there is information for developing rules with the graphical rule builder of the Tivoli Enterprise Console product. The graphical rule builder is provided to help beginners develop and deploy simple rules, while more sophisticated rule development requires writing rules in the Tivoli Enterprise Console rule language.

The Tivoli Enterprise Console product also provides a collection of ready-to-use rule sets that you can deploy with minimal configuration. See the IBM Tivoli Enterprise Console Rule Set Reference for more information.

Who should read this guide

This guide is for programmers who need to develop rules for automating the management of Tivoli Enterprise Console events by the Tivoli Enterprise Console adapter, gateway, or event server. Readers of this book should have knowledge of the following topics:

- UNIX® and Windows® operating systems
- Shell programming
- Tivoli Enterprise Console functionality
- Extensible Markup Language (XML)

What this guide contains

This guide contains the following sections:

Part 1—Correlating events at the event server

Part 1 explains how to develop rules using the Prolog-based rule engine that runs at the Tivoli Enterprise Console event server. The Prolog engine can perform sophisticated correlation and establish causal links between events from multiple gateways, as well as historical events stored in the event cache. Rules for the Prolog engine are written in a rule language based on Prolog.

Part 2—Correlating events at the gateway or adapter

Part 2 explains how to develop rules using the XML-based state correlation engine that runs at either the Tivoli Enterprise Console gateway or the adapter. The state correlation engine can perform high-speed filtering and correlation of events closer to the source, which can significantly reduce event traffic at the event server. Rules for the state correlation engine are written in XML, with rule actions implemented as Java classes.
Part 3—Appendixes

Part 3 contains appendixes for advanced rule programmers. These appendixes include information about using the Prolog language in rules, as well as the formal grammar of the BAROC files used to define event classes.

Publications

This section lists publications in the IBM Tivoli Enterprise Console library and related documents. It also describes how to access Tivoli publications online and how to order Tivoli publications.

IBM Tivoli Enterprise Console library

The following documents are available in the IBM Tivoli Enterprise Console library:

- **IBM Tivoli Enterprise Console Adapters Guide, SC32-1242**
  Provides information about supported adapters, including how to install and configure these adapters.

- **IBM Tivoli Enterprise Console Command and Task Reference, SC32-1232**
  Provides details about IBM Tivoli Enterprise Console commands, predefined tasks that are shipped in the task library, and the environment variables that are available to tasks that run against an event.

- **IBM Tivoli Enterprise Console Installation Guide, SC32-1233**
  Describes how to install, upgrade, and uninstall the IBM Tivoli Enterprise Console product.

- **IBM Tivoli Enterprise Console Rule Developer’s Guide, SC32-1234**
  Describes how to develop rules and integrate them for event correlation and automated event management.

- **IBM Tivoli Enterprise Console Rule Set Reference, SC32-1282**
  Provides reference information about the IBM Tivoli Enterprise Console rule sets.

  Provides an overview of the IBM Tivoli Enterprise Console product and describes how to configure and use the IBM Tivoli Enterprise Console product to manage events.

  Describes how to install and configure the warehouse enablement pack for the IBM Tivoli Enterprise Console product and describes the data flow and structures that are used by the warehouse pack.

- **Tivoli Event Integration Facility Reference, SC32-1241**
  Describes how to develop your own event adapters that are tailored to your network environment and the specific needs of your enterprise. This reference also describes how to filter events at the source.
Related publications

The Tivoli Software Glossary includes definitions for many of the technical terms related to Tivoli software. The Tivoli Software Glossary is available, in English only, at the following Tivoli software library Web site:


Access the glossary by clicking the Glossary link on the left pane of the Tivoli software library window.

The following list identifies additional resources for developing rule-writing skills. These resources were available when the IBM Tivoli Enterprise Console Rule Developer’s Guide was published. This list is not exhaustive of available resources, and IBM does not provide opinions or recommendations about any of the publications in the list.

- Programming Rules for the Tivoli Enterprise Console
  This training course offered by IBM Global Services covers introductory rule writing using the Tivoli Enterprise Console rule language. You learn to program custom rules to respond to events, correlate multiple events, automate system administration tasks, and read trace output to verify and troubleshoot rules. Visit the Tivoli Worldwide Education Web site at http://www.ibm.com/tivoli/education/.

- Event Management and Monitoring Design Methodology
  The Event Management and Monitoring Design Methodology (EMMD) is a service offering from IBM Global Services. It provides Tivoli software customers with robust, detailed designs for implementation of Tivoli Availability products. The EMMD includes a systematic analysis of your business systems with a focus on event management and distributed monitoring requirements. Based upon this analysis, you and IBM consultants create a set of detailed specifications for monitoring, event filtering, event forwarding, and event correlation customized for the end-to-end management of your systems. Contact IBM at wribmtiv@us.ibm.com.

- Real-world examples of rules, BAROC files, and event relationship diagrams are available on the event server host in the $BINDIR/TME/TEC/samples/correlation directory.

- TEC Rule Writing by Example
  This document describes the basics of rule writing using some of the rule language predicates, and shows some of the advanced techniques rule developers commonly use. It is authored by Giles McGarry of Orb Data Limited and is available in PDF at http://www.orb-data.com.


Accessing publications online

The documentation CD contains the publications that are in the product library. The format of the publications is PDF, HTML, or both. Refer to the readme file on the CD for instructions on how to access the documentation.
IBM posts publications for this and all other Tivoli products, as they become available and whenever they are updated, to the Tivoli Software Information Center Web site. Access the Tivoli Software Information Center by first going to the Tivoli software library at the following Web address:


Scroll down and click the Product manuals link. In the Tivoli Technical Product Documents Alphabetical Listing window, click the IBM Tivoli Enterprise Console link to access the product library at the Tivoli Information Center.

**Note:** If you print PDF documents on other than letter-sized paper, select the Fit to page check box in the Adobe Acrobat Print window. This option is available when you click File → Print. **Fit to page** ensures that the full dimensions of a letter-sized page print on the paper that you are using.

### Ordering publications

You can order many Tivoli publications online at the following Web site:


You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, see the following Web site for a list of telephone numbers:

http://www.ibm.com/software/tivoli/order-lit/

### Contacting software support

If you have a problem with any Tivoli product, refer to the following IBM Software Support Web site:


If you want to contact software support, see the IBM Software Support Guide at the following Web site:

http://techsupport.services.ibm.com/guides/handbook.html

The guide provides information about how to contact IBM Software Support, depending on the severity of your problem, and the following information:

- Registration and eligibility
- Telephone numbers and e-mail addresses, depending on the country in which you are located
- Information you must have before contacting IBM Software Support
Participating in newsgroups

User groups provide software professionals with a forum for communicating ideas, technical expertise, and experiences related to the product. They are located on the Internet and are available using standard news reader programs. These groups are primarily intended for user-to-user communication and are not a replacement for formal support.

To access a newsgroup, use the instructions appropriate for your browser.

Use these instructions for a Microsoft Internet Explorer browser.
1. Open an Internet Explorer browser.
2. From the Tools menu, click Internet Options.
3. On the Internet Options window, click the Programs tab.
4. In the Newsgroups list, click the Down Arrow and then click Outlook Express.
5. Click OK.
6. Close your Internet Explorer browser and then open it again.
7. Cut and paste the newsgroup address of a product into the browser Address field, and press Enter to open the newsgroup.

Use these instructions for a Netscape Navigator browser.
1. Open a Netscape Navigator browser.
2. From the Edit menu, click Preferences. The Preferences window is displayed.
3. In the Category view, click Mail & Newsgroups to display the Mail & Newsgroups settings.
4. Select the Use Netscape mail as the default mail application check box.
5. Click OK.
6. Close your Netscape Navigator browser and then open it again.
7. Cut and paste the newsgroup address of a product into the browser Address field, and press Enter to open the newsgroup.

IBM Tivoli Enterprise Console

news://news.software.ibm.com/ibm.software.tivoli.enterprise-console

IBM Tivoli NetView® for UNIX and IBM Tivoli NetView for Windows

news://news.software.ibm.com/ibm.software.tivoli.netview-unix-windows

Conventions used in this guide

This guide uses several conventions for special terms and actions, operating system-dependent commands and paths, and margin graphics.

Typeface conventions

This guide uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes,
multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as Tip:, and Operating system considerations:)

- Column headings in a table
- Keywords and parameters in text

**Italic**
- Citations (titles of books, diskettes, and CDs)
- Words defined in text
- Emphasis of words (words as words)
- Letters as letters
- New terms in text (except in a definition list)
- Variables and values you must provide

**Monospace**
- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

### Operating system-dependent variables and paths

This guide uses the UNIX convention for specifying environment variables and for directory notation.

When using the Windows command line, replace `$variable` with `%variable%` for environment variables and replace each forward slash (`/`) with a backslash (`\`) in directory paths.

**Note:** If you are using the bash shell on a Windows system, you can use the UNIX conventions.

---

### IBM Tivoli Enterprise Console icons

The following icons are used in the IBM Tivoli Enterprise Console product.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Represents</th>
</tr>
</thead>
</table>
| ![Event Server](event_server_icon.png) | Event Server  
The red arrow indicates the event server is running.  
A hollow arrow indicates the event server is initializing. |
| ![Default](default_icon.png)           | Rule Base  
The red arrow indicates the rule base is active. |
<table>
<thead>
<tr>
<th>Icon</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Folder Icon" /></td>
<td>Profile</td>
</tr>
<tr>
<td><img src="image" alt="NT Icon" /></td>
<td>A profile is a collection of application-specific data.</td>
</tr>
</tbody>
</table>
Part 1. Correlating events at the event server
Chapter 1. Rule development fundamentals

Proactive systems management identifies problems early and addresses them quickly. This is accomplished by automating both performance monitoring and problem management. Event management, using the Tivoli Enterprise Console, provides the link between performance monitoring and problem management by receiving information from monitoring sources and initiating management responses.

Events are generated to notify administrators of changes or problems. The Tivoli Enterprise Console can receive these events and provide automated responses based on its rules. The rules can perform responses such as further notification (page or e-mail an administrator), create or update trouble tickets, run programs, and so forth.

In addition to providing automated responses, the IBM Tivoli Enterprise Console product console serves as a window into the system by displaying incoming events to administrators. To provide the most accurate information to administrators, the event console should be configured to display only those events that are significant to the enterprise. Of these enterprise-significant events, those that are most important and require immediate attention should be emphasized. This can be accomplished with rules by various means, including:

- Duplicate detection: Automatically discard duplicate events within a specific time interval.
- Thresholding: Accumulate events until a certain number are received within a time interval, then issue one representative event.
- Escalation: Increase the severity of an event or perform an action if a certain number of events are received within a time interval.
- Correlation: Based on the relationships between events and system problems, emphasize the most important event relating to a problem (typically the root cause of the problem) and reduce the significance of those events that are merely effects that are corrected when the root cause is corrected. For example, if an uninterruptible power supply shuts down a system gracefully during a power failure, the Tivoli Enterprise Console product can indicate that the root cause of the shut down is the power failure before personnel are dispatched to repair the system.

Designing rules

Rules should be based on an analysis of the events in your environment. Ideally, this analysis should examine all of the events each hardware and software component can generate and then determine the required handling for each event. The rule base should generally be developed as follows:

1. Compile a list of all the hardware and software components to manage. The Tivoli Inventory product can help with this task.
2. Compile a list of all the Tivoli Enterprise Console events each component can generate. This information might be available from product documentation, MIB files, event class definition files, or directly from a manufacturer.
3. Examine the list of events to determine which events are:
   - Enterprise-significant and to be sent to the enterprise-level Tivoli Enterprise Console event server.
• Domain-significant and to be handled at a Tivoli Availability Intermediate Manager or lower-level Tivoli Enterprise Console event server.

**Note:** Tivoli Availability Intermediate Manager refers to a separate Tivoli software product that processes Tivoli Enterprise Console events using a Tivoli Enterprise Console rule base and rule engine located locally on each Tivoli Availability Intermediate Manager host. See the *Tivoli Availability Intermediate Manager User’s Guide* for additional information.

• Insignificant to the enterprise and to be discarded at the source.

4. For each event handled by a Tivoli Enterprise Console event server or a Tivoli Availability Intermediate Manager, determine the required processing. For example, duplicate detection, thresholding, issuing trouble tickets, escalation, running a script, and so forth.

5. Determine and record the causal relationships between events; that is, record which events precede or follow other events. This information is used to create rules that perform correlation between events. Start by recording the relationships between events from a single component before recording the relationships of events among components.

6. With all this information, create the rules to manage the events in your environment. In creating rules, it might be helpful to refer to the rule sets shipped with the product. For more information about these rule sets, see the *IBM Tivoli Enterprise Console Rule Set Reference*.

Chapter 5, “Correlation examples”, on page 217 contains examples of rules that correlate events. For more information about designing event classes, see “Designing event classes” on page 35.

If you need help with event management analysis and design, IBM Global Services provides the Event Management and Monitoring Methodology service offering. This methodology includes a systematic analysis of your business systems with a focus on event management and distributed monitoring requirements. For additional information, see the description in “Related publications” on page xi.

---

**Events**

The central unit of information for the IBM Tivoli Enterprise Console is the event. An event represents a significant change in the state of a system resource or application. Event information includes the name of the source and a description of the condition. The source is a system resource or application being monitored by an event adapter. The information is formatted by the event adapter into a data stream understandable by the event server. The event server adds additional information when it stores the event as a database entry and processes it. Event information is provided in the form of attributes, which are `name=value` pairs.

**Event classes** are a classification of events. They are the protocol between an event adapter and event server for determining the information that can be sent. The event class name is also sent as event information, although not in the format of an attribute.

**Event adapters**

Event adapters are processes that monitor sources. When an event adapter receives information from its source, it formats the information and sends it to an event...
A set of event adapters is shipped with the IBM Tivoli Enterprise Console. Other products also ship event adapters, and you can create your own event adapter using the Event Integration Facility (EIF). See the IBM Tivoli Enterprise Console Adapters Guide for additional information about adapters shipped with the IBM Tivoli Enterprise Console. See the IBM Tivoli Enterprise Console Event Integration Facility Reference for additional information about creating your own event adapters.

**Event attributes supplied by an event adapter**

An event adapter does not have to provide values for all attributes. It must at least supply the source and event class values, and usually supplies the origin and host name values; other values are supplied by the event server. If an event adapter does not supply a value for a particular attribute, the attribute is given the default value specified in the Basic Recorder of Objects in C (BAROC) file for the particular event class at the event server. BAROC files are described in Chapter 2, "Event class concepts", on page 35. For additional information about attributes supplied by event adapters, see the IBM Tivoli Enterprise Console Adapters Guide. For additional information about attributes supplied by the event server, see “Event attribute values supplied by the event server” on page 7.

The following sequence of figures illustrates the flow of event information from a Windows NT® application event log being monitored by a Windows NT event log adapter. The first figure shows a Windows NT event log entry in the Windows NT Event Viewer. The second figure shows a Tivoli-formatted event sent from the adapter, and received and stored in the reception log of the event server (using the wtdumpri command to display the reception log). The third figure shows an event in the Details dialog of the event console.

1. The following figure shows the Windows NT event log entry in the Windows NT Event Viewer.
2. The following figure shows the Tivoli-formatted event sent from the adapter, and received and stored in the reception log of the event server (using the `wtdumpfl` command to display the reception log).

![Tivoli-formatted event](image)

3. The following figure shows the event in the Details dialog of the event console.

![Event Details](image)

**Events sent from other products or the command line**

Events can also be sent to the IBM Tivoli Enterprise Console from other products and components, such as IBM Tivoli Monitoring, the NetView component, Security Management, and Workload Scheduler, among others.

You can send events from a shell command line using the `wpostemsg`, `postemsg`, `wpostzmsg`, and `postzmsg` commands. These are mostly used for testing purposes but you can also use them in shell scripts. See the *IBM Tivoli Enterprise Console Command and Task Reference* for additional information about these commands.

**Note:** Do not send TEC_Start and TEC_Stop events to the event server. These events are intended for internal use only.

**Event filtering**

Processing at the event server should be dedicated to those events that you have designated as enterprise significant. You have various means for filtering events that you do not want to send across the network or use processing resources at the event server. Filtering events closest to the their source saves you both network and processing resources, in addition to eliminating clutter on an administrator’s event console.
The following list illustrates some options you have for filtering events:

- For some system resources, you can filter at the agent level. Some examples are:
  - Cabletron products let you filter by sending all or no traps, filter by destination, filter by message, and filter by event class.
  - Shiva products let you filter by trap severity level.
  - Net Ware products let you filter by alarm type, destination for alarms, duplicate alarms, specific alarms, and alarm severity level.
  - Cisco products let you filter by specifying which traps to send to which recipients.
- For logfile-type adapters that manage an ASCII log file, you can create format files that only map significant events to event classes.
- For the Windows NT event log adapter and the Windows event log adapter, you have prefiltering capabilities available as a configuration file option for filtering out native events before adapter processing (PreFilter option). For the Windows event log adapter, you can also specify which particular event logs to monitor (WINEVENTLOGS option).
- For most adapters, you have filtering capabilities available during adapter processing as a configuration file option (Filter option).
- With the Tivoli Event Integration Facility component and the Tivoli Enterprise Console gateway, you can perform state correlation. State correlation analyzes incoming events for user-defined states to suppress duplicate events, identifies event thresholds, and collects or groups similar events.
- With the Tivoli Availability Intermediate Manager, you create event filters and also perform Tivoli Enterprise Console rule processing. It contains a version of the Tivoli Enterprise Console rules engine.
- A lower-level Tivoli Enterprise Console installation can filter events with rules processing and forward only significant events to a higher-level installation.

### Special characters in events

The characters Ctrl+A and Ctrl+B cannot be sent as data in an event. If they must be included in an event, use the escape sequences \001 and \002 instead.

### Event attribute values supplied by the event server

The event server adds the following information to the attributes of a valid event:

- **acl**
  The list of authorization roles that enables an administrator to modify the event.

- **administrator**
  The administrator who acknowledged or closed the event.

- **cause_date_reception**
  The value of the date_reception attribute of the specified cause event. This value is used to link an effect event to its cause event. The cause_event_handle attribute is also needed to link an effect event to its cause event.

- **cause_event_handle**
  The value of the event_handle attribute of the specified cause event. This value is used to link an effect event to its cause event. The cause_date_reception attribute is also needed to link an effect event to its cause event.
credibility
Indicates how the event was sent from the adapter. The value is 1 if an event was sent using a communications channel provided by Tivoli Management Framework services—as is the case for a TME® adapter. The value is 0 if an event was sent from a non-TME adapter.

date_reception
A time stamp indicating the time the event server received the event. It is an integer representing the number of seconds since the epoch, which is January 1, 1970. This value is used as a component to uniquely identify the event. An event is uniquely identified by a combination of the values for the date_reception, event_handle, and server_handle attributes.

duration
For closed events, the age (in seconds) of the event from when it was received by the event server until it was closed. For all non-closed events, the value is 0.

Note: If an event was closed by calling the set_event_status predicate from within a rule, this attribute is not modified to give the age. The value remains at 0.

event_handle
A number used to reference the event. An event is uniquely identified by a combination of the values for the date_reception, event_handle, and server_handle attributes.

num_actions
The number of actions (tasks or programs) currently being tracked by the event server for this event.

server_handle
A number identifying the event server that received this event. An event is uniquely identified by a combination of the values for the date_reception, event_handle, and server_handle attributes.

server_path
A list of elements that provides information about each event server through which an event has passed. Each element value is in the same format as an event ID. Event ID format is described in “Event cache” on page 9. The re_split_event_id predicate is provided to parse this value in a rule. See “re_split_event_id” on page 192 for additional information.

status
The status of an event. It is initially set to OPEN or to a default value specified by the event class. Possible values during an event’s lifetime are as follows:

ACK An administrator or rule has acknowledged the event.

CLOSED An administrator or rule has fixed the problem that was reported by the event. An event adapter can also send an event with a status of CLOSED to indicate that a previously received event of the specified class should have its status changed to CLOSED; the previously received event to be closed is the most recent duplicate of the same event. The event being sent with a CLOSED status is dropped and not stored in the event repository.

custom_status
A status that has been added to the STATUS enumeration for site-specific purposes. The STATUS enumeration as shipped is
defined in the root.baroc file. To add a new status, edit this file, recompile the rule base, and restart the event server.

OPEN  The event has been received by the event server, but no administrator or rule has acknowledged it.

RESPONSE  A rule has automatically responded to the event. This status is assigned by a rule language predicate. It is not available from an event console.

---

Event cache

The event cache is basically a list of received events in RAM that have been through rule processing. The default size is 1000 events. It is configured with the `wsetesvrcfg` command or from the Event Server Parameters dialog.

Events are uniquely identified by a number that is a combination of the following event attributes, sometimes referred to as an event ID:

**event_handle**
If this is the first event received within a second, the value is 1. If more than one event is received within the same second, this value is incremented for each subsequent event received within the second.

**server_handle**
This value is 1 for the event server in the local Tivoli Management Region.

**date_reception**
The number of seconds since the *epoch*.

The following figure illustrates how event IDs are constructed.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1189578295</td>
<td>Tec_Start</td>
</tr>
<tr>
<td>1189578296</td>
<td>UPS_Fan_Down</td>
</tr>
<tr>
<td>2189578296</td>
<td>universal_oserv</td>
</tr>
<tr>
<td>1189578315</td>
<td>UPS_Temp_Degraded</td>
</tr>
<tr>
<td>1189578420</td>
<td>NFS_No_Response</td>
</tr>
<tr>
<td>2189578420</td>
<td>UPS_Fan_Up</td>
</tr>
</tbody>
</table>

---

Rules

A Tivoli Enterprise Console rule is a construct that lets you specify what action to take when a certain event is received. Rules are written in a high-level language called the rule language. The rule language provides a simplified interface to the Prolog programming language, which is the language actually used internally by the rule engine. Your rules in the rule language are precompiled into Prolog source code, which is then compiled into Prolog executable files.
As part of the rule language, a set of predefined predicates is provided. These predicates are frequently used actions in rules. Reference information about these predicates can be found in “Rule language predicates” on page 69.

Although knowledge of Prolog is not required to develop rules, you can use its features in conjunction with the rule language to create sophisticated actions for certain rule types. Complete details about Prolog are beyond the scope of this document, but Appendix A, “Using Prolog in rules”, on page 345 provides an overview and reference material covering some useful Prolog features you can use for rule development. Also, “Related publications” on page xi lists some publications about Prolog you might find helpful.

Rule types

There are five types of rules:

Plain rule
Used with incoming new events, or with previously received events to be re-analyzed. Re-analysis of a previously received event is called a redo request.

Plain rules allow you the flexibility to use any predicate or Prolog feature in its actions.

Change rule
Used with previously received events that have a request to change their information. A request to change an event’s information is called a change request. For change requests, the change rules are checked before the change is actually made. This timing lets you develop rules to take action depending on the old value of an attribute, the new value of the attribute, and the origin of the change request. Change requests can be generated by:

- An event console, for example, an administrator changes the status of an event from OPEN to CLOSED
- Calling certain predicates within rules, for example, the place_change_request predicate
- Receiving an event from an adapter with a value of CLOSED for the status attribute

Change rules allow you the flexibility to use any predicate or Prolog feature in its actions.

Change rules can only specify plain actions. Redo actions and reception actions are considered errors when they are specified in change rules. See “Action types” on page 68 for additional information.

Timer rule
Used when a previously set timer on an event expires. Timers can be set on an event with the set_timer predicate in a rule. Sometimes you might want to wait for a period of time so related events come in that help identify the root cause of a problem, or perhaps you want to wait to ensure the event condition lasts long enough to be a problem where action is needed. With timer rules, you have the flexibility to use any predicate or Prolog feature in its actions.

Timer rules can only specify plain actions. Redo actions and reception actions are considered errors when they are specified in timer rules. See “Action types” on page 68 for additional information.
Simple rule
Used with incoming new events, or with a redo request. A simple rule is not as flexible as a plain rule, for example, it contains predefined conditions and actions, and you cannot use a predicate or any Prolog feature in its actions. A simple rule does not do any correlation with other events in the event cache, except for dropping duplicate events.

Correlation rule
Used with incoming new events, or with a redo request. A correlation rule lets you establish a causal relationship between two event classes. One event either causes the other to be generated, or one event causes the other to be closed. With a correlation rule, you can propagate the value of the status attribute from a cause event to an effect event. For example, when closing a cause event, a linked effect event can be automatically closed, also. Correlation rules are called compound rules in the rule builder dialogs.

General rule structure
Rules are generally structured as shown in the following pseudocode. Of course, each rule type might have a little different structure and syntax. Rule structure for each type of rule is described in “Rule structure” on page 53.

rule_type: rule_name:
{
    description
    event_filter
    action1
    action2
    ...
    ).

The following figure shows an example of the structure of a plain rule.
What causes a rule to run

A rule is run when the event under analysis has satisfied all of the conditions specified in the rule’s event filter. An event filter can contain tests for an event class name and event attribute conditions. Event filters are described in detail in “Event filters” on page 60.

Example actions you can take with rules

The following list provides a few examples of actions you can take with rules:

• Search the event cache for:
  – The first instance of a particular event
  – All instances of a particular event
  – Duplicates of a particular event
• Link cause events to effect events so the effect event has a reference to the cause event.
• Modify events, for example, change its status or severity.
• Run Tivoli tasks or external programs.
• Generate new events.
• Set timers on events, so that when a timer expires the event is evaluated against the timer rules.

Rule bases

Generally speaking, a rule base is a collection of event class definitions, rules that apply to those event classes, and predicates that are used by rules. A rule base on the IBM Tivoli Enterprise Console event server is the master container from which rule base targets are created. Rule base targets are the actual rule bases used by rule engines to process events. Depending on the context, the terms rule base and rule base target can be used interchangeably. Rule base targets are described further in “Rule base targets” on page 14.

You can have only one rule engine processing events in your environment or you can have multiple rule engines. An example of using multiple rule engines is when you deploy multiple Tivoli Availability Intermediate Manager event servers that not only manage events locally at each Tivoli Availability Intermediate Manager event server, but also have the capability to send events to other Tivoli Availability Intermediate Manager event servers and the IBM Tivoli Enterprise Console event server. When there is more than one rule engine managing events in the environment, the rule bases used by the rule engines in that environment are referred to as distributed rule bases.

In a distributed rule base environment, event classes and rules must be synchronized among all the rule engines. To keep these synchronized, all rule base development must be done with the IBM Tivoli Enterprise Console event server, which is the centralized point of control for managing a distributed rule base environment. Rule bases needed by other event servers in the distributed environment (such as Tivoli Availability Intermediate Manager event servers) are obtained as rule base targets created by the IBM Tivoli Enterprise Console event server.

Note: For information about importing rule bases into a Tivoli Availability Intermediate Manager event server and distributing them, see the Tivoli Availability Intermediate Manager User’s Guide.
Because of the support for distributed rule bases, creation of rule bases has changed slightly from IBM Tivoli Enterprise Console release 3.6.2 and earlier. Whether your environment contains only one rule engine at the IBM Tivoli Enterprise Console event server or you have multiple rule engines in your environment, it is recommended that you use the \texttt{wrb} command for rule base manipulation procedures. This command provides more flexibility and additional function than the current rule builder available from the Tivoli desktop. Do not modify any files used internally by an event server to manage rule bases with a text editor—use the \texttt{wrb} command or the rule builder to manipulate rule bases.

\begin{description}
\item[Notes:]\\
1. In a single rule engine environment with only the IBM Tivoli Enterprise Console event server processing events, rule bases compiled with the rule compiler from IBM Tivoli Enterprise Console release 3.6.2 and earlier work as-is, although it is recommended that you upgrade your rule bases so they use the rule compiler provided with release 3.7 and later. A rule base upgrade script is provided by IBM. See the \texttt{wrbupgrade} command in the \textit{IBM Tivoli Enterprise Console Command and Task Reference} for additional information.

In a distributed rule base environment containing multiple rule engines, the rule compiler provided with release 3.7 and later must be used to compile rule bases. Although an older rule base (pre-release 3.7) might appear to compile successfully with the newer compiler, the proper objects for a distributed rule base environment are not created. Older rule bases must be upgraded before being used in a distributed rule base environment.

2. When importing an object into a rule base (for example, rule sets, event classes, rule packs into rule base targets, and so forth), an object that already exists in the rule base must be deleted before you can replace it with a newer version of the object.
\end{description}
The following figure illustrates the different parts that can comprise a rule base.

Rule base targets
A rule base target is the actual rule base that is used by a rule engine. Each rule base target has a meaningful logical name that you provide.

Rule base targets, after compilation of the rule base on the IBM Tivoli Enterprise Console event server, are located in the `rule_base_directory/.rbtargets/target_name` directories (note the leading period in the `.rbtargets` subdirectory name). The name for the rule base target used by the rule engine on the IBM Tivoli Enterprise Console event server is `EventServer`. The `EventServer` rule base target is automatically created in every rule base. Other distributed event servers (for example, Tivoli Availability Intermediate Manager event servers) retrieve copies of rule base targets for their use. Distributed event servers only need the rule base target directory structure starting from the `target_name` subdirectory for their use.

The event classes and predicates in a rule base target are the same throughout a distributed environment, and are replicated from the rule base on the IBM Tivoli Enterprise Console event server to each rule base target during compilation.

The rule sets in rule base targets can differ, depending on design and implementation of distributed rule bases in your environment. Because of this, you must specify which rule sets are to be included with each rule base target. There are multiple ways to do this with various options of the `wrb` command. For example, you can:
• Use the –crttarget option with the –import suboption. These options create a rule base target and import rule sets into it.
• Use the –imptgtgrule option. This imports rule sets into an existing rule base target.

To list the rule base targets in a rule base, you can use the –lsrbtarget option with the –detailed suboption.

Supporting data files can be distributed with the rule base target when the rule base is compiled. For example, you can distribute an event forwarding configuration file (for the `re_send_event_conf` predicate) or a Prolog fact or data file. This can be done by using the –imptgtdata option of the `wrb` command.

See the IBM Tivoli Enterprise Console Command and Task Reference for complete details about the `wrb` command.

**Rule sets and rule packs**

Rule sets are the files that contain rules. Typically, related rules are contained within a rule set. Rule sets can be imported into a rule base target using the `wrb` command, as mentioned in “Rule base targets” on page 14. When a rule base is compiled, rule sets are replicated to those rule base targets that have specified which rule sets to import.

When a rule base is being used by a rule engine, generally the rules are processed in the order defined within a rule set and within the order of how the rule sets were imported into the rule base target. The regular rule processing order can be altered with the use of certain predicates called from within rules.

**Note:** The order of rule sets defined for a rule base target is important, because it affects rule engine performance. Placement of rule sets determines evaluation order by the rule engine.

A default set of rule sets is provided by IBM with the Default rule base. A default rule set for the Tivoli Availability Intermediate Manager is also included with the Default rule base. For more information, see the IBM Tivoli Enterprise Console Rule Set Reference.

Another way to import rule sets into a rule base target are with rule packs. Rule packs are a convenient way to package a group of rule sets so they can be imported into a rule base target in a single operation. Rule packs are used to combine a group of rule sets that are used in multiple rule base targets. When a rule base is compiled, those rule base targets that are defined to receive rule packs receive their rule pack contents, which are rule sets.

Before rule sets and rule packs can be imported into rule base targets, they must first be imported into the rule base on the IBM Tivoli Enterprise Console event server. Again, the rule base on the IBM Tivoli Enterprise Console event server is the master container for all of the objects that comprise rule base targets and rule base targets are the actual rule bases loaded into rule engines. There are multiple ways to import rule sets and rule packs into a rule base, using various options of the `wrb` command. For example, you can use:

• The –cprb option with the –rulesets and –rulepacks suboptions. This copies an existing rule base into another rule base and also copies the rulesets and rule packs from the source rule base.
• The –crtrp option with the –import suboption. This creates a rule pack and imports rule sets into it.
• The –imprprule option. This imports rule sets into an existing rule pack.
• The –imprbrule option. This imports rule sets into a rule base.

To list the rule packs in a rule base, you can use the –lsrbrpack option with the –detailed suboption.

To list the rule sets in a rule base, you can use the –lsbrrule option.

The following figure illustrates the relationships and identifies the procedures that can be used for manipulating rule sets and rule packs in a rule base.

See the IBM Tivoli Enterprise Console Command and Task Reference for complete details about the wrb command.

You can also import rule sets into a rule base and compile a rule base using the rule builder. See “Rule base manipulation procedures using the rule builder” on page 22 for additional information.

**Rule pack example**: Suppose an existing rule base named Production exists on the IBM Tivoli Enterprise Console event server. Using the wrb command as shown in the following example lists its rule sets. The $ character shown in the following examples is the command-line prompt.

```
$ wrb -lsbrrule Production
Rule Set files
-------------
ov_default.rls
```
Of the rule sets shown in the preceding list, the first two contain rules that, for this example, need to be imported into every rule base target except the EventServer target. If the Production rule base had three rule base targets named Support, Mail, and Development, the three rule sets can be added in eight separate operations by importing each rule set individually with the `wrb –imprbrule` command, or they can be added in only three separate operations by importing a rule pack containing the three rule sets with the `wrb –imprprule` command.

To import with a rule pack, the following procedure can be followed. Each step shows the appropriate command to issue.

1. Create a rule pack named defaultRules in the Production rule base.
   ```
   $ wrb -crtrp defaultRules Production
   ```
2. List all of the rule packs in the Production rule base to verify the previous step.
   ```
   $ wrb -lsrbpack Production
   Rule Pack Files
   --------------
   defaultRules
   ```
3. Create a rule pack by importing the two rule sets into it.
   ```
   $ wrb –imprprule ov_default defaultRules Production
   $ wrb –imprprule log_default defaultRules Production
   ```
   The rule sets can also be imported into the rule pack when the rule pack is created.
   ```
   $ wrb -crtrp defaultRules -import ov_default log_default Production
   ```
4. Import the rule pack into each rule of the three rule base targets.
   ```
   $ wrb –imptgtrule defaultRules Support Production
   $ wrb –imptgtrule defaultRules Mail Production
   $ wrb –imptgtrule defaultRules Development Production
   ```

When a new rule base target is created in the Production rule base, and the target requires the rule sets in the defaultRules rule pack, you can import the rule pack into the target in the same step as creating the target. For example, the following command creates a new rule base target named Marketing and imports the defaultRules rule pack in the same operation:
```
$ wrb -crttarget Marketing -import defaultRules Production
```

When the Production rule base is compiled, the rule sets that comprise the defaultRules rule pack are automatically replicated to the rule base targets.

**Note:** Deleting a rule set from a rule pack does not delete it from any rule base targets that have imported the rule pack.

**Modifying rule sets:** The following procedure describes how to modify a rule set. You can use the `wrb` command or the rule builder for the rule base manipulation steps.

**Note:** A rule set must be deleted from a rule pack before it can be modified.
The `wrb` command syntax is shown in the following steps. See “Rule base manipulation procedures using the rule builder” on page 22 for information about performing these steps with the rule builder.

1. Make modifications to the file with a text editor in a different directory than the rule base directory structure.
2. Delete the old file from the rule base.
   
   ```bash
   wrb -delrbrule rule_set rule_base
   ```
3. Import the new file into the rule base.
   
   ```bash
   wrb -imprbrule rule_set.rls rule_base
   ```
4. Compile the rule base. The new file is replicated to all rule base targets defined in the rule base.
   
   ```bash
   wrb -comprules rule_base
   ```

**Event classes**

All of the rule base targets defined for a rule base must use the same set of event classes. This is because rule engines within a distributed rule base environment have the capability to forward events to other rule engines within that environment. If event classes are not synchronized among them, an event is discarded if received by an event server that has no knowledge of the class of the received event. When an event is received at an event server and its class is not in that event server’s rule base, the event is given a status of PARSING_FAILED in the event server’s reception log and the event is discarded.

Because all rule base targets for a rule base use the same set of classes, all rule builder and `wrb` commands manipulate BAROC files at the rule base level on the IBM Tivoli Enterprise Console event server. When the rule base is compiled, the event classes are replicated to the rule base targets defined in the rule base.

Notes:

1. The order of BAROC files is important. Placement of the files determines evaluation order at run time.
2. The root.baroc and tec.baroc files are imported automatically when you create a rule base.

To manipulate BAROC files, see “Rule base manipulation procedures using the rule builder” on page 22 for rule builder procedures, and the IBM Tivoli Enterprise Console Command and Task Reference for using the `wrb` command.

For additional information about BAROC files, see Chapter 2, “Event class concepts”, on page 35

**Predicates**

Like event classes, rule language predicates provided by IBM and predicates that you create must be stored at the rule base level on the IBM Tivoli Enterprise Console event server. When the rule base is compiled, these predicates are replicated to the rule base targets defined in the rule base. There are no commands for manipulating predicates. Predicates are imported automatically when you create a rule base.

Rule language predicates are described in “Rule language predicates” on page 69

Some of the Prolog built-in predicates can be used as building blocks to create your own predicates. The Prolog predicates are described in Appendix A, “Using Prolog in rules”, on page 345
Rule base example
This example describes a process for creating a rule base using the `wrb` command and some of its options. There are multiple ways to create a rule base with the `wrb` command; this example simply shows one way. To fully understand the many capabilities of the `wrb` command, see the IBM Tivoli Enterprise Console Command and Task Reference. For information about rule base manipulation procedures using the rule builder, see “Rule base manipulation procedures using the rule builder” on page 22.

The event management environment for this example consists of an IBM Tivoli Enterprise Console event server and two Tivoli Availability Intermediate Manager event servers. The Tivoli Availability Intermediate Manager event servers are configured to process most events locally, only sending those of significance to the IBM Tivoli Enterprise Console event server for further processing. Events of significance were identified during an analysis phase of event management design and are detected by logic programmed into the rules.

Requirements for rule base example: The following list describes the requirements for this example:

- The rule base name is Operations. It is used for managing events for the operations organization. The personnel and accounting units comprise the operations organization.
- The rule base is created in the `/tec_rule/Operations` directory.
- The following rule base targets are created in the Operations rule base, as they represent the rule engines that will make use of the Operations rule base:
  - EventServer: The rule base for the rule engine on the IBM Tivoli Enterprise Console event server. This rule base target is automatically created.
  - aimPersonnel: The rule base for the rule engine on the Tivoli Availability Intermediate Manager event server that manages events for the personnel unit.
  - aimAccounting: The rule base for the rule engine on the Tivoli Availability Intermediate Manager event server that manages events for the accounting unit.

Assumptions for rule base example: The following list describes the assumptions for this example:

- Event class definitions from the Default rule base provided by IBM are imported into the rule base. The BAROC file needed is `tecad_nt.baroc`, which defines event classes for the Windows NT event log adapter. The Default rule base is located in the `$BINDIR/TME/TEC/default_rb` directory. The BAROC files are located in the TEC_CLASSES subdirectory.
- The following rule sets have already been created but need to be imported into the rule base. They are stored in the `/tec_rule_dev/rls` directory:
  - `tec_server.rls`: The rule set for processing events at the IBM Tivoli Enterprise Console event server. These rules handle events generated by the IBM Tivoli Enterprise Console event server and those that are forwarded by the two Tivoli Availability Intermediate Manager event servers.
  - `aim_ops_perf.rls`: The rule set for processing performance related events on both Tivoli Availability Intermediate Manager event servers. This rule set is common to both.
  - `aim_ops_sec.rls`: The rule set for processing security related events at both Tivoli Availability Intermediate Manager event servers. This rule set is common to both.
- `aim_pers.rls`: The rule set for processing personnel application related events at the Tivoli Availability Intermediate Manager event server for the personnel unit.

- `aim_acct.rls`: The rule set for processing accounting application related events at Tivoli Availability Intermediate Manager event server for the accounting unit.

- The `aimOps` rule pack needs to be created. It contains the `aim_ops_perf.rls` and `aim_ops_sec.rls` rule sets. These rule sets are common to both Tivoli Availability Intermediate Manager event servers.

The following figure illustrates the topology for the rule base example.
The following figure illustrates the different parts that comprise the Operations rule base at the IBM Tivoli Enterprise Console event server.

**Rule base - Operations**

- **Rule sets**
  - tec_server
  - aim_ops_perf
  - aim_ops_sec
  - aim_acct
  - aim_pers

- **Event classes**
  - root.baroc
  - tec.baroc.
  - tecad_nt.baroc

- **Predicates**
  - event_specifiers.wic
  - filters.wic
  - templates.wic

**Rule packs**

- aimOps
  - aim_ops_perf
  - aim_ops_sec

**Rule base targets**

- **EventServer**
  - Rules sets
    - tec_server
  - Event classes and predicates
    - From rule base

- **aimPersonnel**
  - Rules sets
    - aimOps
      - aim_ops_perf
      - aim_ops_sec
      - aim_pers
  - Event classes and predicates
    - From rule base

- **aimAccounting**
  - Rules sets
    - aimOps
      - aim_ops_perf
      - aim_ops_sec
      - aim_acct
  - Event classes and predicates
    - From rule base

**Procedure for rule base example:** To create the Operations rule base:

1. Create the rule base.
   ```bash
   wrb -crtrb -path /tec_rule/Operations Operations
   ```

2. Import classes into the rule base.
   ```bash
   wrb -imprbclass /data/TME/TEC/default_rb/ \\
   TEC_CLASSES/tecad_nt.baroc Operations
   ```

3. Import the rule sets into the rule base.
   ```bash
   wrb -imprbrule c:/tec_rule_dev/rls/tec_server.rls \\
   Operations
   wrb -imprbrule c:/tec_rule_dev/rls/aim_ops_perf.rls \\
   Operations
   wrb -imprbrule c:/tec_rule_dev/rls/aim_ops_sec.rls \\
   Operations
   wrb -imprbrule c:/tec_rule_dev/rls/aim_pers.rls \\
   Operations
   wrb -imprbrule c:/tec_rule_dev/rls/aim_acct.rls \\
   Operations
   ```

4. Verify the previous step to ensure all rule sets were successfully imported into the rule base. The $ character represents the command-line prompt.
   ```bash
   $ wrb -lsrbrule Operations
   Rule Set files
   ---------------
   ```
5. Create the rule pack in the rule base.
   `wrb -crtp aimOps -import aim_ops_perf aim_ops_sec Operations`

6. Import the rule set into the EventServer rule base target. (Remember, the
   EventServer rule base target is automatically created.)
   `wrb -imptgtrule tec_server EventServer Operations`

7. Create the aimPersonnel rule base target and import its rule pack and rule set.
   `wrb -crttarget aimPersonnel -import aimOps \aim_pers Operations`

8. Create the aimAccounting rule base target and import its rule pack and rule
   set.
   `wrb -crttarget aimAccounting -import aimOps \aim_acct Operations`

9. Compile the rule base.
   `wrb -comprules Operations`

10. Load and activate the rule base.
    `wrb -loadrb -use Operations`

11. Stop and restart the event server to use the Operations rule base. The rule
    engine on the IBM Tivoli Enterprise Console event server uses the rules, event
    class definitions, and predicates defined for the EventServer rule base target.

12. The Tivoli Availability Intermediate Manager event server that manages events
    for the personnel unit can import the aimPersonnel rule base for its rule
    engine from the `/tec_rule/Operations/.rbtargets/aimPersonnel` directory. See

13. The Tivoli Availability Intermediate Manager event server that manages events
    for the accounting unit can import the aimAccounting rule base for its rule
    engine from the `/tec_rule/Operations/.rbtargets/aimAccounting` directory.
    See the `Tivoli Availability Intermediate Manager User's Guide` for the procedure.

**Rule base manipulation procedures using the rule builder**

This section describes procedures for manipulating rule bases using the rule builder from the Tivoli desktop. Equivalent and additional functionality is available with the `wrb` command, which is described in detail in the *IBM Tivoli Enterprise Console Command and Task Reference*.

**Notes:**

1. Manipulating rule bases with the rule builder does not provide functionality for
   managing rule base targets or rule packs, or for profiling rules.

2. Rule bases created with the `wrb –crtrb` command are distributed rule bases and
   have a default rule base target named EventServer. Any rule set imported into
   the rule base using the rule builder must then be imported into the EventServer
   rule base target, or any other rule base target, using the `wrb –imptgtrule`
   command in order for the rule set to be loaded by the rule engine.

3. The rule builder cannot be used to edit rule set files that have been created
   with a text editor.

4. Rule set files that have been created with the rule builder must not be
   manually edited.

5. You can use the `upgrade_gui.sh` command to convert rules created by the
   version 3.6.2 and earlier rule builder to the rule syntax supported by the
version 3.7 and later version of the rule compiler. The converted rules take advantage of features implemented by the newer version of the compiler and are easier to read. If you convert rules with this command, you can no longer edit them with the rule builder; you must use a text editor. See the IBM Tivoli Enterprise Console Command and Task Reference for details about the upgrade_gui.sh command.

The procedures described in this section are:
- Creating a new rule base
- Editing an existing rule base
- Listing all known rule bases
- Listing the loaded rule base
- Listing rule sets and BAROC files in a rule base
- Copying a rule base
- Deleting a rule base
- Adding a rule set to a rule base
- Adding BAROC files to a rule base
- Compiling a rule base
- Loading and activating a rule base

Creating a new rule base: The rule base named Default cannot be edited. It is the default rule base provided by IBM. Therefore, you must create a new rule base to add event classes and rules that are not part of the Default rule base.

Note: In some cases, it is easier to copy an existing rule base and modify it to begin creating a new rule base. See “Copying a rule base” on page 26 for additional information.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

To create a new rule base:
1. Select Rules Bases from the Event Server icon pop-up menu to display the Event Server Rule Bases window.
2. Select **Rule Base** from the Create menu to display the Create a Rule Base dialog.

3. Enter the name of the new rule base in the Name text field.

4. Enter the absolute path name for the new rule base in the Directory Path text field or click on the Directory button to use the File Browser dialog. You can optionally specify the host that will contain the new rule base.

5. To create the new rule base and close the Create a Rule Base dialog, click on **Create & Close**. A new rule base is created and the Create a Rule Base dialog is closed. The icon representing the new rule base is displayed in the Event Server Rule Bases dialog.

**Editing an existing rule base:** An existing rule base can be edited with either the rule builder or with a text editor. If you edit a rule base with a text editor, it cannot be subsequently edited with the rule builder. The emphasis of this guide is on developing rules with a text editor, although there is information for developing rules with the rule builder. Currently, you can develop more robust rules by manually coding them in a text editor instead of using the rule builder. See Chapter 7, “Creating rules with the graphical rule builder”, on page 243 for information about editing a rule base with the rule builder.

**Listing all known rule bases:** You can list all rule bases that are known to the event server. Each rule base is represented by an icon.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>List all known rule bases</td>
<td>Event server</td>
<td>user</td>
</tr>
</tbody>
</table>
To list all known rule bases, select **Rules Bases** from the Event Server icon pop-up menu. All known rule bases are then displayed in the Event Server Rule Bases window. The loaded rule base is indicated by the icon on the left in the following figure.

![Event Server Rule Bases](image1)

**Listing the loaded rule base:** You can list the rule base currently loaded on the event server. This might or might not be the active rule base, depending upon whether it was activated when loaded or the event server was restarted after it was loaded.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>List the loaded rule base</td>
<td>Event server</td>
<td>user</td>
</tr>
</tbody>
</table>

To list the loaded rule base select **Rules Bases** from the Event Server icon pop-up menu. All known rule bases are displayed in the Event Server Rule Bases window. The loaded rule base is indicated by the icon on the left in the following figure.

![Event Server Rule Bases](image2)

**Listing rule sets and BAROC files in a rule base:** A rule base is comprised of rule sets, which are files that contain rules, and BAROC files, which are files that contain event class definitions.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>List rule sets and BAROC files contained in a rule base</td>
<td>Event server</td>
<td>user</td>
</tr>
</tbody>
</table>
To display rule set files and BAROC files in a rule base:

1. From the applicable rule base pop-up menu, select Import. The Import Into Rule Base dialog is displayed. The rule set files are shown in the top scrolling list and the BAROC files are shown in the bottom scrolling list.
2. Click Close to exit.

**Copying a rule base:** In many cases, it is easier to copy an existing rule base and then modify it when beginning development of a new rule base.

**Notes:**

1. You can only copy a rule base to an existing rule base. If the destination rule base does not exist, you must first create it. See “Creating a new rule base” on page 23 for additional information.
2. If you do not choose the Remove files in destination before copying option, the rule set file is empty except for a label. It can be edited, compiled, and loaded as is without any problems. However, to see the source immediately after the copy, edit the rule base in the rule base editor and save the rule base. After saving, the rule set file is the same as the original, if no changes were made to the copy.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy a rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

To copy a rule base:

1. From the Event Server icon pop-up menu, select Rules Bases to display the Event Server Rule Bases window.
2. From the pop-up menu of the rule base that is to be copied, select Copy. The Copy Rule Base dialog is displayed.

3. From the Destination rule base scrolling list, select the destination rule base. You can copy the specified rule base to more than one rule base by selecting more than one destination rule base from the Destination rule base scrolling list.

4. To copy the rules from the specified rule base, select Copy rules.
5. To copy the classes from the specified rule base, select Copy classes.
6. To copy files that have the same name as files currently in a destination rule base, select Overwrite files that exist in destination.
7. To remove all existing files in the destination rule base prior to the copy, select Remove files in destination before copying.
8. To complete your option selections and copy the source rule base to the target rule base, click Copy & Close. The specified rule base is copied to the specified destination rule base and then, the updated Event Server Rule Bases window is displayed.

Deleting a rule base: The rule base is only deleted from the Tivoli object database. The rule base directory structure on disk is not deleted.

Note: Do not delete the loaded or active rule base.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete a rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

To delete a rule base:
1. From the Event Server icon pop-up menu, select **Rules Bases** to display the Event Server Rule Bases window.

2. Select the rule base to delete.

3. From the Edit menu of the Event Server Rule Bases dialog, select **Delete**. A confirmation dialog is displayed.

4. Click **Yes** to delete the rule base.

**Importing a rule set into a rule base**: If you create a new rule set or modify a rule in a rule set that is part of the loaded rule base, you should import the rule set into the rule base.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import a rule set to a rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

To import a rule set to a rule base:

1. From the Event Server icon pop-up menu, select **Rules Bases** to display the Event Server Rule Bases window.

2. From the pop-up menu of the rule base, select **Import** to add the rule set. The **Import Into Rule Base** dialog is displayed.

3. Select **Import Rule Set**.

4. Type the absolute path name of the rule set to add in the Directory Path text field or click **File** to use the File Browser dialog. You can optionally specify the host where the rule set is located. The event server host is the default.
Note: The order of these files is important, as they affect rule engine performance. Placement of the files determines the evaluation order at run time.

5. From the Position to insert imported rule set scrolling list, select Insert Before or Insert After to specify the placement of the rule set being imported. Then, select a file.

6. To import the rule set to the rule base and have the Import Into Rule Base dialog remain displayed, click Import; otherwise, click Import & Close.

Importing BAROC files into a rule base: If you create a new BAROC file, modify a class definition in a BAROC file that is part of the loaded rule base, or install an event adapter with new event classes currently not known to the rule base, you must import the class definitions contained in BAROC files into the rule base.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import BAROC files into a rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

To import a BAROC file to a rule base:

1. To display the Event Server Rule Bases window, select Rule Bases from the Event Server icon pop-up menu.

2. From the pop-up menu of the rule base to import the class definitions, select Import. The Import Into Rule Base dialog is displayed.

3. Select Import Class Definitions.

4. Type the absolute path name of the BAROC file to add in the Directory Path text field or click File to use the File Browser dialog. You can optionally specify the host that contains the BAROC file. The event server host is the default.

5. To specify the placement of the BAROC file being imported, select either Insert Before or Insert After and then select a file from the Position to insert imported class file scrolling list.

   Note: The order of these files is important when you are defining new classes. Placement of the file determines the evaluation order at run time.

6. To import the BAROC file and have the Import Into Rule Base dialog remain displayed, click Import; otherwise, click Import & Close.
Compiling a rule base: A newly created rule base or a rule base that has been modified must be compiled before it can be used by a rule engine. A compiled rule base is a binary file.

Note: If you do not compile a rule base before attempting to load it, an error message is issued.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile a rule base</td>
<td>Event server</td>
<td>user</td>
</tr>
</tbody>
</table>

To compile a rule base:
1. From the appropriate rule base pop-up menu, select **Compile** to display the Compile Rule Base dialog.
2. If you plan to perform rule tracing on this rule base later, select **Trace Rules**. Selection of this option enables tracing directives in the rule set files for tracing the entire rule base.
3. Click **Compile**. The compiler output appears in the Compile Rule Base dialog.
4. Click **Close** to exit.

Loading and activating a rule base: The loaded rule base is a copy of a rule base placed in a known location to the event server. You can specify whether a rule base should be loaded and activated immediately, or if it should be loaded and then activated the next time the event server is restarted. See “Loaded and active rule bases” on page 31 for additional information.

The following table provides the context and Tivoli authorization role required for this procedure:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and activate a rule base</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

Note: You must stop and restart the event server if you modify, add, or delete any BAROC files.

To load a rule base and optionally activate it:
1. From the Event Server icon pop-up menu, select **Rules Bases** to display the Event Server Rule Bases window.

2. From the pop-up menu of the rule base, select **Load** to load. The Load Rule Base dialog is displayed.

3. To load the rule base and make it the active rule base immediately, select **Load and activate the rule base**.

   —OR—

   To load the rule base, but make it the active rule base when the event server is restarted, select **Load, but activate only when server restarts**.

4. Click **Load & Close**.

   Clicking **Close** closes the Load Rule Base window without performing any action, similar to a cancel operation.

**Loaded and active rule bases**

The active rule base is the rule base in memory being used by a rule engine. The loaded rule base is a rule base stored on disk in a known location by the event server. The loaded rule base can be a different rule base than the active rule base. The following examples explain this concept in a little more detail:

- Suppose the loaded and active rule base is named Default. While the event server is running, you load (but do not activate) the rule base named New. The loaded rule base is now New and the active rule base is still Default. If you stop and restart the event server, the New rule base becomes both the loaded and active rule base. Starting the event server always activates the loaded rule base.

- Suppose the currently loaded and active rule base is again Default. While the event server is running, you load and make active the New rule base. The currently loaded and active rule base is now New. You do not have to stop and restart the event server to accomplish this task, unless you made modifications to any BAROC files used by the rule base. Modifications to BAROC files always necessitate stopping and restarting the event server.

Again, loading a rule base copies rule base directories and files to a specific location known by the event server. Stopping and restarting the event server, or activating a loaded rule base if no BAROC file modifications were done, causes the rule base in this known location on disk to be placed into memory for use by the
rule engine. Specifying the option to activate as well as load a rule base copies the files to the known location and automatically stops and restarts certain components of the event server so the rule base can be placed into memory and used.

When you create a rule base, you give it a meaningful name and specify where it is located on disk. The name is associated with the storage location so the name is used in subsequent references to the rule base.

**UNIX directory permissions for rule bases**

For rule bases created on UNIX operating systems, the directories are created with permission values of 755. You can override these default permissions with the TEC_UMASK environment variable.

1. Enter the following command to copy the Tivoli environment settings to a temporary file:
   
   `odadmin environ get > temp_file`

   Edit the file with a text editor and add the following line: The owner permissions cannot be altered with TEC_UMASK. Owner permissions always have a value of 7. You can alter the user and group permissions. For example, to set the permissions on a rule base directory structure to 750, perform the following steps:

   `TEC_UMASK=750`

2. Enter the following command to copy the Tivoli environment settings back:
   
   `odadmin environ set < temp_file`

3. Either kill the tec_config process or wait five minutes until the tec_config process ends.

**Testing, debugging, and analyzing rules**

The IBM Tivoli Enterprise Console product provides facilities to test, debug, and analyze rules.

For testing rules, commands that simulate events and an event simulator program are provided.

For debugging rules, a tracing facility is provided. The tracing facility lets you trace rules at different levels of granularity, such as trace all rules, a particular rule set or part of a rule set, or a particular rule.

For analyses of rules, a profiling function is provided. Profiling produces a report containing rule firing information. The report contains information such as:

- The amount of time (in seconds) spent by the rule to process the last event evaluated by the rule
- The number of events processed by the rule
- The amount of time (in seconds) all events spent in the rule
- The throughput of events for the rule, expressed as the number of events per second

Creating rules in international environments

This section contains information about how to create rules in non-English environments. For more information about international environments, see "BAROC files in international environments" on page 44 and "Graphical rule builder considerations for international environments" on page 243.

Using non-English text in rules

You can enter non-English text in rules for the following uses, but the text must be UTF-8 characters. The rule set file must be converted into UTF-8 data before compiling the rule base. You can use the wiconv command to convert a rule set file into UTF-8 data. See the Tivoli Management Framework Reference Manual for additional information about the wiconv command.

- In event attribute filters for attributes of type STRING. The UTF-8 characters must be enclosed by single quotation marks.
- In predicates that are in the action body of a rule. The UTF-8 characters must be enclosed by single quotation marks.
- In the name and description of a rule. The UTF-8 characters must be enclosed by single quotation marks.
- In comments. No single quotation marks are required.

You cannot use UTF-8 characters in language keywords, event class names, or attribute names.

Passing data to tasks and programs

The Tivoli Enterprise Console product passes data internally in UTF-8 encoding. If you need to convert UTF-8 data to local encoding before passing the data to a program, task, or script, the tec_exectask_dbcs variable is provided to accomplish this.

The tec_exectask_dbcs variable in the $BINDIR/TME/TEC/.tec_config file toggles local encoding translation on and off. The default value is false. To enable local encoding translation, stop the event server and change this line to the following:

tec_exectask_dbcs=true

After setting tec_exectask_dbcs=true in the .tec_config file, you must restart the event server. To pass data to Windows or a shell script, or Perl script for UNIX or Windows, the tec_exectask_dbcs variable should be set to true. UTF-8 data cannot be processed by Windows or by a shell or Perl script. UTF-8 is not supported on Perl versions prior to 5.6. If you want to use UTF-8 data with Perl scripts, you can install Perl 5.6 or later, which does support UTF-8.

Passing data from exec_program and exec_program_local predicates: On UNIX systems, the exec_program and exec_program_local rule language predicates call the version of Perl that is provided by the operating system, so local encoding can be used. Set the tec_exectask_dbcs variable to true.

Data for executable modules: An executable module can use UTF-8 or local encoding in a UNIX environment, so the tec_exectask_dbcs variable can be set to either true or false.

In a Windows environment, you must set the tec_exectask_dbcs variable to true and pass the data as arguments.

Data for exec_task and exec_task_local predicates: The exec_task and exec_task_local rule language predicates can only pass non-UTF-8 data, so the
The `tec_exectask_dbcs` variable must be set to `true`. If you need to use UTF-8 data, you should use the `exec_program` or `exec_program_local` predicates instead of the `exec_task` or `exec_task_local` predicates.

In a Windows environment, the data must be passed as arguments.

**UTF-8 data and local-encoded data compatibility:** The `tec_exectask_dbcs` variable is available in the Tivoli Enterprise Console, Version 3.7.1 and later, and in the Tivoli Enterprise Console, Version 3.7 with the 3.7.0-TEC-0004 patch installed.

The following table lists the compatibility of UTF-8 data and local encoding data for the different types of scripts and programs:

<table>
<thead>
<tr>
<th>Rule Language Predicate</th>
<th>Program Type</th>
<th>UTF-8 Encoded Data</th>
<th>Local Encoded Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec_program and exec_program_local</td>
<td>shell</td>
<td>Not supported1</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Perl 5.5.x or earlier</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Perl 5.6 or later</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>executable</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>exec_task and exec_task_local</td>
<td>shell</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Perl 5.5.x or earlier</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Perl 5.6 or later</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>executable</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Notes:

1. Passing UTF-8 data is supported in a Solaris environment.

2. You must pass the data as arguments in a Windows environment. You cannot pass the data in environment variables.
Chapter 2. Event class concepts

The different types of classes of events that an event server can receive are defined in Basic Recorder of Objects in C (BAROC) files, sometimes referred to as event class definition files. These classes are stored in a rule base. When an event is received by the event server, the event class is compared to those stored in the active rule base. If the event class exists in the rule base and the event attributes are valid, the event is accepted as valid by the event server. If the class does not exist in the rule base, or the attributes are not valid, the event is given a status of PARSING_FAILED in the reception log and the event is discarded. BAROC files for the adapters supplied by the IBM Tivoli Enterprise Console product are provided. For other adapters, you must install their BAROC files (or add the event class definitions to existing BAROC files) before the adapters are started. A rule base can contain multiple BAROC files.

Note: To see the BAROC file grammar in Backus Naur Format (BNF) notation, see Appendix B, “BNF grammar for BAROC files”, on page 445.

An event adapter determines the class of an event before sending it to an event server. As a native event is received at the source monitored by an adapter, the adapter builds an event instance of the event class that corresponds to the native event, based on event class information defined in a BAROC file. For example, assume that the following native event is written to a system logfile:

Mar 10 08:51:42 aspen su:'su root' failed for jsmith on /dev/ttyp0

The event adapter matches the format of this native event, assigns the event class and attribute values, and sends the following Tivoli-formatted event to the event server:

Su_Failure:source=LOGFILE;origin=aspen;
date="Mar 10 08:51:42";host=aspen;sub_source=SYSLOG;
from_user=jsmith;msg='su:su root failed for jsmith on /dev/ttyp0';to_user=root;END

Designing event classes

The creation of event classes is based on an analysis of the events in your environment. For more information about this process, see “Designing rules” on page 3. Once a high-level analysis of events in your environment is complete, event classes are generally developed as follows:

1. Examine your current IBM Tivoli Enterprise Console environment, if applicable. If you decide to retain some of the existing rules, new ones must be defined such that there are no conflicts. In this case, analyze the behavior and sequence of the existing rules to ensure that neither set of rules adversely affects the other. For example, one rule set should not close or discard an event that another rule set uses for correlation. The rule sets should be compared for name or attribute conflicts. Also, because rules do not typically fire on non-leaf classes, use caution when changing leaf classes to superclasses.

2. Determine how the hierarchy of BAROC classes should be structured. Some customers define an event class for each event severity, such as Temp_Warning, Temp_Critical, or Temp_Normal. Others define a single event, such as Temp_Event, and use an attribute, such as severity, for differentiation. The first method typically provides better performance when running rules, because
events can be identified quickly by class name alone rather than having to examine the severity attribute, which is slower. The second method requires many more event classes and might require more configuration of the event adapters for certain sources. Which configuration is appropriate depends on your environment.

3. Determine which classes should be superclasses. A superclass should be defined whenever there exists a group of events that are logically related. In such cases, defining a superclass gives structure to the classes and can also reduce coding, simplify maintenance and improve performance. However, superclasses should not be used merely to simplify or reduce code; they should be used only when the subclasses are logically related. For example, if one administrator happens to support both databases and printers, these events should not be defined under a single superclass (e.g., Gwyna_Events) in order to make it easier to configure the event console or to write rule filters.

4. Determine an event class naming convention. It is helpful to indicate the full inheritance line of each subclass by using a naming convention in which the names of an event’s superclasses are concatenated in order of increasing specificity.

5. Determine file management policies and file naming conventions. It is beneficial to create a separate BAROC file for each event source. This makes each file specific and self-contained, which helps prevent inconsistencies. This policy leads to many files, which necessitates a naming policy that indicates precisely which events are contained in the named file. This is best accomplished by concatenating the event source and event category to form the file name.

For example, the superclass for a generic event pertaining to a Cisco Cat 5000 router would be called Cisco_Cat5000_Event. This class definition would reside with the class definitions for autonomous Cisco events (which would be subclasses) in the file cisco_cat5000.baroc or cat5000_auto.baroc. The Cisco_Cat5000_Event event could itself be a subclass to a more generic Cisco_Event class.

If several environmental events from the cat5000 router are to be monitored, a superclass for environmental events, Cisco_Cat5000_Env_Event, would be defined as a subclass to the Cisco_Cat5000_Event and would reside in a file called cat5000_env.baroc. Specific environmental events such as Cisco_Cat5000_Env_Temp and Cisco_Cat5000_Env_Voltage would then be defined as subclasses to this Cisco_Cat5000_Env_Event; these definitions could reside in the same BAROC file cat5000_env.baroc.

6. With all of this information, create the event classes for your environment.

**Event class definitions**

Event class definitions define each possible event type that can be received at the event server. Each definition must have a unique name.

Event class definitions are generally structured as shown in the following example. Keyword syntax is in uppercase.

```
TEC_CLASS: class_name ISA super_class_name
    DEFINES {
        attribute_definitions;
    }
END
```

**Note:** BAROC file syntax is case sensitive.
An event class definition inherits some attribute definitions, can define new attribute definitions, or can replace the values of inherited attribute definitions.

Event classes are organized in a hierarchy. At the top of the hierarchy is the base event class named EVENT. The following figure shows an example of event class definitions for the Su_Success and Su_Failure event classes, taken from the tecad_logfile.baroc BAROC file. These classes inherit their attribute definitions from the Logfile_Su class, which in turn inherits its attribute definitions from the Logfile_Base class. The Logfile_Base class inherits its attribute definitions from the base event EVENT class (not shown). Inheritance is implemented with the ISA operator. See "Inheritance" on page 40 for additional information about event class hierarchy.

```plaintext
TEC_CLASS :
  Logfile_Base ISA EVENT
  DEFINES {
    source: default= "LOGFILE";
    sub_source: default= "LOGFILE";
    sub_origin: default= "N/A";
    adapter_host: default= "N/A";
    msg_catalog: default= "none";
    msg_index: default= 0;
    repeat_count: default= 0;
    pid: STRING, default="N/A";
    severity: default = WARNING;
  }
END

TEC_CLASS :
  Logfile_Su ISA Logfile_Base
  DEFINES {
    from_user: STRING, dup_detect = yes;
    to_user: STRING, dup_detect = yes;
    on_tty: STRING, dup_detect = yes;
    severity: default = WARNING;
  }
END

TEC_CLASS :
  Su_Success ISA Logfile_Su;
END

TEC_CLASS :
  Su_Failure ISA Logfile_Su;
END
```

**Event attribute definitions**

Event information is contained in attributes. Attributes are defined in an event class and receive values for specific instances of the event class. Attribute values can come from a default value specified in an event class definition, from an event adapter, or from the event server supplying the value. See "Event attribute values supplied by the event server" on page 7 for additional information. See the IBM Tivoli Enterprise Console Adapters Guide for a description of attribute values provided by event adapters.

Attributes are inherited from superclasses of the event class in which they are defined. Attributes can be redefined or specialized in these subclasses, except their data types cannot be changed.
For example, the severity attribute has a default value of WARNING in the BAROC file that defines the base event EVENT, but it can be redefined for a particular subclass of the base event. For example, the Su_Failure event class might redefine the value of the severity attribute to CRITICAL.

An attribute definition consists of an attribute name, type, and one or more attribute facets, in the following format:

\[attribute_name: type, facet1, \ldots;\]

**Attribute names**

Attribute names consist of a meaningful string.

The following example defines attribute names of name, address, employer, and hobbies for the Person event class:

```
TEC_CLASS:
 Person ISA EVENT
 DEFINES {
     name: STRING, dup_detect=YES;
     address: STRING, dup_detect=YES;
     employer: STRING;
     hobbies: STRING;
};
```

**Note:** Do not give an attribute the name of `object`. It will cause an error.

**Attribute data types**

The data type of an attribute can be:

- A simple type, such as INTEGER, REAL, STRING, INT32, ENUM. An ENUM (enumeration), is an arbitrarily chosen set of integer and string pairs. See "Enumeration definitions" on page 43 for additional information.
- A list of simple types, which is a LIST_OF data type.

The data type for an attribute must be specified when the attribute is defined. The following is the syntax for specifying a data type. The brackets mean that the particular specification is optional.

```
[complex_type] simple_type
```

The syntax is described as follows:

**complex_type**

Specifies the complexity. If not specified, the default value is SINGLE. Valid values are:

**LIST_OF**

A list of values of the simple type. A list is enclosed within brackets and entries are separated by a comma. The following example shows the syntax of a list with one element:

```
acl: LIST_OF STRING, default=[admin];
```

**SINGLE**

A single value.

**simple_type**

Specifies the base type. This specification is required. Valid values are:

**ENUM**

A value of an enumeration.
INT32 A 32-bit integer value.

INTEGER
A 29-bit integer value.

REAL A real value.

STRING
A string value of up to 255 characters.

Attribute facets
A facet is used to define additional information about an attribute. An attribute can be defined with any or all of the following facets:

**default**
Specifies a default value for the attribute. The default value is used when the attribute value is not explicitly assigned by an adapter. This facet is optional. The syntax is:

default=value

The default value must correspond to the attribute value’s type.

The following example shows that a default value of WARNING is set for the severity attribute. Notice that the data type is an enumeration, as described in “Enumeration definitions” on page 43.

severity: SEVERITY, default = WARNING;

**parse**
Specifies whether an event adapter can write a value to an attribute. It is analogous to a write-protect flag for an attribute. The default value is YES. The syntax is:

parse=value

Valid values are:

no An adapter cannot write a value to that attribute

yes An adapter can write a value to that attribute.

If an event is received by the event server that contains an attribute defined as parse=no and a value has been inserted for the attribute, the event is given a status of PARSING_FAILED in the reception log and it is discarded.

Typically, attributes defined with parse=no are reserved for the event server to provide their values. The following example shows that the date_reception attribute is reserved for the event server to provide a value indicating when the event was received:

date_reception: INT32, parse = no

To perform mathematical operations in rules with attributes of INT32 data type, which are pointers, see “pointeroffset” on page 416.

**dup_detect**
Defines the criteria for determining whether two events are the same; that is, they are duplicates of each other.
**Note:** Setting the dup_detect facet only provides a definition. You must create rules to test for duplicate events and specify the actions to take when they’re detected by rule processing.

Two events are considered duplicates if they have the same values for all attributes defined with the dup_detect facet set to yes and if they are of the same event class. For example, assume the following event class definition:

```
TEC_CLASS:
    Person ISA EVENT
    DEFINES {
        name: STRING, dup_detect=yes;
        city: STRING, dup_detect=yes;
        employer: STRING;
        hobbies: STRING;
    };
```

The following events are considered duplicates because the attribute values of their respective name and address attributes are the same (assuming both events are of the same event class):

- `<"Joe", "Lafayette", "ABC Widgets", "Computers">`
- `<"Joe", "Lafayette", "XYZ Widgets", "Ham Radio">`

By default, dup_detect is no.

### Inheritance

Event class definitions are organized in a hierarchy of superclasses and subclasses by using the ISA inheritance operator. *Inheritance* means a subclass inherits the attribute definitions of its superclass. IBM Tivoli Enterprise Console only supports single inheritance.

Attributes defined in a superclass are common to all of its subclasses. Attributes defined in a subclass are not visible to peer subclasses or any superclasses.

A subclass can override the attribute values and facets inherited from a superclass, but a subclass cannot change the data type of the attribute.

Information common to subclasses is usually located in a superclass to avoid duplication. Information unique to a subclass is located in the subclass.

A superclass must be defined before all of the subclasses that inherit from it.

Only events of subclasses at the lowest level of the class hierarchy trigger rules to run. These lowest-level event classes are known as *leaf classes* or *leaf-node* classes. The exceptions to this rule are:

- When the fire on non_leaf directive has been enabled (see “Directives” on page 69 for additional information).
- When class filters like those shown in the following examples are specified.

**Note:** As these two filters trigger a rule for every event, their use might be detrimental to rule engine performance.

```
event: _event of_class _class
event: _event of_class _
```
If an event is received that is a superclass, it can appear on an event console but no rules are triggered by it (providing the fire_on_non_leaf directive is not enabled). If a rule is written for a superclass event, any subclass of the superclass can trigger that rule to run.

The following figure illustrates a simple class hierarchy.

![Class Hierarchy Diagram]

Event class definitions are loaded in the order specified when the BAROC files were imported into the rule base, and in the order of their position within the BAROC files. See “Rule base manipulation procedures using the rule builder” on page 22 or the wrb command in the IBM Tivoli Enterprise Console Command and Task Reference for information about viewing the order of, importing, or deleting BAROC files.

At the top of the hierarchy is the base event, whose event class name is EVENT. The base event class definition contains all of the attributes that all other event classes inherit. The base event class is automatically imported into every rule base you create. The following figure shows the event class definition for the base event:

```tec_class
TEC_CLASS : EVENT
DEFINES {
  server_handle: INTEGER,
  parse = no;
  date_reception: INT32,
  parse = no;
  event_handle: INTEGER,
  parse = no;
  source: STRING;
  sub_source: STRING;
  origin: STRING;
  sub_origin: STRING;
  hostname: STRING;
  fqhostname: STRING;
  adapter_host: STRING;
  date: STRING;
  status: STATUS,
  default=OPEN;
  administrator: STRING,
  parse = no;
  acl: LIST_OF STRING,
  default = [admin],
  parse = no;
  credibility: INTEGER,
  default = 1,
  parse = no;
  severity: SEVERITY,
}
```
Inheritance example

The following example shows a fragment from the tecad_logfile.baroc file, which contains the event class definitions for the UNIX logfile event adapter. The following is an inheritance structure for this example:

EVENT

Logfile_Base
  Logfile_Su
    Logfile_Su_Success
    Logfile_Su_Failure

*****************************************************************************

TEC_CLASS : Logfile_Base ISA EVENT
DEFINES {
  source: default= "LOGFILE";
  sub_source: default= "LOGFILE";
  sub_origin: default= "N/A";
  adapter_host: default= "N/A";
  msg_catalog: default= "none";
  msg_index: default= 0;
  repeat_count: default= 0;
  pid: STRING, default= "N/A";
  severity: default = WARNING;
};

END

TEC_CLASS : Logfile_Su ISA Logfile_Base
DEFINES {
  from_user: STRING, dup_detect = yes;
  to_user: STRING, dup_detect = yes;
  on_tty: STRING, dup_detect = yes;
  severity: default = WARNING;
};

END

TEC_CLASS : Su_Success ISA Logfile_Su;
END

TEC_CLASS : Su_Failure ISA Logfile_Su;
END
Enumeration definitions

Enumerations define the data type and valid values for some attributes. They allow the transparent use of an integer or string value within a set of pairs. The following example shows the SEVERITY enumeration definition that is used for the severity attribute of an event class:

```
ENUMERATION SEVERITY
  10 UNKNOWN
  20 HARMLESS
  30 WARNING
  40 MINOR
  50 CRITICAL
  60 FATAL
END
```

Enumerations are defined in BAROC files and they must be defined before they are used. An enumeration name must be unique among all event classes and other enumerations in a rule base. Pre-defined enumerations are located in the root.baroc and tec.baroc files.

If a new status is added and has an enumeration value higher than that of the CLOSED value, then it is treated as a CLOSED event and will stay in the cache. If the new status has an enumeration value lower than that of the CLOSED value, then it will not be treated as a CLOSED event and will stay in the cache according to the parameters set for non-closed events.

Design recommendation for using enumerations instead of event class definitions

Creation of a large number of leaf classes is not recommended for any given root class. This can significantly reduce rule engine performance. If possible, fewer leaf classes in conjunction with ENUMERATION type event attributes should be used. ENUMERATION type comparisons are integer comparisons while event class comparisons are string comparisons. Since integer comparisons are more efficient, it’s often the better approach. This can also simplify event class design since fewer event classes are needed.

For example, assume there are 6000 distinct SNMP error types to manage. One approach is to create an event class for each error type of the form SNMP_Error_xxxx, where xxxx is the error number. A better approach is to create an ENUMERATION type definition, for example, SNMP_ERROR_ENUM, with 6000 indexes and corresponding error identifiers. Then, create an SNMP_Error event class definition that contains an error_type attribute of type SNMP_ERROR_ENUM.

Overview of the root.baroc and tec.baroc files

The root.baroc and tec.baroc BAROC files are provided and automatically imported into every rule base that gets created. The root.baroc file is imported first, followed by tec.baroc. These files contain enumeration definitions and event class definitions used by the event server.

Note: Do not change the loading order of these two files. Any BAROC files you import into a rule base must be imported after these two files.
Re-importing modified BAROC files

The following procedure describes how to re-import BAROC files that you’ve modified to change or create new class definitions, or define enumerations. You can use the `wrb` command or the rule builder for the rule base manipulation steps. The `wrb` command syntax is shown in the following steps. See “Rule base manipulation procedures using the rule builder” on page 22 for information about performing these steps with the rule builder.

1. Make modifications to the file with a text editor in a different directory than the rule base directory structure.
2. Delete the old file from the rule base.
   ```
wrb -delrbclass filename rule_base
   ```
3. Import the new file into the rule base. You can use the –before or –after option for the specific loading position of the class file if required. This example simply places the class file at the end of the loading sequence.
   ```
wrb -imprbclass filename.baroc rule_base
   ```
4. Compile the rule base. The new file is replicated to all rule base targets defined in the rule base.
   ```
wrb -comprules rule_base
   ```

BAROC files in international environments

You can enter non-English text in a BAROC file for the following uses, but the text must be UTF-8 characters. The BAROC file must be converted into UTF-8 data before compiling the rule base. You can use the `wiconv` command to convert a BAROC file into UTF-8 data. See the Tivoli Management Framework Reference Manual for additional information about the `wiconv` command.

- When assigning a default value to attributes of type STRING. The UTF-8 characters must be enclosed by single quotation marks.
- In comments. No single quotation marks are required.

You cannot use UTF-8 characters in language keywords, event class names, attribute names, enumeration names, and enumeration values. For more information about UTF-8 characters and international environments, see “Creating rules in international environments” on page 33.
Chapter 3. Rule engine concepts

The rule engine is a rule-based event processor. It is driven by the arrival of a new event or an internally generated event, a change request or redo request for a previously received event, or the expiration of a timer associated with a previously received event.

When the event server is started, it activates a rule base into memory for rule engine use. The rule base contains all rules and event class definitions that are to be evaluated against events.

Rules are run by the rule engine. When the rule engine processes an event, a separate transaction begins and the rules are evaluated against the event. If the event satisfies the criteria for triggering rules, the rule engine controls the processing of those rules.

Multiple transactions originating from an event console can be grouped internally into a single transaction. For example, the multiple transactions that can result from an administrator closing or acknowledging several events at the same time from an event console would be grouped into one transaction for rule engine processing.

Forced cleaning of the event cache

When the internal event TEC_Notice event “Rule Cache full: forced cleaning” occurs, five percent of the events are removed from the cache. Events are removed in order by age, with the oldest events removed first.

Note: The event cache might have different contents than the event repository or an event console. This is because it is primed with events from the event repository when the IBM Tivoli Enterprise Console server is started. If this TEC_Notice event is received, the solution is either to increase the size of the event cache or to reduce the time that events are kept in the event cache. For more information about setting these parameters, see the description of the wsetesvrcfg command in the IBM Tivoli Enterprise Console Reference Manual. These parameters can also be configured through the Tivoli Desktop using the Event Server Parameters dialog.

Processing events

To process an event, the rule engine compares the event against the applicable rules in the active rule base. When the event matches the event filter criteria for a rule, the rule is triggered. Rules are run one at a time, based on the order of their placement in the rule base, which is basically the rule order within each rule set and within rule-set order. Note that there are predicates that can alter the regular flow of control for rule evaluation, for example, the commit_action, commit_rule, and commit_set predicates alter the flow of control.

Events are stored in the event database after all applicable rules have been run for the event, except in the case where a rule contains an action to drop an event.

The rule engine processes events in the following order:
1. Incoming new events, for example, those received from adapters, other products, or a `wpostmsg` or `postmsg` command.

2. Internally generated events, for example, those created by the `generate_event` predicate.
   The rule engine queues internally generated events and then processes them first after it has finished processing the current transaction. Internally generated events are processed in first-in, first-out (FIFO) order. The internally generated event and all modifications resulting from that event are included in a transaction.

3. Internal change requests, for example, those created by the `change_event_administrator`, `change_event_severity`, or `change_event_status` predicates.
   The rule engine queues internal change requests and then processes them in FIFO order after the current transaction and all internally generated events have been processed.

4. Redo requests, for example, those created from the `redo_analysis` predicate.
   The rule engine queues redo requests and then processes them in FIFO order after the current transaction completes, all internally generated events have been processed, and all internal change requests have been processed. A redo request directs the rule engine to perform a new pass through the rules for a previously received event.

5. External change requests, for example, those caused by event adapters or administrator action from an event console.
   The rule engine queues external change requests and then process them in FIFO order after the current transaction completes, all internally generated events have been processed, all internal change requests have been processed, and all redo requests have been processed. When a change request is processed, only change rules are evaluated. The change request is applied to the event only after all change rules have been evaluated.
**Processing timer expiration**

The rule engine evaluates timer rules for an event, based on the expiration of a timer that was set on the event with the `set_timer` predicate. When the timer expires, the event is evaluated against all timer rules. For example, you can create a timer rule that specifies that all `Su_Success` events close one hour after reception, and then set a timer on all `Su_Success` events as they are received.

**Processing the next transaction**

Internally generated events, internal change requests, and redo requests are processed within the transaction for the event in which they were called. Each event transaction is completed before the next event transaction is processed; that is, a new event is not processed by the rule engine until all processing is complete for the current event.

**Performance tips**

The following tips are offered to improve rule engine performance:

- Use the `commit_action`, `commit_rule`, and `commit_set` predicates to alter the regular flow of control to avoid unnecessary rule processing. Using these predicates requires that you logically structure your rule sets.
- When performing a redo request, thoroughly analyze which actions have to be run for new events, for redo requests, or for both. Use different types of actions as appropriate.
- Order attribute conditions of `where` clauses in event filters so all attribute tests for values are located before simple instantiation of variables with attribute values.
Chapter 4. Rule language reference

This chapter provides detailed descriptions and syntax for items that comprise the rule language. It is intended for users who are familiar with the rule concepts and need additional details about the rule language.

Naming conventions

The following table describes naming conventions for rule-related objects.

<table>
<thead>
<tr>
<th>Object</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAROC file</td>
<td>Any file name supported by the operating system on which the event server is running.</td>
</tr>
<tr>
<td>Event class</td>
<td>Any meaningful string, using uppercase, lowercase, or mixed characters. Valid characters for names are underscore (_), a–z (all cases), and 0–9.</td>
</tr>
<tr>
<td>Rule base</td>
<td>Any meaningful string, using uppercase, lowercase, or mixed characters. Rule base names are displayed as labels for icons in dialogs so keep them short but meaningful. Valid characters for names are underscore (_), a–z (all cases), and 0–9. Directory paths to rule bases cannot contain any blank characters.</td>
</tr>
<tr>
<td>Rule set</td>
<td>The physical file name of a rule set must be in the form of filename.rls. When referring to a rule set that has already been imported into a rule base, the .rls file name extension is implied and not entered. Also, do not use a period mark in a rule set name.</td>
</tr>
<tr>
<td>Rule pack</td>
<td>A rule pack name must be in the form of filename.rpk.</td>
</tr>
<tr>
<td>Rule name</td>
<td>Rule names can begin with a lowercase or uppercase letter; however, a rule name that begins with an uppercase letter must be enclosed within single quotation marks. Each rule within a rule set must have a unique name. Rule names should only contain alphabetic and numeric characters, except an underscore character is permitted, so long as it is not the first character of the name. Although rule names are not required, you should specify them with meaningful names to make rule traces easier to analyze.</td>
</tr>
<tr>
<td>Rule action</td>
<td>Rule action names must begin with a lowercase letter and can optionally be enclosed within single quotation marks. Each action within a rule must have a unique name. Rule action names should only contain alphabetic and numeric characters, except an underscore character is permitted, so long as it is not the first character of the name. Although rule action names are not required, you should specify them with meaningful names to make rule traces easier to analyze.</td>
</tr>
<tr>
<td>Variables</td>
<td>See &quot;Variables&quot; on page 50</td>
</tr>
</tbody>
</table>
Variables

Variables in rules are usually identified by a leading underscore (_), although they can also begin with an uppercase letter instead of an underscore. The recommended convention is to use the form with a leading underscore, as it makes rules easier to understand.

No declaration of variables is required. If a variable is not instantiated, it is instantiated with a value in the first call where it is used.

Note: When writing rules, pay special attention to the use of variables. Using a variable that has not been instantiated with a value can cause the rule not to work as intended in certain circumstances. For example, in an event filter where the variable _no_value is not instantiated, the following attribute condition would always succeed and instantiate the variables _origin and _no_value to the value of the origin attribute:

where [origin:_origin equals _no_value]

Variables are stored in memory. When the event server shuts down, they are lost. To store variables so they can be reloaded when the event server restarts, write them to a file before the event server shuts down, and then read them back in after the event server is restarted. Prolog predicates such as read and write can be used in rules to perform these actions. See "read" on page 421 and "write" on page 441 for additional information.

If you do not want strings that begin with an uppercase letter to be interpreted as variables, you must delimit them with single quotation marks. For example, Level_1 would be interpreted as a variable, whereas 'Level_1' would be interpreted as a string.

To illustrate the importance of variable syntax, the following rules perform exactly the same function. If the intent of the second rule is to filter on TEC_Start events only, the TEC_Start class specification should be delimited with single quotation marks in the event filter.

rule: unquoted_single_class1: (  
  event: _event of_class _class  
% The name of the event class is stored in variable  
% _class.  
  where [  
    msg: equals 'unquoted event filter 1'  
  ],  
  action: (  
    set_event_message(_event,'msg changed to %s',  
      _class)  
  )  
).

rule: unquoted_single_class2: (  
  event: _event of_class TEC_Start  
% The name of the event class is stored in variable  
% TEC_Start. To filter on event class TEC_Start only,  
% specify 'TEC_Start' (within quotes)  
  where [  
    msg: equals 'unquoted class filter 2'  
  ],
Regular variables

Regular variables provide a way of keeping information that can apply to a single rule or to the actions within a rule. The scope of a regular variable is as follows:

- Variables in the initial event filter of a rule have a scope of the rest of the rule in which the variable was instantiated. These variables can be used to instantiate the event class name or attribute values of the event under analysis for use later in the rule.
- Variables in an event filter of a predicate in a rule action have a scope that is limited to the current rule action, which means they can be accessed only by other predicates that are called within the same action.

The naming convention for variables used to hold the value of an attribute is typically the attribute name prefixed with an underscore character, as shown in the following example in the where clause:

```
rule:
  maintenance_started:
    (event: _event of_class 'TEC_Maintenance'
      where [origin: _origin])
```

Global variables

Global variables provide a way of keeping information that can apply to multiple events in the memory of the rules engine, also referred to as the knowledge base. You might want to use global variables to keep track of the state of a particular object (for example, a host) or to keep counters that apply across multiple events.

The structure of a global variable consists of a group key and key, which are used together to identify the data being stored. A group key is also used to perform operations on all members of a group; for example, initializing all members to a value. A group key is typically a literal value. A key is usually dynamic and assigned its identity based on the attributes of an event. Both a group key and key can be arbitrary strings. The group key can be thought of as the name of an array and the key as an index into the array.

Multi-dimensional global variables can be defined by passing in a key that is derived from each dimension. For example, you can have a two-dimensional array using the origin and sub_origin attributes. Construct the key using the atomconcat and set_global_var predicates as shown in the following example:

```
atomconcat([_origin, _sub_origin], _key),
set_global_var('My group key', _key, 'My value')
```

The stored values of global variables can be of any type, such as integers, strings, lists, and so forth. To indicate the stored value is a list, enclose the list members within brackets ([ ]). Whenever you retrieve the list members data into regular variables, enclose the regular variables in brackets as shown in the following example:

```
set_global_var('My group key', _key, ['a', 'b', 'c']),
% Assign global variable values.
get_global_var('My group key', _key, [ _var1, _var2, _var3],
```
Manipulating global variables

There are several predicates available to manipulate global variables. In the following descriptions, the [i] and [o] notations following an argument indicate whether the argument is an input or output—meaning whether the argument value is provided by you to use with the predicate or the value is returned by the predicate call, respectively.

The following example unifies the global variable value with the regular variable _into. If the global variable was not set, it is initialized to the value of _init and _into is set to that value.

```prolog
get_global_var(_groupkey[i], _key[i], _into[o], _init[i])
```

The following example sets the global variable to the value of _to. The _groupkey and _key arguments must be specified.

```prolog
set_global_var(_groupkey[i], _key[i], _to[i])
```

The following example resets all global variables that belong to the group to the value of _to.

```prolog
reset_global_grp(_groupkey[i], _to[i])
```

The following example unifies the key and global variable values with the _key and _into regular variables for all global variables of the group in succession. The _groupkey argument must be specified. This predicate can be used to iterate over the members of a group. Note that _key is an output which can then be passed into the set_global_var predicate to change the value, if preferred.

```prolog
get_global_grp(_groupkey[i], _key[o], _into[o])
```

Constants

String constants in the rule language are literal strings within single quotation marks. For example, 'WARNING', 'TEC_Error', and 'on' are all string constants.

Numeric constants do not require single quotation marks. For example, 9, 5.4, and 0xFF1953 are all numeric constants.

Comments

There are two forms of comment delimiters that can be used in rules. Text embedded within the /* and */ delimiters is treated as a comment and ignored by the compiler. You can create comments that span multiple lines using these delimiters. The other comment delimiter is the percentage symbol (%). All text after this character until the end of the single line is a comment. You cannot nest the first form of comments, as it will cause a compilation error. You can use a % within /* */ delimiters, but it is treated as literal text and not a comment delimiter.

In BAROC files, the single line comment delimiter is the number sign (#).
Rule structure

This section describes the general rule structure, provides usage notes, and shows examples for the following rule types. Explanations of each rule type can be found in "Rule types" on page 10:

- Plain rule
- Change rule
- Timer rule
- Simple rule
- Correlation rule

Rule parser grammar

An HTML file containing the grammar for the rule parser in Backus Naur Form (BNF) notation is located on the event server at the following location:

$BINDIR/../generic_unix/TME/TEC/BOOKS/HTML/rulep.html

and on the product CD at the following location:

/BOOKS/HTML/rulep.htm

The grammar provides complete structure and syntax for rule language statements.

General structure of a plain rule

The following example shows the general structure of a plain rule:

```
rule: rule_name: ( 
  description: 'description_text',
  directive: directives,
  event: event_filter 
    where [ 
      attribute_conditions 
    ],
  action: action_name: ( 
    action_body 
  )).
```

Usage notes

- Event filters are described in "Event filters" on page 60

Examples

1. The following plain rule example correlates the clearing of printer problem events with a Printer_Error_Cleared event.

```
rule: print_reset :
  
  event: _event of_class 'Printer_Error_Cleared'
    where [
      status: equals 'OPEN',
      hostname: _hostname
    ],

  reception_action: ( 
    all_instances(event: _prt_ev of_class 
      within ['Printer_Paper_Out',
        'Printer_Toner_Low',
        'Printer_Offline',
        ...]
    )
  ).
```
'Printer_Output_Full',
'Printer_Paper_Jam',
'Printer_Door_Open'
]

where [hostname: equals _hostname,
       status: outside ['CLOSED']
],
_event - 3600 - 3600 ),
change_event_status( _prt_ev, 'CLOSED' ),
drop_received_event
).

2. The following plain rule example sets a timer for 90 seconds on a printer problem event. If the timer expires and the problem is not fixed, the timer rules (including the one shown in example 1 of "General structure of a timer rule" on page 55) are evaluated.
rule: print_assist : (event: _event of_class
   within ['Printer_Paper_Out',
            'Printer_Toner_Low',
            'Printer_Offline',
            'Printer_Output_Full',
            'Printer_Paper_Jam',
            'Printer_Door_Open'
   ]
]

where [
       status: _status equals 'OPEN' ,
       hostname: _hostname,
       msg: _msg
     ],
reception_action:
   set_timer(_event, 90, '')
).

**General structure of a change rule**

The following example shows the general structure of a change rule:
change_rule: rule_name: (description: 'description_text',
   directive: directives,
   event: event_filter,
   where [attribute_conditions ],
   sender: sender_filter,
   attribute: attribute_change_filter,
   action: action_name: (action_body)
).


Usage notes

- Creating a change rule requires knowledge of the values that attributes can be changed to.
- Change rules are only evaluated when a change request has been issued.
- Event filters are described in “Event filters” on page 60.
- Sender filters and attribute change filters are described in “Change rule filters” on page 64.

Example

The following change rule example sends a mail message to tec_print when an error has been cleared on the printer. This message is only sent when a preceding message indicating a printer problem was issued (the num_actions attribute equals 1).

```plaintext
change_rule: print_chg_assist : (  
  event: _event of_class  
  within ['Printer_Paper_Out',  
    'Printer_Toner_Low',  
    'Printer_Offline',  
    'Printer_Output_Full',  
    'Printer_Paper_Jam',  
    'Printer_Error_Cleared',  
    'Printer_Door_Open'  
  ]  
  where [  
    hostname: _hostname,  
    status: _status outside ['CLOSED'],  
    num_actions: equals 1,  
    msg: _msg  
  ],  
  attribute: status set_to _new_status within ['CLOSED'],  
  action: (  
    exec_program(_event,  
      'scripts/TEC_Send_Mail.sh',  
      "$TEC - ½s: ½s" tec_print "The following condition for printer \  
½s has been fixed:
\n½s",  
      [_new_status, _msg, _hostname, _msg],  
      'YES')  
  ).
```

General structure of a timer rule

The following example shows the general structure of a timer rule:

```plaintext
timer_rule: rule_name: (  
  description: 'description_text',  
  directive: directives,  
  event: event_filter,  
  where [  
    attribute_conditions  
  ],  
  timer_duration: timer_duration_filter,  
  timer_info: timer_info_filter,
```
Usage notes

- When a timer that is set in a rule on an event expires, all timer rules are evaluated.
- Event filters are described in "Event filters" on page 60.
- Timer rule filters are described in "Timer rule filters" on page 66.
- Timers are set with the set_timer predicate, which is described on page 212.
- A plain rule example that sets a timer is shown in example 2 on page 54.
- A timer rule example that sets a timer is shown in example 2.
- An event of class TEC_Tick always exists in the event cache; that is, it is never aged out of the cache. You can search for this event in the cache and use it to start a timer, knowing that it is always there.
- When setting a timer, you can use the optional timer_info parameter to specify the purpose of a particular timer; the timer rule should then act only if the timer_info of the timer is set to the appropriate value. Using timer_info, you can handle each timer separately, which can help to avoid performance problems.

Examples

1. The following timer rule example sends a mail message to tec_print for assistance with a printing problem, if the problem has persisted past the expiration of a timer set on one of the events shown in the list of the event filter. The expiration of the timer set in example 2 in "General structure of a plain rule" on page 53 causes this rule to be evaluated.

   ```
   timer_rule: timer_print_assist : { 
      event: _event of_class 
      within ['Printer_Paper_Out',
              'Printer_Toner_Low',
              'Printer_Offline',
              'Printer_Output_Full',
              'Printer_Paper_Jam',
              'Printer_Door_Open' 
      ]
      where [ 
         status: _status equals 'OPEN',
         hostname: _hostname,
         msg: _msg
      ],
      action: ( 
         exec_program(_event,
           'scripts/TEC_Send_Mail.sh',
           '"T/EC - %s: %s" tec_print "The printer %s has the 
            following condition:
            %s",
           [status, msg, hostname, msg],
           'YES')
      )
   })
   ```

2. The following timer rule example raises the severity of an event to FATAL that has had a Level 1 timer expire and has not yet been acknowledged or closed. A Level 2 timer is set as part of this rule.

   ```
   timer_rule: 'upgrade_level1': { 
      event: _event of_class 'universal_host'
   })
   ```
where [  
  status: outside ['CLOSED','ACK']  
],

timer_info: equals 'Level 1',

action: raise_sev: (  
  set_event_severity(_event, 'FATAL'),  
  set_timer(_event, 90, 'Level 2')  
).

**General structure of a simple rule**

The following example shows the general structure of a simple rule:

```
simple_rule: rule_name:  
description: 'description_text'  
directive: directives  
event: event_filter  
  where [  
    attribute_conditions  
  ]  
  when:  
    event_conditions:  
      event_actions
```

**Usage notes**

- The `_event` and `_class` variables, and the `of_class` operator typically found in an event filter can not be used for a simple rule. These items are processed internally for a simple rule.
- The following event conditions are the only ones that can be used; that is, the rule parser recognizes them and no others:
  - `n` minutes_after_event_reception (`n` is an integer)
  - `event_acknowledged`
  - `event_closed`
  - `event_received`
  - `frequency_exceeds m within n minutes (`m` and `n` are integers)
  - `severity_downgraded`
  - `severity_upgraded`
- The event actions in the following table are the only ones that can be used; that is, the rule parser recognizes them and no others.

<table>
<thead>
<tr>
<th>Event Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>change_event_severity</td>
<td>Specifies a new value for the severity attribute. Severity can be UNKNOWN, HARMLESS, WARNING, MINOR, CRITICAL, and FATAL. Syntax: <code>change_event_severity(new_severity)</code></td>
</tr>
<tr>
<td>change_event_status</td>
<td>Specifies a new value for the status attribute. Status can be OPEN, RESPONSE, ACK, or CLOSED. Syntax: <code>change_event_status(new_status)</code></td>
</tr>
<tr>
<td>drop_duplicate_event</td>
<td>Checks for the existence of a duplicate event in the event cache having a status other than CLOSED within a time window (in whole minutes). If one exists, the <code>repeat_count</code> attribute of the existing event is incremented by one and the newly received event is dropped. Syntax: <code>drop_duplicate_event(time_window)</code></td>
</tr>
<tr>
<td>Event Action</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>exec_program</td>
<td>Runs a system command, shell script, or other program. The default search path is $BINDIR/TME/TEC. The commands, scripts, or programs are run on the same node as the event server. See &quot;exec_program&quot; on page 142 for additional information about the arguments for this event action. The arguments for the exec_program rule language predicate are similar. <strong>Note:</strong> For the exec_program event action in a simple rule, only event attributes are valid values for the _arg_list argument, unlike the exec_program rule language predicate for which you can specify any values for the _arg_list argument. Syntax: exec_program(prog_name, _format_string, _arg_list, watch_status)</td>
</tr>
<tr>
<td>exec_task</td>
<td>Runs a task from the specified task library. By default, tasks are run on the same node as the event server. Proper access to a task library is necessary to use this action. See &quot;exec_task&quot; on page 147 for additional information about the arguments for this event action. The arguments for the exec_task rule language predicate are similar. <strong>Note:</strong> For the exec_task event action in a simple rule, only event attributes are valid values for the _arg_list argument, unlike the exec_task rule language predicate for which you can specify any values for the _arg_list argument. Syntax: exec_task(task_name, format_string, _arg_list, watch_status)</td>
</tr>
<tr>
<td>forward_event</td>
<td>Sends an event to a different event server. There must be a ServerLocation option specified in the tec_forward.conf file in the TEC_RULES subdirectory of the rule base. See the description of &quot;forward_event&quot; on page 160 for additional information about the tec_forward.conf file. Syntax: forward_event()</td>
</tr>
<tr>
<td>set_event_message</td>
<td>Specifies the text for the msg attribute in the event. This can be an informational message or it can contain the value of another attribute. See &quot;set_event_message&quot; on page 206 for additional information about the arguments for this event action. The arguments for the set_event_message rule language predicate are similar. <strong>Note:</strong> For the set_event_message event action in a simple rule, only event attribute names are valid values for the _value argument, unlike the set_event_message rule language predicate for which you can specify any values for the _value argument. Syntax: set_event_message(_format, [_value])</td>
</tr>
</tbody>
</table>

- For a simple rule, names of event classes and attributes do not require single quotation-mark delimiters unless they have embedded spaces.
- Event filters are described in "Event filters" on page 60

**Example**

The following example simple rule changes an Su_Failure event to a severity of FATAL if an su command (switch user) to root fails three times within five minutes.

```
simple_rule: escalate_su_failure:

    description: 'Escalate more than 3 su root failures in 5 minutes.'

    event: Su_Failure
    where [ to_user: equals 'root' ]

    when: frequency_exceeds 3 within 5 minutes:
        change_event_severity('FATAL').
```

**General structure of a correlation rule**

The following example shows the general structure of a correlation rule:
correlation_rule: rule_name:
  description: 'description_text'

directive: directives

event_relation: event_class1 operator event_class2
  within: integer minutes

  when: [
    event_class1.attribute1 equals
    event_class2.attribute2
  ]

Usage notes
• The operators for the event_relation clause are as follows:
  cancelled by
    The reception of the event specified on the right of the cancelled by
    operator closes the event specified on the left.
  cancels
    The reception of the event specified on the left of the cancels operator
    closes the event specified on the right.
  caused by
    Links the event on the left (the effect event) of the caused by operator to
    the event on the right (the cause event). The values of the date_reception
    and event_handle attributes from the cause event are written to the
    cause_date_reception and cause_event_handle attributes of the effect
    event. The value of the status attribute of the cause event is written to
    the status attribute of the effect event. Any changes to the status of the
    cause event are automatically propagated to the status attribute of the
    effect event.
  causes
    The event on the right (the effect event) to the event on the left
    (the cause event). The values of the date_reception and event_handle
    attributes from the cause event are written to the cause_date_reception
    and cause_event_handle attributes of the effect event. Any change to the
    status of either event is automatically propagated to the status attribute
    of the other event.
• The conditions in the when clause are of the form event_class_name.attribute,
  where attribute is a valid attribute for the event_class_name event class.
• The equals operator is the only valid operator for a correlation rule.
• For a correlation rule, names of event classes and attributes do not require single
  quotation mark delimiters unless they have embedded spaces.

Example
The following two correlation rules examples can be used together to set up a
simple correlation event sequence that links a fan failure event to a temperature
warning event, and a temperature warning event to a temperature shutdown event.
Filters define the criteria that must be satisfied by an event before a rule is run. If the evaluation of a filter fails, the rule is skipped. The properties tested in a filter must be those defined for the class of the event. There are three types of filters:

- Event filter
- Change rule filter
- Timer rule filter

Event filters

An event filter tests event class names and optionally tests attribute values of the event under analysis. If the event passes the event filter criteria, the rule is run. Event filters can also be used in rule actions to test for related events in the event cache, in which case if the event in the cache is found, subsequent predicates within the action are run.

You can use variables in an event filter that defines the conditions that must be met for triggering a rule. These variables can be used to instantiate the event class name or attribute values of the event under analysis for use later in the rule. The scope of a variable instantiated in this event filter is the rest of the rule in which the variable is instantiated.

You can also use variables in an event filter that is used with a predicate in a rule action to accomplish the same goal for the referenced event in the predicate. These variables have a scope that is limited to the current rule action, which means they can be accessed only by other predicates that are called within the same action.

An event filter usually begins with a variable that is instantiated with a pointer to the event under analysis. For this variable, the recommended naming convention is _event, although you can use any variable name. The event pointer variable is
required for event filters of all rule types except for simple and correlation rules. You cannot specify it in simple and correlation rules.

The of_class operator follows the _event variable and is required for event filters of all rule types except simple or correlation rules. You cannot specify it in simple or correlation rules. The of_class operator signifies that an event class specification follows.

The event class specification tests the class of the event under analysis to see if the rule applies to that event class. The following list describes variations of a class specification and shows examples of each:

- The _ variable (a single underscore character), also referred to as the anonymous variable, is instantiated with the name of the event class of the event under analysis. This variable means that you won’t be retrieving the name of the event class for use later in the rule. The following example event filter succeeds for every event of every event class and instantiates the name of the event class to the anonymous variable:
  
  
  event: _event of_class _

- _class The _class variable is instantiated with the name of the event class of the event under analysis. The following example event filter succeeds for every event of every event class and instantiates the _class variable with the name of the event class:
  
  event: _event of_class _class

- **class_name**
  
  This is the explicit name of an event class. The following filter succeeds if the event under analysis is of the Printer_Error_Cleared event class.
  
  event: _event of_class 'Printer_Error_Cleared'

**Note:** If **class_name** is a superclass of the event under analysis, the class specification part of the filter succeeds for all events that are subclasses of **class_name**. If the fire_on_non_leaf compiler directive is enabled, the class specification part of the filter also succeeds if **class_name** is a superclass (also referred to as a non-leaf node).

- **class within [class_name1,...]**
  
  This class specification tests for one or more event classes in a list. If the event under analysis is a member of the list, the class specification part of the filter succeeds.

  If any of the event class names in the list is a superclass, the class specification part of the filter succeeds for all events that are subclasses of the superclass.

  If the fire_on_non_leaf directive is enabled, the class specification part of the filter also succeeds if event class is a superclass. The list can either be specified as a variable that has been previously defined, or an explicit list of class names. For lists, the brackets are required and event class names must be separated by a comma. The following filter succeeds if the event under analysis is one of those within the list and is an example of specifying an explicit list within the rule:

  event: _event of_class
  within ['Printer_Paper_Out',
          'Printer_Toner_Low',
          'Printer_Offline',
          'Printer_Error_Cleared',
          'Printer_Error_Filled']
Note: You can optionally use the _class variable to capture the event class name for use later in the rule. For example, an event filter can look like the following example. If the event under analysis is one of those in the list, the _class variable is instantiated with the event class name.

```java
event: _event of_class _class
within ['Printer_Paper_Out',
  'Printer_Toner_Low',
  'Printer_Offline',
  'Printer_Output_Full',
  'Printer_Paper_Jam',
  'Printer_Door_Open']
```

The following example shows the use of a variable in place of a literal LIST_OF value:

;Class file
TEC_CLASS:
  Logfile_Base ISA EVENT
  DEFINES {
    alist: LIST_OF STRING, default=['Printer_Paper_Out',
                                   'Printer_Toner_Low',
                                   'Printer_Offline'];
  }
END

;Rules file
print_reset:
  ( event: _event of_class 'Printer_Error_Cleared'
    where [ status: equals 'OPEN',
            alist: _aclasslist ],
    reception_action:
      ( first_instance( event: _prt_ev of_class
                        within _aclasslist
                        where [ status: outside ['CLOSED'] ],
                        event - 3600 - 3600),
        change_event_status(_prt_ev, 'CLOSED') )
  ).

_class outside [class_name1, ...]
This class specification tests for one or more event classes in the form of a list. The brackets are required and event class names must be separated by a comma. If the event under analysis is a member of the list, the class specification part of the filter does not succeed. If any of the event class names in the list is a superclass, the class specification part of the filter does not succeed for all events that are subclasses of the superclass. If the fire_on_non_leaf compiler directive is enabled, the class specification part of the filter also does not succeed if the event class is a superclass. The following filter does not succeed if the event under analysis is one of those within the list:

```java
event: _event of_class
  outside ['Printer_Paper_Out',
    'Printer_Toner_Low',
    'Printer_Offline',
    'Printer_Paper_Jam',
    'Printer_Door_Open']
```

This class specification tests for one or more event classes in the form of a list. The brackets are required and event class names must be separated by a comma. If the event under analysis is a member of the list, the class specification part of the filter does not succeed. If any of the event class names in the list is a superclass, the class specification part of the filter does not succeed for all events that are subclasses of the superclass. If the fire_on_non_leaf compiler directive is enabled, the class specification part of the filter also does not succeed if the event class is a superclass. The following filter does not succeed if the event under analysis is one of those within the list:

```java
event: _event of_class
  outside ['Printer_Paper_Out',
    'Printer_Toner_Low',
    'Printer_Offline',
    'Printer_Paper_Jam',
    'Printer_Door_Open']
```
Note: You can optionally use the _class variable as described in the _class within specification.

Attribute conditions: You can optionally further restrict the criteria of an event filter with attribute conditions. Attribute conditions are the where clause of an event filter. Attribute conditions let you optionally specify tests that must be satisfied by certain attribute values of the event under analysis. You can also instantiate variables with attribute values in the attribute conditions section. This lets you use those variables later in the rule—similar to the _class variable in an event filter.

The event under analysis must have passed the class specification part of the event filter first, before the attribute conditions are tested. If the event passes both the class specification and attribute conditions of the event filter, the rule is run.

Attribute conditions can take the following general forms. The where clause begins with the where keyword and an open bracket. It ends with a closed bracket. The entire event filter, including the where clause, ends with a comma. If you don’t specify any attribute conditions, you must still create an empty where clause.

- Test an attribute’s value. The following operators are valid for testing an attribute’s value:

  comparison_operator
  
  Compares an attribute’s value to a certain value. Valid operators are:
  
  - equals
  - greater_than
  - greater_or_equals
  - smaller_than
  - smaller_or_equals
  - within
  - outside

  The within operator tests whether an attribute value is equal to one of the values in a list, while the outside operator tests whether an attribute value is equal to a value not in the list.

  The following example tests whether the value of the status attribute is OPEN. OPEN is delimited by single quotation marks because it begins with an uppercase letter and would be interpreted by the rule compiler as a variable if it was not enclosed in single quotation marks.

where [status: equals 'OPEN'],

The following example tests whether the value of the status attribute is either OPEN or ACK and instantiates the _status variable with the value:

where [status: _status within ['OPEN','ACK']],

The following example tests whether the value of the status attribute is anything other than CLOSED:

where [status: outside ['CLOSED']]
**Note:** Ensure that you use the appropriate operator for the data type to test.

- Simple instantiation of a variable with an attribute’s value. The following example simply instantiates the _status variable with the value of the status attribute. There is no testing of the status attribute’s value.
  
  where [status: _status],

- A combination of both preceding forms. The following example tests for events that are not closed and are of a critical severity. If the event passes those tests, the _hostname variable is instantiated with the name of the host where the event occurred.
  
  where [
    status: _status outside ['CLOSED'],
    severity: equals 'CRITICAL',
    hostname: _hostname
  ],

**Change rule filters**
You can have the following two additional filters in change rules:

- Sender filter
- Attribute change filter

**Sender filter:** A sender filter of a change rule lets you filter on the sender of a change request. Senders of change requests can be one of the following:

  - **agent**  An event adapter.
  - **engine**  The rule engine.
  - **operator(name)**  An administrator from an event console.

The valid operators for a sender filter are:

  - **equals**  Tests whether the sender is a specific value. The following example tests whether the sender is the rule engine:
    
    sender: equals engine,

    The following example tests whether the sender is an operator. This example also instantiates a new variable, _sender, which can be used in a subsequent action. (The _x variable is not available outside the filter.)
    
    sender: _sender equals operator(_x),

  - **outside**  Tests whether the sender is not equal to one of the values in a list. The following example tests whether the sender is an administrator:
    
    sender: outside [agent,engine],

  - **within**  Tests whether the sender is equal to one of the values in a list. The following example tests whether the sender is member of a set of administrators, and if so, instantiates the _sender variable with the value:
    
    sender: _sender within [operator('root@orange'),
    operator('root@red'),
    operator('root@blue')],

You can also use a sender filter to simply instantiate a variable with the value of the sender, as shown in the following example:
sender: _sender,

**Attribute change filter:** An attribute change filter of a change rule lets you filter on the requested change to an attribute value of an event.

The only required operator for an attribute change filter is set_to. The operand on the left side of the set_to operator is the name of the attribute to test. The operand on the right side of the set_to operator is the value specified in the requested change, or a comparison to the requested change. The operand on the right side of the set_to operator can be a single value or a list specification. For a list specification, a variable must be used to hold the value of the requested change.

The following operators are valid for comparisons in an attribute change filter:
- equals
- greater_than
- greater_or_equals
- smaller_than
- smaller_or_equals
- within
- outside

The **within** operator tests whether an attribute value is equal to one of the values in a list, while the **outside** operator tests whether an attribute value is equal to a value not in the list.

**Note:** Ensure that you use the appropriate operator for the data type to test.

The following examples show various attribute change filters:
- The following attribute filter succeeds if the status attribute was changed to CLOSED:
  
  ```
  attribute: status set_to 'CLOSED',
  ```

  **Note:** The **slot** keyword for an attribute change filter was used in earlier releases of the IBM Tivoli Enterprise Console product. The attribute keyword is now used. Although the slot keyword is still valid, it might be removed in a future release.

- The following attribute filter succeeds if the status attribute was changed to any value. The **newstatus** variable is instantiated with the value in the requested change.
  
  ```
  attribute: status set_to _newstatus,
  ```

- The following attribute filter succeeds if the status attribute was changed to ACK or CLOSED. The **newstatus** variable is instantiated with the value in the requested change.
  
  ```
  attribute: status set_to _newstatus within ['ACK','CLOSED'],
  ```

- The following attribute filter succeeds if the repeat_count attribute was changed to a value greater than 5. The **newrepeatcount** variable is instantiated with the value in the requested change.
  
  ```
  attribute: repeat_count set_to _newrepeatcount greater_than 5,
  ```

- The following attribute filter succeeds if the administrator attribute was changed to fred, wilma, or betty. The **newadministrator** variable is instantiated with the value in the requested change.
attribute: administrator set_to _newadministrator
within ['fred','wilma','betty'],

**Timer rule filters**
You can have the following two additional filters in timer rules:
- timer_info filter
- timer_duration filter

**timer_info filter:** A timer_info filter of a timer rule lets you filter on the timer information specified with the timer_info argument of the set_timer predicate. The set_timer predicate is used to set a timer on an incoming event. The timer_info argument of the set_timer predicate can be any value, such as an integer, string, or a structured item (such as a list).

The following operators are valid for comparisons in a timer_info filter:
- equals
- greater_than
- greater_or_equals
- smaller_than
- smaller_or_equals
- within
- outside

The `within` operator tests whether an attribute value is equal to one of the values in a list, while the `outside` operator tests whether an attribute value is equal to a value not in the list.

**Note:** Ensure that you use the appropriate operator for the data type to test.

The following examples show various timer_info filters:
- The following timer_info filter succeeds if the timer information was specified as Level 1:
  
  timer_info: equals 'Level 1',

- The following timer_info filter succeeds if the timer information was specified as either Level 1 or Level 2.
  
  _timerinfo variable is instantiated with the value of the timer information.

  timer_info: _timerinfo within ['Level 1','Level 2'],

- The following timer_info filter simply instantiates the _timerinfo variable with the timer information:
  
  timer_info: _timerinfo,

**timer_duration filter:** A timer_duration filter of a timer rule lets you filter on the value specified for the timer_duration argument of the set_timer predicate. The set_timer predicate is used to set a timer on an incoming event. The timer_duration argument of the set_timer predicate is an integer value representing the number of seconds for the duration of the timer.

The following operators are valid for comparisons in a timer_duration filter:
- equals
- greater_than
- greater_or_equals
- smaller_than
• smaller_or_equal
• within
• outside

The within operator tests whether an attribute value is equal to one of the values in a list, while the outside operator tests whether an attribute value is equal to a value not in the list.

Note: Ensure that you use the appropriate operator for the data type to test.

The following examples show various timer_duration filters:
• The following timer_duration filter succeeds if the timer was specified to last 60 seconds:
  timer_duration: equals 60,
• The following timer_duration filter succeeds if the timer was specified to last either 60 or 90 seconds. The _timerduration variable is instantiated with the value of the timer duration.
  timer_duration: _timerduration within ['60', '90'],
• The following timer_duration filter simply instantiates the _timerduration variable with the timer duration:
  timer_duration: _timerduration,

Actions

Note: The actions described in this section only pertain to plain, change, and timer rules. Simple rules use simple conditions and simple actions, as described in “General structure of a simple rule” on page 57. Correlation rules use event relationships for their actions, as described in “General structure of a correlation rule” on page 58.

When the event under analysis matches the conditions in the initial event filter of a rule, the rule is triggered and the rule engine runs the actions of the rule. A rule can contain one or more actions, each separated by a comma. A rule action contains one or more predicate calls. The following example shows the structure of a very simple rule action.

```
Action name

Action            action:ticket_sh:
                   exec_program(_event,'tpm_ticket.sh','',[],'YES')

Action body
```

The types of actions are shown in the following list, and described in “Action types” on page 68.

• action
• reception_action
• redo_action
An action should be given a meaningful name, so it can be easily identified when tracing rules. Action names must also be unique within each rule. If you don’t provide a name for an action, the rule compiler assigns its own names, such as action_1, action_2, and so forth.

The rule engine runs actions in sequential order as defined by their position in the rule, unless a predicate is called that changes the flow of control, such as the commit_action predicate. There is no guarantee of execution order of predicate calls within an action. (Actually, predicate execution order within an action conforms to Prolog logic execution rules.) If you do not know Prolog logic execution rules and want predicates to run in a certain order, place them in separate actions within a rule and order the actions accordingly.

**Action types**

There are three types of rule actions, described as follows:

- **action** Run each time a rule is triggered. This action is run for all events.

- **reception_action** Run only the first time a rule is triggered for a newly received event. This action is typically used for filtering duplicate events and placing redo requests. A reception_action is only valid in a plain rule.

  **Note:** If an internal event is generated using the generate_event predicate, the event is not placed into the reception buffer but reception_actions are still run on the event.

- **redo_action** Run only when a redo request is applied to a previously received event. A redo request is a reanalysis of an event. The redo_analysis predicate places a redo request. A redo_action is only valid in a plain rule.

The following example shows the usage of different action types:

- The oserv_script reception_action is only run for newly arriving events of class universal_oserv with a severity of WARNING. The reception_action is not run for redo requests.

- The link_host regular action is run for all events with the same criteria as stated previously, whether the event is newly received or has been previously received but is being reanalyzed from a redo request.

If this action would have been written as a redo_action instead of a regular action, it would only run upon a redo request. By specifying the event linking to occur in a regular action, the action will run regardless of the order of arrival of events. For additional information about redo requests, see [“redo_analysis” on page 196](#).
first_instance(  
    event: _host_ev of_class 'universal_host'
    where [severity: within ['CRITICAL','FATAL'],
        probe_arg: equals _probe_arg,
        status: outside ['CLOSED']
    ],
  ),

  set_event_status(_event,'ACK'),
  link_effect_to_cause(_event, _host_ev)
}
).

Directives
Directives can be included in rules. Directives specified within rule sets or rules are defined with the directive: keyword syntax. If you specify more than one directive in a directive clause, separate the directives with commas; for example, directive: fire_on_non_leaf, profile. The directives are described as follows:

fire_on_non_leaf
Enables the evaluation of rules for events that are non-leaf node classes; that is, superclasses. By default, rules are only triggered by events that are leaf-node classes. This directive must be placed before the initial event filter in a rule.

The following example shows a use of the directive. This particular example causes the rule to evaluate all events because it specifies the base event class EVENT in its event filter. All other classes inherit from the base event class.

rule: test_rls: (  
    directive: fire_on_non_leaf,
    event: _evt of_class within ['EVENT']
    where [],

    reception_action: action0:
    {  
        drop_received_event
    }
    ).

profile
Enables profiling of rule execution. This directive enables you to obtain detailed information in report form about the execution of each rule being profiled. The profile directive can be placed at the beginning of a rule set or within individual rules. You can also profile an entire rule base with the wrb –comrules –profile command. Profiling is not enabled by default. See “Profiling rules” on page 239 for additional information.

trace
Enables tracing of rule execution. The trace directive can be placed at the beginning of a rule set or within individual rules. You can also trace an entire rule base with the wrb –comrules –trace command or using the rule builder dialog. Tracing is not enabled by default. See “Tracing rules” on page 234 for additional information.

Rule language predicates
The action of a rule consists of calls to predicates. A predicate is generally equivalent to a function call in other programming languages. There are predicates built-in to the version of Prolog supported by the rule language, and there are predefined predicates to help you create useful rules. You can also create your own
predicates and make them available for use in your rules. Chapter 4, “Rule language reference”, on page 49 contains information about Prolog and how you can create your own predicates.

This section describes the predicates defined by IBM. The predicates are categorized in a quick-reference section by the function they provide. Following the quick-reference section is an alphabetical listing that describes each predicate in detail.

Quick Reference of Rule Language Predicates

This section categorizes the predicates. Each category contains a table that lists and briefly describes the predicates for a particular category. The predicates are described in detail in “Alphabetic listing of rule language predicates” on page 77.

Event correlation

These predicates let you define event sequences of root cause, problem, effect, and clearing events so each rule can manage many events based on defined event behavior rather than the number of events in the environment. Predicates are provided to define the event relationships and to search the event cache based on those defined relationships.

Chapter 5, “Correlation examples”, on page 217 provides examples of how to use these predicates.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_clear_targets</td>
<td>Returns all events in the cache that the specified clearing event clears.</td>
</tr>
<tr>
<td>any_clear_target</td>
<td>Returns the most recent event in the cache that the specified clearing event clears.</td>
</tr>
<tr>
<td>any_clearing_event</td>
<td>Returns the first event in the cache that clears the reference event.</td>
</tr>
<tr>
<td>attr_condition</td>
<td>Defines absolute attribute conditions for a single event in an event sequence.</td>
</tr>
<tr>
<td>attr_exception</td>
<td>Defines an attribute that must match a different attribute in other events in an event sequence.</td>
</tr>
<tr>
<td>attr_sequence</td>
<td>Defines the values of an attribute that change due to an event’s position in an event sequence.</td>
</tr>
<tr>
<td>clears</td>
<td>Defines a clearing event for one or more events in an event sequence.</td>
</tr>
<tr>
<td>create_clearing_event</td>
<td>Defines a clearing event.</td>
</tr>
<tr>
<td>create_event_sequence</td>
<td>Defines a sequence of events for correlation.</td>
</tr>
<tr>
<td>first_causal_event</td>
<td>Searches the event cache for the root cause event related to an effect event.</td>
</tr>
<tr>
<td>first_effect_event</td>
<td>Searches the event cache for the logically earliest effect event related to a cause event.</td>
</tr>
<tr>
<td>first_related_event</td>
<td>Searches the event cache for the logically earliest event related to a reference event.</td>
</tr>
<tr>
<td>is_clearing_event</td>
<td>Tests whether an event has been defined as a clearing event with the create_clearing_event or create_event_sequence predicate.</td>
</tr>
</tbody>
</table>
**Predicate Description**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>link_effect_to_cause</td>
<td>Links an effect event to a cause event.</td>
</tr>
<tr>
<td>unlink_from_cause</td>
<td>Unlinks an effect event from a cause event.</td>
</tr>
</tbody>
</table>

**Event cache searching**

These predicates let you query the event cache for events that are related to a reference event, which is typically the event under analysis but can be any event. Searching the event cache for related events begins with the most recent event in the cache and progresses backwards to the oldest. Some examples of tasks you can do with the assistance of event cache search predicates are:

- Modify an event only if another specified event has already been received
- Modify previously received events
- Make an event a consequence of another specified event
- Efficiently handle duplicate events
- Search for specific events within a specified time window
- Write the event cache to a file

Depending on the predicate used, you can provide the following information for querying the event cache:

- Event class
- Attribute conditions
- A time interval before or after the reception of the reference event to limit the search
- Whether the search criteria you specify is applicable to leaf classes or both leaf classes and non-leaf classes
- Whether to apply duplicate detection criteria to the search
- The maximum number of events to be returned by the search
- Change the order of how the events are returned from the search

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_duplicates</td>
<td>Succeeds once for each duplicate event that satisfies the specified additional attribute and time window conditions.</td>
</tr>
<tr>
<td>all_instances</td>
<td>Succeeds once for each event that satisfies the specified conditions.</td>
</tr>
<tr>
<td>create_cache_search_criteria</td>
<td>Defines a named search for an event cache query.</td>
</tr>
<tr>
<td>first_duplicate</td>
<td>Succeeds once for the first duplicate event that satisfies the specified additional attribute and time window conditions.</td>
</tr>
<tr>
<td>first_instance</td>
<td>Succeeds once for the first event that satisfies the specified conditions.</td>
</tr>
<tr>
<td>print_cache</td>
<td>Writes the event cache to a file.</td>
</tr>
<tr>
<td>search_cache</td>
<td>Performs a query of the event cache based on a named search defined with the create_cache_search_criteria predicate.</td>
</tr>
</tbody>
</table>

**Usage notes:**
• The scope of a variable in an event cache search predicate is limited to the rule action where it first appears; unlike a regular variable, whose scope is the entire rule where it appears.

• For a predicate that uses time window arguments, both the before and after time arguments must be specified. This is true even when the event used as the basis for the time window is the current event under analysis and the after time argument doesn’t apply because the event under analysis is the last one received. In the case of the event under analysis, the after time argument is set to 0 no matter what value is specified in the predicate. If you do not specify both time window arguments, a syntax error occurs.

• If any of these predicates are called during a redo_analysis, change rule, or timer rule, it is possible for the event to be returned, since the event might also be in the event cache.

What is a duplicate event?: Duplicate events are event instances of the same class that have the following characteristics:

• The same values for all attributes defined with the dup_detect facet set to YES

• If there are no attributes defined with the dup_detect facet set to YES, all events of that class are duplicates

Note: You can specify which subset of attributes are to be considered for identifying duplicate events. See “Attribute conditions” on page 63 for additional information.

Generally, duplicate events are not kept in the event database, but are instead used to increase the severity of the event or to count the number of times the event has been received.

Duplicate events are managed with the first_duplicate and all_duplicates predicates.

Flow control
These predicates let you control rule exit actions, such as immediately exit a set of rule actions, a set of rules, or an entire rule base.

For each event, the active rule base is traversed sequentially starting at the first rule in the first rule set, moving to the second rule in the first rule set, and so forth for each rule set. When an event causes a rule action to run, you might want to immediately exit the rule, rule set, or rule base for performance reasons.

The following table lists the predicates of this category:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit_action</td>
<td>Controls execution within the actions of the current rule.</td>
</tr>
<tr>
<td>commit_rule</td>
<td>Controls execution within a rule set.</td>
</tr>
<tr>
<td>commit_set</td>
<td>Controls execution within a rule base.</td>
</tr>
</tbody>
</table>

BAROC manipulation
These predicates let you manipulate various BAROC items in a rule base or event.
The following table lists the predicates of this category:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bo_add_at_slotval_begin</td>
<td>Adds an element to the beginning of an attribute’s list of values.</td>
</tr>
<tr>
<td>bo_add_at_slotval_end</td>
<td>Adds an element to the end of an attribute’s list of values.</td>
</tr>
<tr>
<td>bo_get_class_of</td>
<td>Gets the class of an event.</td>
</tr>
<tr>
<td>bo_get_class_slots</td>
<td>Gets attribute information for an event class.</td>
</tr>
<tr>
<td>bo_get_enum_options</td>
<td>Gets the elements of an enumeration.</td>
</tr>
<tr>
<td>bo_get_slotval</td>
<td>Gets the value of an attribute from an event.</td>
</tr>
<tr>
<td>bo_is_defined_for_class</td>
<td>Checks whether an attribute is defined for an event class.</td>
</tr>
<tr>
<td>bo_is_direct_super_of</td>
<td>Checks whether an event class is a direct superclass of another event class.</td>
</tr>
<tr>
<td>bo_is_super_of</td>
<td>Checks whether an event class is a superclass of another event class.</td>
</tr>
<tr>
<td>bo_remove_from_slotval</td>
<td>Removes a value from an attribute’s list of values.</td>
</tr>
<tr>
<td>bo_reset_default_slotval</td>
<td>Resets the attribute value for an event to the default value.</td>
</tr>
<tr>
<td>bo_set_slotval</td>
<td>Updates an event attribute value.</td>
</tr>
<tr>
<td>print_class_tree</td>
<td>Formats and writes an event class hierarchy tree from the active rule base to a file.</td>
</tr>
</tbody>
</table>

**Event criteria**

These predicates let you define the criteria for determining the state of an event. A criteria is created once, using the create_event_criteria predicate. A defined criteria is used where needed in any rule action, by the check_event_criteria predicate. Any condition for an event can be determined using these predicates, including finding patterns within the values of string attributes with regular expressions.

The following table lists the predicates of this category:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_event_criteria</td>
<td>Applies criteria to an event instance to determine if the instance meets the criteria.</td>
</tr>
<tr>
<td>create_event_criteria</td>
<td>Specifies criteria for an event. These criteria are used by the check_event_criteria predicate and the create_cache_search_criteria predicate.</td>
</tr>
</tbody>
</table>

**Event activity**

These predicates let you define the criteria for creating text reports of event activity.

The following table lists the predicates of this category:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_event_activity</td>
<td>Defines the reporting criteria for generating an event activity report.</td>
</tr>
</tbody>
</table>
### Predicate Description

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print_event_activity</code></td>
<td>Writes the event activity report defined with the <code>init_event_activity</code> predicate.</td>
</tr>
<tr>
<td><code>reset_event_activity</code></td>
<td>Resets the counts for all event reporting criteria to 0.</td>
</tr>
<tr>
<td><code>update_event_activity</code></td>
<td>Captures event information for reporting by the <code>print_event_activity_report</code> predicate.</td>
</tr>
</tbody>
</table>

### Thresholds

These predicates let you define the criteria for querying the event cache to check if a certain number of events have been received within a certain amount of time. Also included are predicates that perform the query. The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_all_thresholds</code></td>
<td>Applies criteria to events to determine if any threshold is exceeded.</td>
</tr>
<tr>
<td><code>check_threshold</code></td>
<td>Applies criteria to events to determine if a specific threshold is exceeded.</td>
</tr>
<tr>
<td><code>create_threshold</code></td>
<td>Defines a threshold.</td>
</tr>
</tbody>
</table>

### Regular expressions

These predicates let you define named regular expressions and use them for pattern matching and manipulation of strings. The regular expression functionality uses Perl notation and is the same as that provided by the Tivoli Management Framework. All of these predicates use the `log_error` predicate to report errors, so invalid arguments are reported to the error file and a TEC_Error event is generated. The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>re_after_match</code></td>
<td>Searches for a match in a string using a named regular expression and returns the substring located after the match as a result.</td>
</tr>
<tr>
<td><code>re_before_match</code></td>
<td>Searches for a match in a string using a named regular expression and returns the substring located before the match as a result.</td>
</tr>
<tr>
<td><code>re_create</code></td>
<td>Defines a regular expression for use with other regular expression predicates.</td>
</tr>
<tr>
<td><code>re_match</code></td>
<td>Searches for a match in a string using a named regular expression and returns a result.</td>
</tr>
<tr>
<td><code>re_search_string</code></td>
<td>Searches for a match in a string using a named regular expression.</td>
</tr>
<tr>
<td><code>re_substitute</code></td>
<td>Searches for a match in a string using a named regular expression, replaces the match, and returns the new string as a result.</td>
</tr>
<tr>
<td><code>re_substitute_global</code></td>
<td>Searches for all matches in a string using a named regular expression, replaces them, and returns the new string as a result.</td>
</tr>
</tbody>
</table>
Global variables
Global variables let you store information in the knowledge base that is accessible to all rules. Global variables are described in "Global variables" on page 51.

The following table lists the predicates in this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eraseGlobals</td>
<td>Removes all the global variables in a group from the knowledge base.</td>
</tr>
<tr>
<td>getGlobalGrp</td>
<td>Gets the value of all global variables in a group.</td>
</tr>
<tr>
<td>getGlobalVar</td>
<td>Gets a value of a global variable.</td>
</tr>
<tr>
<td>getGlobals</td>
<td>Gets all global variables.</td>
</tr>
<tr>
<td>global_exists</td>
<td>Checks the existence of a global variable.</td>
</tr>
<tr>
<td>loadGlobals</td>
<td>Loads global variables from a file into the knowledge base.</td>
</tr>
<tr>
<td>resetGlobalGrp</td>
<td>Resets the value of all global variables in a group.</td>
</tr>
<tr>
<td>saveGlobals</td>
<td>Writes all global variables from a group to a file.</td>
</tr>
<tr>
<td>setGlobalVar</td>
<td>Sets the value of a global variable.</td>
</tr>
</tbody>
</table>

Time manipulation
These predicates let you get and convert various formats of time.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convertAsciiTime</td>
<td>Converts a time structure to an atom.</td>
</tr>
<tr>
<td>convertGmTime</td>
<td>Converts an epoch time number to a time structure in Greenwich mean time.</td>
</tr>
<tr>
<td>convertLocalTime</td>
<td>Converts an epoch time number to a time structure in local system time.</td>
</tr>
<tr>
<td>getGmTime</td>
<td>Gets the current time represented in Greenwich mean time.</td>
</tr>
<tr>
<td>getLocalTime</td>
<td>Gets the current local system time.</td>
</tr>
<tr>
<td>getTime</td>
<td>Gets the current time represented by an integer since the epoch.</td>
</tr>
<tr>
<td>resolveTime</td>
<td>Retrieves the attributes of a time structure.</td>
</tr>
</tbody>
</table>

Launching programs and tasks
These predicates allow you to launch programs or tasks.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execProgram</td>
<td>Launches a program.</td>
</tr>
<tr>
<td>execProgramLocal</td>
<td>Launches a program on a local server.</td>
</tr>
<tr>
<td>execTask</td>
<td>Launches a task from a task library.</td>
</tr>
<tr>
<td>execTaskLocal</td>
<td>Launches a task from a task library on the local server.</td>
</tr>
</tbody>
</table>
Attribute and status manipulation
These predicates allow you to manipulate event attributes or event status.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_to_repeat_count</td>
<td>Adds a number to the repeat_count attribute for an event.</td>
</tr>
<tr>
<td>change_event_administrator</td>
<td>Changes the administrator of an event.</td>
</tr>
<tr>
<td>change_event_severity</td>
<td>Changes the severity of an event.</td>
</tr>
<tr>
<td>change_event_status</td>
<td>Changes the status of an event.</td>
</tr>
<tr>
<td>decrement_slot</td>
<td>Subtracts a number from the value of the specified integer attribute</td>
</tr>
<tr>
<td>get_attributes</td>
<td>Retrieves event attribute values.</td>
</tr>
<tr>
<td>increment_slot</td>
<td>Adds a number to the value of the specified integer attribute.</td>
</tr>
<tr>
<td>place_change_request</td>
<td>Requests a change to an attribute value.</td>
</tr>
<tr>
<td>re_split_event_id</td>
<td>Parses an element of the server_path event attribute.</td>
</tr>
<tr>
<td>set_event_administrator</td>
<td>Sets the administrator for an event.</td>
</tr>
<tr>
<td>set_event_message</td>
<td>Sets the msg attribute of an event.</td>
</tr>
<tr>
<td>set_event_severity</td>
<td>Sets the severity of an event.</td>
</tr>
<tr>
<td>set_event_status</td>
<td>Sets the status of an event.</td>
</tr>
</tbody>
</table>

General purpose
These are general purpose predicates.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_and_increment_count</td>
<td>Checks a counter created by the init_count predicate.</td>
</tr>
<tr>
<td>clear_closed_events</td>
<td>Clears closed events from the event cache.</td>
</tr>
<tr>
<td>drop_change_request</td>
<td>Prevents a change request from being applied after change rules are run.</td>
</tr>
<tr>
<td>drop_received_event</td>
<td>Discards an event after the rules are run.</td>
</tr>
<tr>
<td>forward_event</td>
<td>Forwards an event to an event server.</td>
</tr>
<tr>
<td>generate_event</td>
<td>Generates an internal event.</td>
</tr>
<tr>
<td>get_config_param</td>
<td>Gets a rule engine configuration setting.</td>
</tr>
<tr>
<td>init_count</td>
<td>Creates and initializes a counter.</td>
</tr>
<tr>
<td>ip_node_unreachable</td>
<td>Determines if the event was sent from an unreachable subnet.</td>
</tr>
<tr>
<td>re_mark_as_modified</td>
<td>Updates information for an event in the event database.</td>
</tr>
<tr>
<td>re_send_event_conf</td>
<td>Sends an event to a remote event server.</td>
</tr>
<tr>
<td>redo_analysis</td>
<td>Requests a reanalysis for an event.</td>
</tr>
<tr>
<td>remove_bslashes</td>
<td>Converts back slashes to forward slashes in directory paths.</td>
</tr>
<tr>
<td>set_timer</td>
<td>Sets a timer on an event.</td>
</tr>
</tbody>
</table>
**Debug**
These predicates help you to analyze and debug rules you have created.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_error</td>
<td>Generates error messages to assist in rule development.</td>
</tr>
<tr>
<td>set_detailed_debugging</td>
<td>Enables tracing of user-defined predicates.</td>
</tr>
<tr>
<td>set_log_error_source</td>
<td>Defines a source identifier for a point of reference from an error message generated by the log_error predicate.</td>
</tr>
<tr>
<td>trace_it</td>
<td>Writes rule trace information to the rule trace file for predicates.</td>
</tr>
</tbody>
</table>

**Alphabetic listing of rule language predicates**

The following section lists the rule language predicates in alphabetical order.
**add_to_repeat_count**

Adds a number to the repeat_count attribute for an event.

**Synopsis:**  
`add_to_repeat_count(event, number)`

**Description:**  
This predicate is typically used to keep a count of duplicate events that are received. As they are received the repeat_count attribute of the original event is updated with this predicate and the duplicate event is dropped.

**Arguments:**

- `event`  
The event whose repeat_count attribute is incremented.

- `number`  
The number added to the current repeat_count attribute value.

**Examples:**  
The following example shows how events for printer problems are managed:

1. An event cache query is run searching for the first duplicate event that matches the class of the event under analysis and not closed. The time window for the search is 10 minutes.

2. If a duplicate event is found in the cache, the cached event’s repeat_count attribute is incremented by 1, the newly received event is dropped, and rule evaluation continues with the next rule set.

   If a duplicate is not found, the newly received event is processed as usual and added to the event cache.

   ```
   rule: printer_problem:
     event: _event of_class
     reception_action:
       {
         first_duplicate(_event,
           event: _printer_ev
             where [
               status: outside ['CLOSED']
             ],
           event - 600 - 600
         ),
         commit_rule,
         add_to_repeat_count(_printer_ev, 1),
         drop_received_event
       }
   ).
   ```

**See Also:**  
[first_duplicate](#)
**all_clear_targets**
Returns all events in the cache that the specified clearing event clears.

**Synopsis:**  `all_clear_targets(_clear_event, _target_event)`

—OR—

`all_clear_targets(_clear_event, _target_event, time_before, time_after)`

**Description:** This predicate is used to search the event cache for all of the events that `_clear_event` clears. Each found event is returned in `_target_event`.

If the `time_before` and `time_after` arguments are not specified, the event cache search time window takes the default value of 2 years (1 year before and 1 year after). You should limit a time window to the smallest reasonable window whenever possible for better performance.

**Arguments:**

`_clear_event`
A pointer to the clearing event.

`_target_event`
A pointer to each event found in the cache that `_clear_event` clears. Must be free.

`time_after`
The number of seconds after `_clear_event` has been received. This argument is used to limit the event cache search to a time window.

`time_before`
The number of seconds before `_clear_event` has been received. This argument is used to limit the event cache search to a time window.

**Examples:** The following example searches the event cache for all events that are cleared by the event under analysis. As each event is found, it is closed. This rule triggers on the base event, so every incoming leaf event is tested.

```plaintext
rule: 'clear_target_events':(
    event: _clr_ev of_class 'EVENT',
    action: 'search_for_target':(
        all_clear_targets(_clr_ev, _tgt, 3600, 0),
        set_event_status(_tgt, 'CLOSED')
    )
).
```

**See Also:** [any_clear_target](#), [create_clearing_event](#), [create_event_sequence](#)
**all_duplicates**
Succeeds once for each duplicate event that satisfies the specified additional attribute and time window conditions.

**Synopsis:**  
`all_duplicates(_event, event: _duplicate where attribute_conditions)`

——OR——

`all_duplicates(_event, event: _duplicate where attribute_conditions, _referenceEvent ~time_before ~time_after)`

**Description:**  
The rule engine does not explicitly provide a loop structure. However, the all_duplicates predicate returns multiple solutions one at a time. When the rule engine runs the all_duplicates predicate, it analyzes all solutions individually and runs the remaining sequence of predicates in the action for each solution.

No class specification is required, since duplicate events are the same class as the event under analysis. For additional information about duplicate events, see “What is a duplicate event?” on page 72.

**Arguments:**

* _event*  
A pointer to the event currently under analysis.

* _referenceEvent*  
A pointer to the reference event for the time window, typically the event under analysis.

* event: _duplicate where attribute_conditions*  
Specifies an event filter for querying the event cache. _duplicate is instantiated with a pointer to each duplicate event found. See “Event filters” on page 60 for additional information.

* ~time_after*  
The number of seconds after the reference event.

* ~time_before*  
The number of seconds before the reference event.

**Examples:**  
The following example shows a rule that:
1. Queries the event cache for duplicates of the OV_NODE_DOWN class that are not closed and within a 600 second time window of the event under analysis.
2. When duplicate events are found, their severity is assigned the same severity as the event under analysis.

```rule
dup_nfs_not_respond:
  event: _event of_class 'OV_NODE_DOWN'
    where [severity: _severity],
  action: dup_event_severity:
    all_duplicates(_event, event: _dup_OV_ev
      where [status: outside ['CLOSED'] ],
      event ~300 ~300 ),
    set_event_severity(_dup_OV_ev, _severity)
  ).
```

80 IBM Tivoli Enterprise Console: Rule Developer’s Guide
See Also: first_duplicate
**all_instances**
Succeeds once for each event that satisfies the specified class, attribute, and time window conditions.

**Synopsis:**  
\[
\text{all_instances}\left( \_\text{event}, \_\text{eventInstance of class class where attribute\_conditions} \right)
\]

—OR—

\[
\text{all_instances}\left( \_\text{event}, \_\text{eventInstance of class class where attribute\_conditions, referenceEvent \_time\_before \_time\_after} \right)
\]

**Description:** The rule engine does not explicitly provide a loop structure. However, the all_instances predicate returns multiple solutions one at a time. When the rule engine runs the all_instances predicate, it analyzes all solutions individually and runs the remaining sequence of predicates in the action for each solution.

**Arguments:**

\_\text{event}  
A pointer to the event currently under analysis.

\_\text{referenceEvent}  
A pointer to the reference event for the time window, typically the event under analysis.

\_\text{eventInstance of class class where attribute\_conditions}  
Specifies an event filter for querying the event cache. \_\text{eventInstance} is instantiated with a pointer to each event instance found. See “Event filters” on page 60 for additional information.

\_\text{time\_after}  
The number of seconds after the reference event.

\_\text{time\_before}  
The number of seconds before the reference event.

**Examples:** The following example action closes all NFS_SERVER_NOT_RESPONDING events whose server attribute has a value of Pascal:

\[
\text{action: (}
\text{all_instances(\_\text{event, event: nfs\_ev of class 'NFS_SERVER_NOT\_RESPONDING' where [server: equals 'Pascal']}},
\text{set\_event\_status (nfs\_ev, 'CLOSED')}\text{)}
\]

**See Also:** [first_instance](#)
any_clear_target
Returns the most recent event in the cache that the specified clearing event clears.

Synopsis: any_clear_target(_clear_event, _target_event)

—OR—

any_clear_target(_clear_event, _target_event, time_before, time_after)

Description: This predicate is used to search the event cache for the first event that _clear_event clears. The found event is returned in _target_event.

If the time_before and time_after arguments are not specified, the event cache search time window defaults to two years (one year before and one year after). For performance reasons, you should limit a time window to the smallest reasonable window whenever possible.

Arguments:

_clear_event
A pointer to the clearing event.

_target_event
A pointer to the first event found in the cache that _clear_event clears. Must be free.

time_after
The number of seconds after _clear_event has been received. This argument is used to limit the event cache search to a time window.

time_before
The number of seconds before _clear_event has been received. This argument is used to limit the event cache search to a time window.

Examples: The following example searches the event cache for the first event found that is cleared by the event under analysis. If an event is found, it is closed.

rule: 'clear_target_event':
  event: _clr_ev of_class 'EVENT',
  action: 'search_for_target':{
    any_clear_target(_clr_ev, _tgt, 3600, 0),
    set_event_status(_tgt, 'CLOSED')
  }
).

See Also: all_clear_targets, create_clearing_event, create_event_sequence
any_clearing_event
Returns the first event in the cache that clears the reference event.

Synopsis: any_clearing_event(_event, _clear_event)

—OR—

any_clearing_event(_event, _clear_event, time_before, time_after)

Description: This predicate is used to search the event cache for the first clearing event that can be found for _event.

If the time_before and time_after arguments are not specified, the event cache search time window defaults to 2 years (1 year before and 1 year after). You should limit a time window to the smallest reasonable window whenever possible for better performance.

Arguments:

_clear_event
A pointer to the first clearing event found for _event. Must be free.

_event A pointer to the event whose clearing event is being searched for.

time_after
The number of seconds after _event has been received. This argument is used to limit the event cache search to a time window.

time_before
The number of seconds before _event has been received. This argument is used to limit the event cache search to a time window.

Examples: The following example searches the event cache for the first event found that clears the event under analysis. If a clearing event is found, the event under analysis is closed and processing is finished. This rule triggers on the base event, so every incoming leaf event is tested.

rule: 'check_for_clear':(

  event: _ev of_class 'EVENT',
  action: 'search_for_clear':(
    any_clearing_event(_ev, _clr, 3600, 0),
    set_event_status(_ev, 'CLOSED'),
    commit_set
  ),
).

See Also: create_clearing_event, create_event_sequence
**attr_condition**
Defines absolute attribute conditions for a single event in an event sequence.

**Synopsis:** attr_condition([classes], [attribute_conditions])

**Description:** This predicate is used to define attribute conditions for some of the problem events in an event sequence. It must be called from the event_details argument of the create_event_sequence predicate. (The attribute_conditions argument of the create_event_sequence predicate defines attribute conditions for all events in an event sequence.)

Attribute conditions for clearing events are defined with the clears predicate.

**Arguments:**

*classes* The names of one or more event classes for which the attribute conditions apply, in list format. For example, ['upsOnBattery','lowBattery','upsDischarged'].

*attribute_conditions*
The list of conditions for attributes that must be met by the subset of event classes in the event sequence defined with the classes argument. Only absolute conditions apply. An absolute condition is similar to an attribute condition in an event filter. For example, [severity,equals,'HARMLESS']. To specify multiple attribute conditions, use a nested list; for example, [[severity,equals,'HARMLESS'],[hostname,equals,orange]].

See the attribute_conditions argument description for the create_event_criteria on page 129 for additional details about specifying attribute conditions for this argument, as the syntax is the same.

**Examples:** The following example shows the event sequence for events sent from two monitoring sources: American Power Conversion (APC) uninterruptible power supply and IBM Tivoli Distributed Monitoring. APC events use the hostname attribute to identify affected components. Tivoli Distributed Monitoring uses the probe_arg attribute of the universal_host event to identify affected components.

The attr_condition predicate defines the attribute conditions that must be met by the universal_host event to be eligible for correlation (in this case the severity attribute must equal a value of FATAL).

The attr_exception predicate shows how to do the attribute mapping so the events in the sequence from the two different sources can be correlated.

```plaintext
create_event_sequence(
    ['upsOnBattery',
    'lowBattery',
    'upsDischarged',
    'universal_host'],

    ['hostname', ['status','outside'],['CLOSED']]

    [ clears('powerRestored',[],['upsOnBattery'],[]),
      clears('returnFromLowBattery',[],['lowBattery'],[]),
      clears('dischargeCleared',[],['upsDischarged'],[]),
      clears('universal_host',
        [ ['severity', equals,'HARMLESS' ]
        ['universal_host'],
        []],
      attr_condition('universal_host',
        ['severity',equals,'FATAL'])),
```

Chapter 4. Rule language reference 85
attr_exception('hostname','universal_host','probe_arg')
]
},

See Also: attr_exception, create_clearing_event, create_event_sequence
**attr_exception**
Defines an attribute that must match a different attribute in other events in an event sequence.

**Synopsis:** attr_exception(attribute, [classes], exception_attribute)

**Description:** This predicate is used to define attribute conditions for other problem events in an event sequence when the attributes do not match up between the events. It must be called from the event_details argument of the create_event_sequence predicate.

For example, suppose a monitoring system generates its events with the name of affected machines in the hostname attribute. Suppose a different monitoring system assigns the names of affected machines in the probe_arg attribute. In order to correlate these two events based on machine name, you must map the two different attributes to each other using the attr_exception predicate.

Multiple attribute exceptions for an event sequence must be defined with multiple attr_exception predicates.

**Arguments:**

- **attribute**
  The attribute to map the exception attribute to. This attribute must be defined in the attribute_conditions argument of the create_event_sequence predicate for the event sequence.

- **classes**
  The list of problem event classes for which mapping to attribute is valid.

- **exception_attribute**
  The exception attribute to map to attribute.

**Examples:** The following example shows the event sequence for events sent from two monitoring sources: APC uninterruptible power supply and Tivoli Distributed Monitoring. APC events use the hostname attribute to identify affected components. Tivoli Distributed Monitoring uses the probe_arg attribute of the universal_host event to identify affected components. The attr_exception predicate shows how to do the mapping so events are correlated.

```prolog
create_event_sequence(['upsOnBattery', 'lowBattery', 'upsDischarged', 'universal_host'],
['hostname', ['status', 'outside', ['CLOSED']]],
[attr_exception('hostname', 'universal_host', 'probe_arg')]
)
```

**See Also:** create_event_sequence
attr_sequence
Defines the values of an attribute that change due to an event’s position in an event sequence.

Synopsis: attr_sequence(class, attribute=value_sequence)

Description: This predicate is used for event classes in an event sequence whose names remain the same but use an attribute value to indicate a status change. It must be called from the event_details argument of the create_event_sequence predicate. It defines the sequence of changed values that the attribute can have as it progresses along an event sequence from left to right.

Note: An event class that adheres to this implementation can only have one attribute sequence defined.

Arguments:
attribute
   The name of the attribute whose changing value indicates status.

class
   The name of the event class containing the attribute.

value_sequence
   The values, in event sequence, of the attribute that indicate status. This argument is a list.

Examples: The following example shows an event sequence for Compaq physical drive events. These events are all of the same class (cpqTape3PhDrvStatusChange). The cpqTapePhyDrvCondition attribute value changes to indicate status, with a value of OK signifying a clearing event.

To accommodate this implementation of one event class with a changing attribute value for all of the events in an event sequence, the attribute and sequence of values (in event sequence order from left to right) must be defined with an attr_sequence predicate.

create_event_sequence(
   ['cpqTape3PhDrvStatusChange'],
   ['hostname', ['status', 'outside', ['CLOSED']]]
   [attr_sequence(
       'cpqTape3PhDrvStatusChange',
       'cpqTapePhyDrvCondition'=['Degraded', 'Failed']
    )]
)

See Also: create_event_sequence
**bo_add_at_slotval_begin**

Adds an element to the beginning of an attribute’s list of values.

**Synopsis:**  bo_add_at_slotval_begin(_event, _attribute, _value)

**Description:** This predicate adds the value _value at the beginning of the list of values for attribute _attribute in event _event. This predicate is applicable only if the data type of _attribute is a list.

**Arguments:**

_attribute  
The name of an attribute in _event.

_event  
A pointer to the event under analysis.

_value  
The value to be added to the list.

**Examples:** The following example adds a value to the beginning of the acl attribute. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (

  event: _event of_class _class,  
    % _class is unified with Su_Success.

  action: (

    % Before the call, the 'acl' attribute contains % [admin].

    bo_add_at_slotval_begin(_event, 'acl','user')

    % Now the 'acl' attribute contains [user,admin].

  )
).
```

**See Also:** None.
**bo_add_at_slotval_end**

Adds an element to the end of an attribute’s list of values.

**Synopsis:** `bo_add_at_slotval_end(_event, _attribute, _value)`

**Description:** This predicate adds the value `_value` at the end of the list of values for attribute `_attribute` in event `_event`. This predicate is applicable only if the data type of `_attribute` is a list.

**Arguments:**

- `_attribute`  
  The name of an attribute in `_event`.
- `_event`  
  A pointer to the event under analysis.
- `_value`  
  The value to be added to the list.

**Examples:** The following example rule adds a value to the end of the acl attribute. This example assumes the event under analysis is an instance of the Su_Success class.

```
rule: baroc_example: (  

  event: _event of_class _class,  
  %- _class is unified with Su_Success.

  action: {  

    %- Before the call, the 'acl' attribute contains  
    %- [admin].

    bo_add_at_slotval_end(_event, 'acl','senior')  
    %- Now the 'acl' attribute contains [admin,senior].
  }.

See Also: None.
```
**bo_get_class_of**
Gets the class of an event.

**Synopsis:**  `bo_get_class_of(_event, _classname)`

**Description:** This predicate retrieves in `_classname` the name of the class that `_event` is an instance of. If `_classname` is instantiated, the predicate only succeeds when the value corresponds to the class name of the event.

**Arguments:**
- `_classname`  The name of the class of `_event`.
- `_event`  A pointer to the event under analysis.

**Examples:** The following example shows how to get the class name of the event under analysis. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (  
  event: _event of_class _class,  
  % _class is unified with Su_Success.
  action: (  
    bo_get_class_of(_event, _classname)  
    % _classname is unified with 'Su_Success'.
  )
).
```

**See Also:** None.
**bo_get_class_slots**
Gets attribute information for an event class.

**Synopsis:**  
`bo_get_class_slots(_classname, _attributes)`

**Description:** This predicate retrieves in `_attributes` the list of attributes and their definitions for class `_classname`. The list includes all attributes known to the class; that is, all the attributes defined for the class, as well as any attributes that are inherited from any superclasses.

**Arguments:**

- `_attributes`
  A list of the attributes and their definitions that are known to the event class.
  
  For each attribute, the following information is provided:
  1. Name
  2. Complex type
  3. Element type
  4. Setting for parse facet
  5. Setting for dup_detect facet

  Each element of the list is in the following format:
  
  `slot(attribute_name,complex_type,element_type,parse_setting,dup_detect_setting,'')`

- `_classname`
  The name of the event class.

**Examples:** The following example shows how to get a list containing the attribute definitions for the class of the event under analysis. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (  
    event: _event of_class _class,  
    % _class is unified with Su_Success.
  
    action: {  
      bo_get_class_slots(_class, _slots)
    }
  ).
```

The following list is unified with `_attributes` for this example. The last piece of information for each attribute is empty and reserved for future IBM use.

```prolog
[slot(server_handle,SINGLE,INTEGER,NO,NO,''),  
slot(date_reception,SINGLE,INT32,NO,NO,''),  
slot(event_handle,SINGLE,INTEGER,NO,NO,''),  
slot(source,SINGLE,STRING,YE,NO,''),  
slot(sub_source,SINGLE,STRING,YE,NO,''),  
slot(origin,SINGLE,STRING,YE,NO,''),  
slot(sub_origin,SINGLE,STRING,YE,NO,''),  
slot(hostname,SINGLE,STRING,YE,NO,''),  
slot(fqhostname,SINGLE,STRING,YE,NO,''),  
slot(adapter_host,SINGLE,STRING,YE,NO,''),  
slot(date,SINGLE,STRING,YE,NO,''),  
slot(status,SINGLE,STATUS,YE,NO,''),  
slot(administrator,SINGLE,STRING,NO,NO,''),  
slot(acl,LIST_OF,STRING,NO,NO,''),  
slot(credibility,SINGLE,INTEGER,NO,NO,''),  
slot(severity,SINGLE,SEVERITY,YE,NO,''),
```

92 IBM Tivoli Enterprise Console: Rule Developer's Guide
slot(msg,SINGLE,STRING,YES,NO,''),
slot(msg_catalog,SINGLE,STRING,YES,NO,''),
slot(msg_index,SINGLE,INTEGER,YES,NO,''),
slot(duration,SINGLE,INTEGER,NO,NO,''),
slot(num_actions,SINGLE,INTEGER,NO,NO,''),
slot(repeat_count,SINGLE,INTEGER,YES,NO,''),
slot(cause_date_reception,SINGLE,INT32,NO,NO,''),
slot(cause_event_handle,SINGLE,INTEGER,NO,NO,''),
slot(pid,SINGLE,STRING,YES,NO,''),
slot(from_user,SINGLE,STRING,YES,YES,''),
slot(to_user,SINGLE,STRING,YES,YES,''),
slot(on_tty,SINGLE,STRING,YES,YES,''))

See Also: None.
**bo_get_enum_options**
Gets the elements of an enumeration.

**Synopsis:**  `bo_get_enum_options(_enumname, _options)`

**Description:**  Unifies a list of the string values that are part of the enumeration `_enumname` with list `_options`.

**Arguments:**

- `_enumname`
  The name of an enumeration.

- `_options`
  The values of the different enumeration options.

**Examples:**  The following example gets a list of all valid values for the SEVERITY enumeration for the event under analysis. This example assumes the event under analysis is an instance of the Su_Success class.

```rule: baroc_example: (``
  event: _event of_class _class,
    % _class is unified with Su_Success.

  action: {
    bo_get_enum_options('SEVERITY', _options)
    % _options is unified with [UNKNOWN, HARMLESS,WARNING,MINOR,CRITICAL,FATAL]
  }
).
```

**See Also:**  None.
**bo_get_slotval**

Gets the value of an attribute from an event.

**Synopsis:**  
`bo_get_slotval(_event, _attribute, _value)`

**Description:** This predicate gets the value of attribute `*_attribute*` in event `_event`. The value is unified with `_value`. The value of `_value` must have the correct type for `*_attribute*` for the predicate to succeed.

**Arguments:**

- `*_attribute*`  
  The name of the attribute whose value to obtain.

- `*_event*`  
  A pointer to the event under analysis.

- `*_value*`  
  The value of attribute `*_attribute*`.

**Examples:** The following example gets the value of the sub-source attribute from the event under analysis. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (  
  event: _event of_class _class,
  \% _class is unified with Su_Success.

  action: (  
    bo_get_slotval(_event, 'sub_source',_sub_source)
    \% _sub_source is unified with 'su'.
  )
).
```

**See Also:**  
[bo_set_slotval](#)
**bo_is_defined_for_class**
Checks whether an attribute is defined for an event class.

**Synopsis:**  bo_is_defined_for_class(_attribute, _classname)

**Description:** This predicate succeeds if attribute _attribute is defined in class _classname.

**Arguments:**

_attribute
    The attribute to check.

_classname
    The event class to check for the attribute definition.

**Examples:** The following example determines if there is an attribute named pid defined for the event under analysis. This example assumes the event under analysis is an instance of the Su_Success class.

```plaintext
rule: baroc_example: (
    event: _event of_class _class,
    %_class is unified with Su_Success.

    action: {
        bo_is_defined_for_class('pid', _class)
        % Succeeds.
    }
).
```

**See Also:** None.
**bo_is_direct_super_of**
Checks whether an event class is a direct superclass of another event class.

**Synopsis:**  `bo_is_direct_super_of(_super_classname, _classname)`

**Description:** Succeeds if `_super_classname` is a direct superclass of `_classname`.

**Arguments:**
- `_classname`
The direct subclass of `_super_classname`.
- `_super_classname`
The direct superclass of `_classname`.

**Examples:** The following example determines whether the event under analysis is a direct subclass of the Logfile_Su class. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (  
  event: event of class _class,  
  % _class is unified with Su_Success.  
  action: (    
    bo_is_direct_super_of('Logfile_Su', _class)    
    % This predicate succeeds because    
    % 'Logfile_Su' is a    
    % direct superclass of 'Su_Success'.    
  )  
).  

See Also: None.
**bo_is_super_of**
Checks whether an event class is a superclass of another event class.

**Synopsis:**  
bo_is_super_of(_super_classname, _classname)

**Description:**  This predicate succeeds if _super_classname is either a direct or indirect superclass of _classname.

**Arguments:**

-_classname
A subclass of _super_classname.

-_super_classname
A superclass of _classname.

**Examples:**  The following example determines whether the event under analysis is a subclass of the Logfile_Base class. This example assumes the event under analysis is an instance of the Su_Success class.

```RULE
rule: baroc_example: (  
  event: _event of_class _class,  
  % _class is unified with Su_Success.  

  action: (  
    bo_is_super_of('Logfile_Base', _class)  
    % _class was instantiated to 'Su_Success'  
    % in event filter.  
    % This predicate succeeds because  
    % 'Logfile_Base' is a superclass  
    % of 'Su_Success'.  
  )  
),
```

**See Also:**  None.
**bo_remove_from_slotval**
Removes a value from an attribute's list of values.

**Synopsis:** bo_remove_from_slotval(_event, _attribute, _value)

**Description:** This predicate removes the value _value from the list of values for attribute _attribute in event _event. This predicate is applicable only if the data type of _attribute is a list.

**Arguments:**
- _attribute The name of an attribute in _event.
- _event A pointer to the event under analysis.
- _value The element to be removed from _attribute.

**Examples:** The following example removes a value from the acl attribute. This example assumes the event under analysis is an instance of the Su_Success class.

```prolog
rule: baroc_example: (

event:_event of_class _class,  
   % _class is unified with Su_Success.

    action: (  

      % Before the call, the 'acl' attribute contains [admin].
      bo_add_at_slotval_end(_event, 'acl','senior'),
      % Now the 'acl' attribute contains [admin,senior].
      bo_add_at_slotval_begin(_event, 'acl','user'),
      % Now the 'acl' attribute contains [user,admin,senior].
      bo_remove_from_slotval(_event, 'acl', 'admin')
      % Now the 'acl' attribute contains [user,senior].
    ).
)

See Also: None.
**bo_reset_default_slotval**
Resets the attribute value for an event to the default value.

**Synopsis:**  
`bo_reset_default_slotval(_event, _attribute)`

**Description:**  
This predicate instantiates the default value to attribute `_attribute` of event `_event`.

**Arguments:**

- `_attribute`  
The name of an attribute in `_event`. Must be an atom.
- `_event`  
A pointer to the event under analysis.

**Examples:**  
The following example shows a use of the predicate. This example assumes the event under analysis is an instance of the Su_Success class.

```plaintext
rule: baroc_example: (
    event: _event of_class _class,
    action: (
        bo_reset_default_slotval(_event, 'sub_source')
        % The slot 'sub_source' for the event
        % under analysis is set back to 'su'.
    )
).
```

**See Also:** None.
**bo_set_slotval**
Updates an event attribute value.

**Synopsis:**  `bo_set_slotval(_event, _attribute, _value)`

**Description:** This predicate updates the value of attribute _attribute in event _event with value _value.

Unlike the place_change_request predicate, change rules are not evaluated in response to this action.

Also, unlike place_change_request, bo_set_slotval will not automatically update the attribute value in the event database or the event consoles. Often these are updated automatically as part of other rule base activity, such as when the event is initially processed or during a change rule on that event. However, if you are using bo_set_slotval from a change rule on a different event than the current event, the update will not happen. To ensure the attribute is updated everywhere, follow this up with a call to the re_mark_as_modified predicate.

**Arguments:**

- **_attribute**  
  The attribute to update.
- **_event**  
  A pointer to the pointer to the event to modify.
- **_value**  
  The new value to assign the attribute.

**Examples:** The following example shows how to update the hostname attribute of the event under analysis to the value myhost:

```plaintext
bo_set_slotval(_event, hostname, 'myhost')
```

**See Also:** `bo_get_slotval`, `place_change_request`, `re_mark_as_modified`
**cancel_all_timers**  
Cancels all timers that were set on an event.

**Synopsis:**  
cancel_all_timers(_event)

**Description:**  
This predicate cancels all timers that were set on an event.

**Arguments:**  

[_event] A pointer to the event on which the timers to cancel were set.

**Examples:**  
The following example cancels all timers that were set on an event when a more causal event is received. This example assumes some other rule not shown set the timers when the events were received.

```plaintext
rule: cancel_timers:

  event: _ev of_class 'EVENT',

  action: cancel_timers:
    first_effect_event(_ev, _effect, 300, 300),
    link_effect_to_cause(_effect, _ev),
    cancel_all_timers(_effect)
  ).
```

**See Also:**  
[cancel_timer]
cancel_timer
Cancels a timer that was set on an event.

Synopsis: cancel_timer(_event, _timer_duration, _timer_info)

Description: This predicate cancels a timer that was set on an event based on any
or all of its arguments. The _event argument should be specified whenever
possible, though. Arguments not specified must be represented in the call by a
variable that is not instantiated or by the Prolog anonymous variable (an
underscore character). Variables not instantiated in the call are unified with the
corresponding values of the cancelled timer.

Arguments:

_event  A pointer to the event whose timer to cancel.

_timer_duration  The duration (in seconds) of the timer. This value was specified when the
timer was set. See "set_timer" on page 212 additional information.

_timer_info  The timer information. This value was specified when the timer was set.
See "set_timer" on page 212 additional information.

Examples: The following example performs event correlation and sets a timer for
5 minutes before a trouble ticket is created for an event. The 5 minute delay is
intended to provide enough time for any related cause events to arrive so that the
trouble ticket is hopefully created for a root cause event rather than a related effect
event.

When an event is received, the event cache is searched for related events that have
been received within the past 5 minutes. (Related events were defined with the
create_event_sequence predicate.) If it’s determined that the event under analysis is
an effect event of a known cause event found in the cache, it is linked to the cause
event and rule processing exits. If a related event is not found for the event under
analysis, a timer is set for 5 minutes (to see if related cause events come in) and
processing exits.

If a related cause event is received within the 5 minute timer window, the existing
timer is canceled, the effect event is linked to the cause event, a new timer is set on
the event under analysis, and processing exits.

The create_trouble_ticket rule creates a trouble ticket when the timer expires. This
example assumes that the operator has created the appropriate script for their
trouble ticketing system in scripts/create_trouble_ticket.sh.

rule: set_timer_delay:
  event: _ev of_class 'EVENT',
  action: perform_correlation:
    first_related_event(_ev, _related, _type, 300, 300),
    ( _type == 'c',
       link_effect_to_cause(_ev, _related),
       commit_set
     ;
       cancel_timer(_related, 300, _info),
       link_effect_to_cause(_related, _ev)
     ),
action: set_timer:
    set_timer(_ev, 300, 0),
    commit_set

}).

timer_rule: create_trouble_ticket:

    event: _ev of class 'EVENT',
    timer_duration: equals 300,

    action: create_ticket:
        exec_program(_ev, 'scripts/create_trouble_ticket.sh', '%ld',
                     [_ev], no)

}).

See Also: cancel_all_timers
**change_event_administrator**
Changes the administrator of an event.

**Synopsis:** `change_event_administrator(_event, new_administrator)`

**Description:** This predicate places an internal request to change the administrator attribute of the specified event. This causes the change rules to evaluate the requested change before it is actually applied.

**Arguments:**
- `_event` A pointer to the event for which the administrator is to be changed.
- `new_administrator` The new event administrator.

**Examples:** The following example shows how to change the administrator attribute of the event under analysis to bjones:
```
change_event_administrator(_event, bjones)
```

**See Also:** `set_event_administrator`
**change_event_severity**
Changes the severity of an event.

**Synopsis:**  `change_event_severity(_event, new_severity)`

**Description:** This predicate places an internal request to change the severity attribute of the specified event. This causes the change rules to evaluate the requested change before it is actually applied.

**Arguments:**
- `_event` A pointer to the event for which the severity is to be changed.
- `new_severity` The new event severity.

**Examples:** The following example shows how to change the severity attribute of the event under analysis to CRITICAL:
```
change_event_severity(_event, 'CRITICAL')
```

**See Also:** [set_event_severity](#)
**change_event_status**
Changes the status of an event.

**Synopsis:** `change_event_status(_event, new_status)`

**Description:** This predicate places an internal request to change the status attribute of the specified event. This causes the change rules to evaluate the requested change before it is actually applied.

**Arguments:**
- `_event` A pointer to the event for which the status is to be changed.
- `new_status` The new event status.

**Note:** A change from ACK to OPEN status is not valid for the `new_status` argument.

**Examples:** The following example shows how to change the status attribute of the event under analysis to ACK:

`change_event_status(_event, 'ACK')`

**See Also:** [set_event_status](#)
check_all_thresholds
Applies criteria to events to determine if any threshold is exceeded.

Synopsis: \texttt{check\_all\_thresholds(\_referenceEvent, \_name, \_count)}

Description: This predicate applies criteria to events in the event cache to
determine if any threshold has been exceeded. It succeeds once for each threshold
that is exceeded. The name of each threshold and the number of events that
matched each threshold that is exceeded are unified with the \_name and \_count
arguments, respectively.

Arguments:

\_count The number of events that match a threshold. Must be free.
\_name The name of a threshold that is exceeded. This name was assigned with
the create\_threshold predicate.
\_referenceEvent The reference event to compare with cached events to determine if a
threshold is exceeded. This is typically the event under analysis.

Examples: The following example queries the event cache for exceeded thresholds
whenever an NT\_Performance\_Alert event is received:
\texttt{check\_all\_thresholds('NT\_Performance\_Alert', \_name, \_count)}

See Also: \texttt{check\_threshold}, \texttt{create\_threshold}
check_and_increment_count
Increments a counter and compares it to a threshold value.

Synopsis: check_and_increment_count(_key1, _key2, _max_count, _cur_count)

Description: This predicate first increments a counter and then compares the count to the _max_count argument. If the count is less than _max_count, the predicate succeeds and the count is unified with _cur_count. If the values are equal or the count is greater than _max_count, the predicate fails and _cur_count is not instantiated.

Note: Creation and initialization of a counter can be done with the init_count predicate, or it can be done with the check_and_increment_count predicate if the counter does not yet exist. If it is done with the check_and_increment_count predicate, the counter is initialized to a value of 0 and incremented by 1 in the first call. Once initialized, a counter continues counting until explicitly reinitialized with a new starting value. You must reinitialize a counter that has reached its threshold if it is still needed for counting.

Counters are used to keep track of any arbitrary numeric value. The values of _key1 and _key2 can be set to easily identify the information being recorded. For example:

- To keep track of the number of times a particular event occurs on each host, the keys could be named using an event_class,hostname scheme; thereby creating a counter for each event and each host. For example, perf_alert,orange.
- To keep track of the number of times a particular failure occurs on a set of components, the keys could be named using a failure,component scheme; thereby creating a counter for each component and each failure. For example, paper_jam,flr4rm23.

If the event server stops, all counters are discarded.

Arguments:

_cur_count
The current value the counter. Must be free.
_key1 The primary key name for the counter. Must be instantiated.
_key2 The secondary key name for the counter. Must be instantiated.
_max_count
The threshold value for the count. When this value is reached, the predicate fails. Must be instantiated.

Examples: The following example counts the number of paper jams on a set of printers, based on receiving an event class of Printer_Jam. Printer counters are identified using a failure,component scheme. Printer_Jam events identify each printer in the hostname attribute.

Each counter is created and initialized the first time the check_and_increment_count predicate is called for a particular printer. Each subsequent call for a particular printer increments its count and then compares it to the threshold value.

An administrator is notified when the number of paper jams on a printer reaches 5, and then the count for that printer is reset to 0 using the init_count predicate.
The administrator notification and reset of a count is done in an ELSE clause of a Prolog statement because the check_and_increment_count predicate behavior is failure when the count matches the threshold value.

```prolog
rule: printer_jam: {
  event : _ev of_class 'Printer_Jam'
    where [hostname: _hn within ['flr4rm23', 'flr3rm12', 'flr1rm11', 'flr6rm9'],
      action: check_count: (
        (check_and_increment_count(printer_jam,_hn,5,_count)
        % ELSE clause follows
        exec_program(_ev,'scripts/notify.sh', 'Printer failure on %s', _, no),
        init_count(paper_jam,_hn,0)
      )
    )
}.

See Also: `init_count`
```
**check_event_criteria**
Applies criteria to an event instance to determine if the instance meets the criteria.

**Synopsis:** `check_event_criteria(criteria_name, event)`

—OR—

`check_event_criteria(criteria_names_list, operator, event)`

**Description:** This predicate applies an event criteria created with the `create_event_criteria` predicate to an event. If the criteria matches, the predicate succeeds.

The second form of the predicate lets you specify multiple event criteria to an event instance.

**Arguments:**

`criteria_name`  
The name of event criteria to apply. This name was assigned with the `create_event_criteria` predicate.

`criteria_names_list`  
The names of event criteria to apply, in list format. These names were assigned with the `create_event_criteria` predicate. This argument is specified in list format; for example, `[criteria1, criteria2]`.

`event`  
The name of the event class to apply the criteria.

`operator`  
Specifies how to apply the criteria when a list of named criteria is given. Valid values are:

- **all**  
The predicate succeeds only if all of the named criteria are satisfied; that is, if the criteria defined in both criteria1 and criteria2 of the previous example are satisfied, the predicate succeeds.

- **any**  
The predicate succeeds if any one of the named criteria is satisfied; that is, if only the criteria defined in criteria2 of the previous example is satisfied, the predicate succeeds.

**Examples:**

1. The following example rule checks every incoming event for either a heartbeat or maintenance type of event. The event criteria for these events were defined with the `create_event_criteria` predicate and assigned the names `harmless_heartbeat` and `harmless_maintenance`, respectively. If it is either type of event, it is discarded and rule evaluation for the event proceeds with the next rule set.

   ```
   rule: filter_event: {
     event: _event of_class _class where [ ],
     reception_action: check_criteria: {
       check_event_criteria([harmless_heartbeat, harmless_maintenance], any, _event),
     },
   }
   ```
2. The following example applies the criteria defined in example 2 on page 130 for the `create_event_criteria` predicate to the event under analysis.

```ruby
check_event_criteria('db_critical', _ev)
```

See Also: `create_event_criteria`
check_threshold
Applies criteria to events to determine if a specific threshold is exceeded.

Synopsis: check_threshold(threshold_criteria_name, _referenceEvent, _count)

Description: This predicate applies criteria to events in the event cache to
determine if a threshold has been exceeded. The following algorithm is used:
1. A check is done to evaluate whether the maximum reporting frequency has
   been exceeded since the last time the threshold was reported. This value was
   set with the _max_report_freq argument of the create_threshold predicate.
2. If the maximum reporting frequency has not been exceeded, the
   check_threshold predicate fails. If the maximum reporting frequency value has
   been exceeded, a query is made of the event cache. The query searches for all
   events:
   a. That meet the event cache search criteria (as defined with the
      create_cache_search_criteria predicate).
   b. Whose reception time is within the time window of threshold criteria (as
      defined with the _window argument of the create_threshold predicate).
   c. That exceed the threshold specified for the threshold criteria (as defined
      with the _count argument of the create_threshold predicate).
3. If a matching event is found, the check_threshold predicate succeeds and the
   _check_count variable is unified with the number of matching events.

The not operator can be used to reverse the test to check if a threshold was not
exceeded.

Arguments:
_count  The number of events that match the threshold criteria. Must be free.
_referenceEvent  The reference event to compare with cached events to determine if a
 threshold is exceeded. This is typically the event under analysis.
threshold_criteria_name  The name of the threshold criteria to apply. This name was assigned with
 the create_threshold predicate.

Examples: The first part of the following example shows how to define threshold
criteria with the create_threshold predicate. Its characteristics are:
• The name of the threshold criteria is db_critical_threshold.

Note: A create_threshold predicate for a criteria should only be called once in a
rule that is triggered by a TEC_Start event.
• The name of the event cache search criteria defined by the
  create_cache_search_criteria predicate is db_critical_search.
• The time window for the threshold is 600 seconds surrounding the reference
  event.
• The threshold is 3 occurrences of the event within the time window
• The threshold can be exceeded 300 times before it is reported again as
  exceeded.

The second part of the following example shows how to check threshold criteria
with the predicate using the criteria defined in the first part. If 3 events that match
the search criteria defined in db_critical_search are found within a 10 minute
window of the reception time of the reference event _event, the check_hold predicate succeeds and will not succeed again for at least another 5 minutes.

create_threshold('db_critical_threshold',
    'db_critical_search',
    600,
    3,
    300)
% Define the threshold criteria.
check_threshold('db_critical_threshold',
    _event,
    _count)
% Apply the threshold criteria to received event.

See Also: check_all_thresholds, create_cache_search_criteria, create_threshold
clear_closed_events
Clears closed events from the event cache.

Synopsis:  clear_closed_events

Description:  This predicate removes closed events from the event cache.

Arguments:  None.

Examples:  The following example prints the contents of the event cache before and after clearing closed events:
           print_cache('/tmp/before'),
           clear_closed_events,
           print_cache('/tmp/after')

See Also:  None.
clears
Defines a clearing event for one or more events in an event sequence.

Synopsis: clears(class, [attribute_conditions], [target_events], [target_attribute_conditions])

—OR—
clears(class, [attribute_conditions], [target_events], [target_attribute_conditions], [attribute_exceptions])

—OR—
clears(class, [attribute_conditions], [target_events], [target_attribute_conditions], create_reverse_lookup)

—OR—
clears(class, [attribute_conditions], [target_events], [target_attribute_conditions], [attribute_exceptions], create_reverse_lookup)

Description: This predicate is used to define a clearing event for one or more events in a sequence and specify which events it clears. It must be called from within the event_details argument of the create_event_sequence predicate.

If the target attribute conditions are already defined with the attribute_conditions argument of the create_event_sequence predicate, the target_attribute_conditions argument should be an the empty list.

Note: This is a convenience predicate to define clearing events in an event sequence. Any clearing event can be defined with the create_clearing_event predicate, even if the events it clears are specified in one or more event sequences.

Arguments:
attribute_conditions
The list of conditions for attributes that must be met by an event to be considered a clearing event. There are two types of conditions, defined as follows:

absolute
A condition that can be placed upon an attribute, similar to an attribute condition in an event filter. For example, if an event of a particular class is designated as a clearing event only if its severity is HARMLESS, then the list of conditions should include [severity,equals,'HARMLESS'].

Note: Failure to specify absolute conditions can result in problem events being cleared by other problem events rather than clearing events.

See the attribute_conditions argument description for the create_event_criteria on page 129 for additional details about specifying attribute conditions for this argument, as the syntax is the same.
attribute-match
Names of attributes that must match between correlated events. You should always define at least one attribute-match condition to ensure correlation only between events of the same system. For example, [hostname].

class
The event class of the clearing event.

create_reverse_lookup
Determines whether a reverse lookup record is created for this clearing event. A reverse lookup record enables a rule to search for a problem event’s clearing event when the problem event is received (as opposed to the clearing event triggering a search for its problem events, as is typically the case). Valid values for this argument are yes or no. The default is no if this argument is not specified.

target_attribute_conditions
The list of conditions for attributes that must be met by the events listed in the target_events argument for them to be cleared by the clearing event. These conditions should not be defined if they are defined with the attribute_conditions argument of the create_event_sequence predicate.

target_events
The list of event classes that an event of class class clears.

Examples: The following example creates an event sequence for Compaq physical drive events. These events are all of the same class (cpqTape3PhDrvStatusChange). The cpqTapePhyDrvCondition attribute value changes to indicate status, with a value of OK signifying a clearing event.

The clears predicate defines clearing events as class cpqTape3PhyDrvStatusChange with attribute cpqTapePhyDrvCondition set to a value of OK.

create_event_sequence(
  ['cpqTape3PhyDrvStatusChange'],
  ['hostname', ['status','outside',['CLOSED']]]
  attr_sequence(
    'cpqTape3PhyDrvStatusChange',
    'cpqTapePhyDrvCondition'=['Degraded','Failed']),
  clears(
    'cpqTape3PhyDrvStatusChange',
    [ ['cpqTapePhyDrvCondition',equals,'OK'] ],
    [ ])
)

See Also: create_clearing_event, create_event_sequence, is_clearing_event
**commit_action**
Controls execution within the actions of the current rule.

**Synopsis:** commit_action

**Description:** This predicate prevents analysis of any further solutions for previous predicates in the current rule action and prevents the execution of any further actions in the current rule.

**Arguments:** None.

**Examples:** The following example shows that the use of the all_instances predicate might have multiple solutions, but only the first solution is used to perform the action. Any actions that follow in the current rule are not performed; however, other rules and rule sets will still evaluate the event under analysis.

```plaintext
reception_action: action1:
  (all_instances(_event,
    event:_dup_down_ev
    where [
      status: outside ['CLOSED'],
      event - 600 - 600
    ],
    add_to_repeat_count(_dup_down_ev, 1),
    drop_received_event,
    commit_action
  ),
```

**See Also:** None.
**commit_rule**
Controls execution within a rule set.

**Synopsis:** commit_rule

**Description:** This predicate prevents analysis of any further solutions for previous predicates in the current rule action and prevents the execution of any further actions in the current rule. Additionally, it prevents analysis of any following rules in the current rule set for the event under analysis.

**Arguments:** None.

**Examples:** The following example shows that the all_instances predicate might have multiple solutions, but only the first solution is used to perform the rule action. Any actions that follow in the current rule are not performed and any rules that follow the current rule in the current rule set will not evaluate the event under analysis; however, rules in subsequent rule sets will evaluate the event under analysis.

reception_action:

```prolog
{ all_instances(_event,
                   event:_dup_toner_ev
                   where [status: outside ['CLOSED']
                          ,_event = 600 - 600]
                   ,
                   add_to_repeat_count(_dup_toner_ev, 1),
                   drop_received_event,
                   commit_rule
                   ),
See Also: None.
commit_set
Controls execution within a rule base.

Synopsis:  commit_set

Description:  This predicate prevents the following:
  • Analysis of any further solutions for previous predicates in the current rule
    action
  • Execution of any further actions in the current rule
  • Evaluation by any following rules in the current rule set for the event under
    analysis
  • Evaluation by any rules in any following rule sets for the event under analysis

Arguments:  None.

Examples:  The following example shows that the all_instances predicate might
  have multiple solutions, but only the first solution is used to perform the rule
  action. Any actions that follow in the current rule are not performed and any rules
  that follow the current rule in the entire rule base will not evaluate the event
  under analysis.

reception_action:
  (all_instances(_event,
    event: _dupper
    where [
      status: outside ['CLOSED']
    ]),
   drop_received_event,
   add_to_repeat_count(_dupper, 1),
   commit_set
  ),

See Also:  None.
**convert_ascii_time**
Converting a time structure to a string.

**Synopsis:** `convert_ascii_time(_time_structure, _time_string)`

**Description:** This predicate converts a time structure to an atom representing the time. `_time_structure` must be instantiated before calling `convert_ascii_time`. `_time_string` must be free.

**Arguments:**

- `_time_string`
  The atom representation of `_time_structure`.

- `_time_structure`
  Represents an internal time structure. Do not confuse it with the data returned by the `get_time` predicate, in which the value for the `_time_epoch` argument is a number representing how many seconds have passed since an epoch.

**Examples:** The following example shows how to get the current time structure from the local system, convert the time structure to a string, and update the `time_string` attribute of the event with the string.

```
get_local_time(_time_local_struct),
convert_ascii_time(_time_local_struct, _time_string),
bo_set_slotval(_event, time_string, _time_string)
```

**See Also:** `bo_set_slotval`, `get_local_time`
**convert_gm_time**

Converts an epoch time number to a time structure in Greenwich mean time (GMT).

**Synopsis:** `convert_gm_time(_time_epoch, _time_gm_struct)`

**Description:** This predicate converts an epoch time number to a time structure in GMT. `_time_epoch` must be instantiated before calling `convert_gm_time`. `_time_gm_struct` must be free.

**Arguments:**

- `_time_epoch`
  
  Represents an epoch time number. Do not confuse it with the data returned by the `get_local_time` predicate, in which the value for the `_time_local_struct` argument is a time structure.

- `_time_gm_struct`
  
  Represents a time structure in GMT. Do not confuse it with the data returned by the `get_time` predicate, in which the value for the `_time_epoch` argument is a number representing how many seconds have passed since an epoch.

**Examples:** The following example shows how to:

1. Get the epoch time number.
2. Update the `time_epoch` attribute for the event.
3. Convert the epoch time number to a GMT structure.
4. Convert the GMT structure to a string.
5. Update the `time_string` attribute of the event with the string.

The `time_epoch` attribute could then be used for comparisons and the `time_string` attribute could be used for viewing by a person.

```plaintext
get_time(_time_epoch),
bo_set_slotval(event, time_epoch, _time_epoch),
convert_gm_time(_time_epoch, _time_gm_struct),
convert_ascii_time(_time_gm_struct, _time_string),
bo_set_slotval(event, time_string, _time_string)
```

**See Also:** `bo_set_slotval`, `convert_ascii_time`, `get_time`
convert_local_time
Converting an epoch time number to a time structure in local system time.

**Synopsis:** convert_local_time(_time_epoch, _time_local_struct)

**Description:** This predicate converts an epoch time number to a time structure in local system time. _time_epoch must be instantiated before calling convert_local_time. _time_local_struct must be free.

**Arguments:**

_**_time_epoch
Represented an epoch time number. Do not confuse it with the data returned by the get_local_time predicate, in which the value for the _time_local_struct argument is a time structure.

_**_time_local_struct
Represents a time structure in local system time. Do not confuse it with the data returned by the get_time predicate, in which the value for the _time_epoch argument is a number representing how many seconds have passed since an epoch.

**Examples:** The following example shows how to:

1. Get the epoch time number
2. Update the time_epoch attribute for the event.
3. Convert the epoch time number to a local system time structure.
4. Convert the local system time structure to a string.
5. Update the time_string attribute of the event with the string.

The time_epoch attribute could then be used for comparisons and the time_string attribute could be used for viewing by a person.

get_time(_time_epoch),
bo_set_slotval(_event, time_epoch, _time_epoch),
convert_local_time(_time_epoch, _time_local_struct),
convert_ascii_time(_time_local_struct, _time_string),
bo_set_slotval(_event, time_string, _time_string)

**See Also:** bo_set_slotval, convert_ascii_time, get_time
create_cache_search_criteria
Defines a named search for an event cache query.

Synopsis: create_cache_search_criteria(search_name, criteria_name, attributes, dup_detect)

—OR—

create_cache_search_criteria(search_name, criteria_name, attributes, dup_detect, returnOrder)

Description: This predicate defines named searches for querying the event cache. It is used in conjunction with:
- The search_cache predicate, which is the call to perform the actual query
- The create_event_criteria predicate, which creates the criteria for locating events in the event cache

The second form of the predicate lets you specify a logical order for returning events when multiple events were defined in the event criteria. For example, if order was not defined and the event classes were specified in the event criteria as [A,B,C,D,E], the events found in the cache would be returned as they were located during the search; that is, if a C event was found first because it was the most recent, it would be returned first, if an E event was then found, it would be returned next, and so forth. If you specified to return the events in order, all class A events would be returned as they were found, then all class B events would be returned as they were found, and so forth. Specifying order of return can make it easier for you to develop rules that identify the root cause of a problem.

This predicate should be run in a rule triggered by a TEC_Start event at event server start-up time. This loads the named search once, instead of every time it is needed.

Arguments:
attributes
The attribute names to match with those of the reference event, typically the event under analysis. This argument is specified in list format; for example, [hostname,severity].

criteria_name
The name that uniquely identifies the event class and attribute conditions for the search. The criteria must be created with the create_event_criteria predicate.

dup_detect
Indicates whether to apply duplicate detection as a condition of the query. Valid values are:
no Do not apply duplicate detection as a condition of the query.
es Apply duplicate detection as a condition of the query.

return_order
Indicates whether to return events in the class order defined in the event criteria for the search. If this argument is not specified, the default is random. Valid values are:
order Return events in the class order defined in the event criteria for the search
random
Return events in the order found.

search_name
The name to uniquely identify the search. This name is used by the search_cache predicate.

Examples: The following example creates a search named db_critical_search, which uses the event criteria named db_critical. This search finds any event that passes the db_critical event criteria and whose hostname attribute is the same value as the reference event. The return_order argument has been omitted and therefore defaults to random, meaning events in the list are returned in the order found during the search.

create_cache_search_criteria('db_critical_search',
    'db_critical',
    ['hostname'],
    yes
)

See Also: create_event_criteria search_cache
create_clearing_event
Defines a clearing event.

Synopsis: create_clearing_event(class, [attribute_conditions], [target_events], [target_attribute_conditions])

—OR—

create_clearing_event(class, [attribute_conditions], [target_events], [target_attribute_conditions], [attribute_exceptions])

—OR—

create_clearing_event(class, [attribute_conditions], [target_events], [target_attribute_conditions], create_reverse_lookup)

—OR—

create_clearing_event(class, [attribute_conditions], [target_events], [target_attribute_conditions], [attribute_exceptions], create_reverse_lookup)

Description: This predicate is typically used to define a clearing event for one or more problem events that are not part of an event sequence.

This predicate can be used to define any clearing event, even if the events it clears are specified in one or more event sequences.

Arguments:

attribute_conditions
The list of conditions for attributes that must be met by an event to be considered a clearing event. There are two types of conditions, defined as follows:

absolute
A condition that can be placed upon an attribute, similar to an attribute condition in an event filter. For example, if an event of a particular class is designated as a clearing event only if its severity is HARMLESS, then the list of conditions should include 
[[severity,equals,’HARMLESS’]].

Under typical circumstances, you only need to specify absolute conditions. Attribute-match conditions are evaluated only in the extremely rare case of when clearing events have been received before problem events arrive and a reverse lookup is needed for correlation.

Note: Failure to specify absolute conditions can result in problem events being cleared by other problem events rather than clearing events.

See the attribute_conditions argument description for the ["create_event_criteria" on page 129] for additional details about specifying attribute conditions for this argument, as the syntax is the same.

attribute-match
Names of attributes that must match between correlated events; for
example, [hostname]. These conditions are evaluated only in the extremely rare case of when clearing events have been received before problem events arrive and a reverse lookup is needed for correlation.

**attribute_exceptions**

Identifies an attribute that must match an attribute with a different name in other events in an event sequence. This argument enables event correlation between events containing attributes with different names but the same meaning. For example, Tivoli Distributed Monitoring might assign the name of an affected machine to the probe_arg attribute of an event.

Other monitoring systems might send events containing the name of an affected machine in the hostname attribute. To correlate these events based on machine names, map the two attributes to each other using the `attribute_exceptions` argument.

The `attr_exception` rule language predicate is used to defined the `attribute_exceptions` argument. See “`attr_exception`” on page 87 for additional details.

**class**

The event class of the clearing event.

**create_reverse_lookup**

Determines whether a reverse lookup record is created for this clearing event. A reverse lookup record enables a rule to search for a problem event’s clearing event when the problem event is received (as opposed to the clearing event triggering a search for its problem events, as is typically the case). Valid values for this argument are yes or no. The default is no if this argument is not specified.

**target_attribute_conditions**

The list of conditions for attributes that must be met by the events listed in the `target_events` argument for them to be cleared by the clearing event. There might be absolute conditions specified, but there should at least be on attribute-match condition specified to ensure that clearing events only clear problem events from the same system.

**target_events**

The list of event classes that an event of class `class` clears.

**Examples:**

1. The following example defines a clearing event for a CiscoLinkDown event, assuming that the value of the origin attribute is the same between the two events. A reverse-lookup record is not created because it is not possible to receive a CiscoLinkUp event before a corresponding CiscoLinkDown event.

   ```python
   create_clearing_event('CiscoLinkUp', [], ['CiscoLinkDown'], ['origin'], no)
   ```

2. The following example defines a clearing event for a cpqDa3PhyDrvStatusChange event. The cpqDaPhyDrvStatus attribute of this event can have a value of Fail to indicate a problem or a value of OK to clear a problem, assuming the value of the hostname attribute is the same between the two events.
The event class of the clearing event is also specified in the `target_events` argument because this clearing event clears events of the same class but with a different attribute value.

```python
create_clearing_event(
    'cpqDa3PhyDrvStatusChange',
    [[['cpqDaPhyDrvStatus', 'equals', 'OK']],
     ['cpqDa3PhyDrvStatusChange'],
     [['hostname'],
      [['cpqDaPhyDrvStatus', 'not_equal', 'OK'],
      no],
     no),
```

See Also: clears, create_event_sequence, is_clearing_event
create_event_criteria
Specifies criteria for an event.

Synopsis: create_event_criteria(criteria_name, class, fire_on_non_leaf, attribute_conditions)

Description: This predicate defines criteria used for determining the state of an event. Criteria are used where needed in any rule action by the check_event_criteria predicate, and are also used by the create_cache_search_criteria predicate.

This predicate should be run in a rule triggered by a TEC_Start event at event server start-up. This loads the criteria once, instead of every time it is needed.

The following checks are done at runtime to this predicate. Errors can be obtained with the log_error predicate, which is described on page 180.
- The event class is checked for existence.
- All attribute conditions are checked thoroughly.

Arguments:
attribute_conditions
Specifies the attribute conditions for the reference event. Each attribute condition is defined as a list with three elements. The attribute_conditions argument is a list of attribute conditions, which means it is a list of lists. For example, the following figure illustrates the format of this argument if two attribute conditions are defined:

```
[['attribute', operator, 'value'],
 ['attribute, operator, 'value']]`
```

Notes:
1. The attribute and operator must be compatible. For example, you cannot create a condition for an attribute defined as a STRING type with a greater_than operator. See the following table for attribute-operator compatibility.
2. Attribute conditions can only be defined for attributes of a SINGLE complexity type.
3. The matches operator requires the same Perl regular expression syntax as that supported by the Tivoli Management Framework.
4. ENUM types are evaluated arithmetically based on their integer representation.
5. Attributes of a LIST_OF complexity type are not supported in an attribute condition.

The following table lists the operators you can use for each SIMPLE type of attribute. The operators used in this predicate are similar to those used in an event filter. As such, you might find the information in "Attribute conditions" on page 63 helpful.

<table>
<thead>
<tr>
<th>Simple Type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENUM, INTEGER, REAL</td>
<td>equals greater_than greater_than_equal</td>
</tr>
<tr>
<td></td>
<td>less_than less_than_equal not_equals</td>
</tr>
<tr>
<td></td>
<td>outside within</td>
</tr>
<tr>
<td>STRING</td>
<td>equals not_equals matches outside within</td>
</tr>
</tbody>
</table>
**class**  
The event classes for which the attribute conditions are defined. This argument is specified in list format; for example, ['NT_SNMP', 'NT_Server_Start'].

The order of event classes in the list is significant if you use the `return_order` argument with a value of order in the `create_cache_search_criteria` predicate when defining a search using this event criteria. See the "create_cache_search_criteria" on page 124 for additional information.

**criteria_name**  
The name that uniquely identifies the criteria.

**fire_on_non_leaf**  
Specifies whether the criteria can be used for executing rules on the reference event if it is a superclass. The following values are valid:

- **no**  
The criteria can be used for executing rules on leaf classes only.

- **yes**  
The criteria can be used for executing rules on both leaf and non-leaf classes.

**Examples:**

1. The following example shows how to define event cache search criteria with the predicate. The name of this criteria is `example_criteria`. This is the name referred to from the `create_cache_search_criteria` predicate. This criteria is used for a TEC_DB event and it can only be used for a leaf class.

   **Note:** This example is not a realistic definition. It is simply intended to show the various ways you can define attribute conditions.

   ```
   create_event_criteria(example_criteria,
   'TEC_DB',
   no,
   [[‘hostname’, equals, ‘chair’],
    [‘hostname’, not_equals, ‘chair1’],
    [‘hostname’, matches, ‘ch.*r’],
    [‘repeat_count’, within, [5]],
    [‘repeat_count’, outside, [10,15]],
    [‘repeat_count’, equals, 5],
    [‘repeat_count’, not_equals, 6],
    [‘repeat_count’, greater_than, 4],
    [‘repeat_count’, greater_than_equal, 5],
    [‘repeat_count’, less_than, 6],
    [‘repeat_count’, less_than_equal, 5],
    [‘severity’, within, [‘MINOR’]],
    [‘severity’, outside, [‘FATAL’,‘HARMLESS’]],
    [‘severity’, equals, ‘MINOR’],
    [‘severity’, not_equals, ‘FATAL’],
    [‘severity’, greater_than, ‘HARMLESS’],
    [‘severity’, greater_than_equal, ‘MINOR’],
    [‘severity’, less_than, ‘CRITICAL’],
    [‘severity’, less_than_equal, ‘CRITICAL’]
   ),
   ```

2. The following example creates event criteria named `db_critical`. This criteria applies to a TEC_DB event sent from a database server with a severity greater than or equal to CRITICAL. The severity attribute is defined as an ENUM type with a default value of 60. This example assumes all database server names begin with DB_SRV followed by other characters.
create_event_criteria('db_critical',
    'TEC_DB'
    yes,
    [['hostname',matches, '
        'DB_SRV*']],
    ['severity',
        greater_than_equal, '
        'CRITICAL']
    )

3. The following example creates event criteria named ups_problem. This criteria applies to any upsOnBattery, upsBatteryLow, or upsBatteryDischarged event from host homer.

create_event_criteria('ups_problem',
    ['upsOnBattery',
        'upsBatteryLow',
        'upsBatteryDischarged'],
    yes,
    [['hostname',equals,'homer']]
    )

See Also: check_event_criteria, create_cache_search_criteria
create_event_sequence
Defines a sequence of events for correlation.

Synopsis:  create_event_sequence([event_sequence], [attribute_conditions])
—OR—
create_event_sequence([event_sequence], [attribute_conditions], [event_details])

Description: This predicate is used to define a sequence of events, and any additional information pertaining to those events, that make up an event sequence. An event sequence is a series of events (generally causally related) that occur in a fixed order.

This information is loaded into the knowledge base of the rule engine at event server start-up and is used by rules that call correlation predicates. You can load it with a rule triggered by a TEC_Start event at event server start-up time.

Arguments:
attribute_conditions
The list of conditions for attributes that must be met by both events in a sequence (the event under analysis and the event being searched for in the cache) to be eligible for correlation. There are two types of conditions, defined as follows:

absolute
A condition that can be placed upon an attribute, similar to an attribute condition in an event filter. For example, [severity,equals,'HARMLESS']. This type of condition applies to all events in the event sequence. Any absolute conditions that apply to only a subset of events in an event sequence must be specified with the attr_condition predicate, which is described on page 85.

See the attribute_conditions argument description for the "create_event_criteria" on page 129 for additional details about specifying attribute conditions for this argument, as the syntax is the same.

attribute-match
Names of attributes whose values must match between correlated events. You should always define at least one attribute-match condition to ensure correlation only between events of the same system. For example, [hostname]. This type of condition applies to both events that are being correlated when using the attr_exception predicate in this argument. The attr_exception predicate is described on page 87.

event_details
The list of predicates that provides additional details about individual events in the event sequence, including identifying clearing events. The predicates that can be specified are shown in the following table.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_condition</td>
<td>Defines absolute attribute conditions for a single event in an event sequence.</td>
</tr>
</tbody>
</table>
**Predicate** | **Description**
--- | ---
attr_exception | Defines an attribute that must match a different attribute in other events in an event sequence.
attr_sequence | Defines the values of an attribute that change due to a problem event’s position in an event sequence.
clears | Defines a clearing event for events in an event sequence.

event_sequence
The list of event class names in event-sequence order, from left to right. For example, ['upsOnBattery', 'lowBattery', 'upsDischarged'].

**Examples:**
1. The following example defines an event sequence with clearing events. The sequence contains events generated from two monitoring sources: APC uninterruptible power supply and IBM Tivoli Distributed Monitoring. The uninterruptible power supply event sequence is illustrated in the figure on page 218.

The problem events are specified in event-sequence order, from left to right, the root cause being the upsOnBattery event. The last event in the sequence (universal_host) is generated by Tivoli Distributed Monitoring. Each of the problem events in the sequence has a related clearing event defined with the clears predicate. The uninterruptible power supply events are related if the hostname attributes have the same value.

The Tivoli Distributed Monitoring universal_host event is handled a little differently because the value for the affected host is stored in the probe_arg attribute (rather than the hostname attribute like the uninterruptible power supply events). This requires mapping of the probe_arg attribute to the hostname attribute of the uninterruptible power supply events so correct comparisons can be made. A situation like this is referred to as an attribute exception and requires the use of the attr_exception predicate, which is called from the event_details argument of the create_event_sequence predicate.

A universal_host event with a severity of FATAL is generated when a host is unavailable. When the host is available again, the same event is sent with a severity of HARMLESS. This situation means that two events of the same class are differentiated by the value of an attribute. This requires the use of the attr_condition predicate in the event_details argument of the create_event_sequence predicate to define that a universal_host event is only to be correlated with an uninterruptible power supply event if its severity is FATAL. Furthermore, a universal_host event is a clearing event only if its severity is HARMLESS.

The attribute_conditions argument for the first three clears predicates in the example are empty lists because the clearing events are event classes and attribute conditions are not needed. Their hostname attributes must match because hostname is specified in the attribute_conditions argument for the create_event_sequence predicate.

The attribute_conditions argument for the fourth clears predicate places a condition on the universal_host event’s severity attribute for the value being equal to HARMLESS for defining a clearing event. The specification of the hostname attribute in the attribute_conditions argument for the create_event_sequence predicate and the attr_exception predicate called from
the `event_details` argument of the `create_event_sequence` predicate ensure that the host names match between clearing and problem events.

```ruby
create_event_sequence(
    ['upsOnBattery',
     'lowBattery',
     'upsDischarged',
     'universal_host'],
    ['hostname', ['status', 'outside', ['CLOSED']]],
    [clears('powerRestored', [], ['upsOnBattery']),
     clears('returnFromLowBattery', [], ['lowBattery'],
     clears('dischargeCleared', [], ['upsDischarged']),
     clears('universal_host',
     attr_condition('universal_host',
     ['severity', equals, 'HARMLESS'])),
     attr_exception('hostname', 'universal_host',
     'probe_arg')]
)
```

2. The following example defines an event sequence with clearing events. This sequence contains events generated from Compaq Insight Manager related to physical drive problems. This event sequence is illustrated in the figure on page 218.

This monitor generates events of the same class, using the `cpqTapePhyDrvCondition` attribute to indicate status. This attribute with a value of OK designates a clearing event. To handle this type of event sequence, the attribute name and the sequence of values (in event-sequence order from left to right) it can assume must be defined with the `attr_sequence` predicate, which is called from the `event_details` argument of the `create_event_sequence` predicate.

The `event_sequence` argument only defines one class because this one event comprises the entire event sequence. Status is dependent upon the `cpqTapePhyDrvCondition` attribute.

In the clears predicate, a `target_attribute_conditions` argument is not needed because the conditions for the problem events are already defined with the `attr_sequence` predicate.

```ruby
create_event_sequence(
    ['cpqTape3PhyDrvStatusChange'],
    ['hostname', ['status', 'outside', ['CLOSED']]],
    [attr_sequence('cpqTape3PhyDrvStatusChange',
     'cpqTapePhyDrvCondition'=['Degraded','Failed']),
     clears('cpqTape3PhyDrvStatusChange',
     [ ['cpqTapePhyDrvCondition',equals,'OK'] ],
     ['cpqTape3PhyDrvStatusChange'],
     [ ])
)
```

3. This example is based upon the event sequence defined in the following figure. It defines an event sequence with sequences that branch. This means that two or more sequences merge or emerge from a single sequence. In this example, the `cpqHe3ThermalSystemFan` and the `cpqHe3ThermalCpuFan` type of events
merge with the cpqHe3ThermalTemp event type sequence. This event sequence contains events generated from Compaq Insight Manager that are related to temperature problems. Root cause events are the darkest shade, clearing events are the next lightest, and effect events are the lightest.

The create_event_sequence predicate can be used to define this type of event sequence by specifying one complete sequence with subsequent sequences that include at least one of the events that all of the related sequences have in common.

If a sub-sequence branches from the complete sequence and then reconnects later in the sequence, both the event where it branched and the event where it reconnected must be specified. You should specify the complete sequence first and then specify sub-sequences that branch from or connect to it; this approach will make it easier for you to write the predicates.

The following two create_event_sequence predicates completely define the event sequence shown in the flowchart for this example. The cpqHe3ThermalTempDegraded event in the second predicate specifies that the cpqHe3ThermalCpuFanFailed event joins the sequence defined in the first predicate.

```prolog
create_event_sequence(
  ['cpqHe3ThermalSystemFanDegraded',
   'cpqHe3ThermalSystemFanFailed',
   'cpqHe3ThermalTempDegraded',
   'cpqHe3ThermalTempFailed'],
  [hostname, ['status', equals, 'OPEN']],
  [
    clears('cpqHe3ThermalSystemFanOK', [], [
      ['cpqHe3ThermalSystemFanDegraded'], [],
   ]),
   ...
  ]
)
```
clears('cpqHe3ThermalTempOK',
  [],
  ['cpqHe3ThermalTempDegraded'],
  []),
clears('cpqHe3ThermalConfirmation',
  [],
  ['cpqHe3ThermalTempFailed'],
  [])
],
create_event_sequence(
  ['cpqHe3ThermalCpuFanFailed',
   'cpqHe3ThermalTempDegraded'],
  [hostname, ['status', equals, 'OPEN']]
],
  clears('cpqHe3ThermalCpuFanOK',
    [],
    ['cpqHe3ThermalSystemFanFailed'],
    []
  ),

See Also: None.
**create_threshold**

Defines a threshold.

**Synopsis:** `create_threshold(threshold_criteria_name, cache_search_criteria_name, _window, _count, _max_report_frequency)`

**Description:** This predicate defines the criteria for setting a threshold for events in the event cache. It is used in conjunction with the `check_threshold` predicate. It should be run in a rule triggered by a TEC_Start event at event server start-up time. This loads the criteria once.

**Arguments:**

- `_count` Specifies the threshold. For example, a threshold of 5 means that when the sixth matching event is found, the threshold has been exceeded.

- `_max_report_frequency` Specifies the time, in seconds, that a threshold has to remain exceeded before the threshold is reported again as exceeded.

- `_window` Specifies the time window, in seconds, to count events that match the search criteria towards the threshold. The time window is based upon the reception time of the reference event, which is typically the event under analysis. The time window spans the number of seconds before and after the reference event. For example, a time window of 600 seconds (10 minutes) means that events matching the search criteria received 5 minutes before or 5 minutes after the reference event are counted towards the threshold.

- `cache_search_criteria_name` The name that uniquely identifies the search criteria for a query of the event cache. This criteria was created with the `create_cache_search_criteria` predicate. The search criteria must be defined before using the `create_threshold` predicate.

- `threshold_criteria_name` The name that uniquely identifies the threshold criteria. This name is referred to from the `check_threshold` predicate.

**Examples:** The following example shows how to define threshold criteria with the predicate. Its characteristics are:

- The name of the threshold criteria is `db_critical_threshold`
- The name of the event cache search criteria defined by the `create_cache_search_criteria` predicate is `db_critical_search`
- The time window for the threshold is 600 seconds surrounding the reference event
- The threshold is 3 occurrences of the event within the time window
- The threshold must be exceeded for 300 seconds before it is reported again as exceeded.

`create_threshold('db_critical_threshold', 'db_critical_search', 600, 3, 300)`

**See Also:** `check_threshold` `create_cache_search_criteria`
**decrement_slot**
Subtracts a number from the value of the specified integer attribute.

Synopsis: `decrement_slot(_event, _attribute_name, _by_value, _trigger)`

Description: This predicate subtracts a number from the value of the specified integer attribute.

Note: Generally, the term *slot* has been replaced by the term *attribute*, even though this command name has not been changed.

Arguments:

- `_attribute_name` The attribute to change.
- `_by_value` The amount to subtract.
- `_event` A pointer to the event to change.
- `_trigger` Specifies whether change rules should be evaluated as a result of this attribute change. Valid values are: 'YES', yes, 'NO', or no.

Examples: The following example shows predicate usage:
`decrement_slot(_event, host_down, 1, no)`

See Also: [increment_slot](#)
**drop_change_request**
Prevents a change request from being applied after change rules are run.

**Synopsis:** drop_change_request

**Description:** This predicate prevents a change request from being applied after change rules are run.

**Arguments:** None.

**Examples:** The following example shows that for a request to change the status of an event to ACK or CLOSED, if the requesting user is not Root-myHost-region, the change request is dropped and the msg attribute of the event is set to a denial message. The change request is not evaluated by any other change rules after being dropped.

```plaintext
change_rule: deny_state_change_of_TTs:
    event: _event of_class _class,
    sender: _sender equals operator(_x),
    slot: status set_to _new_status within ['ACK', 'CLOSED'],
    action: (_sender \== operator('Root_myHost-region'),
        bo_set_slotval(_event,'msg','modification denied !'),
        drop_change_request
    )
).
```

**See Also:** None.
**drop_received_event**
Discards an event after the rules are run.

**Synopsis:** drop_received_event

**Description:** This predicate causes the event under analysis to be discarded after the rules are evaluated with it.

**Arguments:** None.

**Examples:** The following example rule shows how to count the number of duplicate NFS_NOT_RESPONDING events that are received and then drop them so they're not stored in the event database. This results in one event kept with its repeat_count attribute updated each time a duplicate is received.

```plaintext
rule: dup_nfs_not_resp:

  event: _event of_class 'NFS_NOT_RESPONDING',

  action: dup_and_drop_event:
  
  first_duplicate(_event,event:_dup_nfs_ev
  where [status: outside ['CLOSED'] ]
  },

  add_to_repeat_count (_dup_nfs_ev, 1),

  drop_received_event
  )


See Also: None.
**erase_globals**
Removes all the global variables in a group from the knowledge base.

**Synopsis:**  `erase_globals(group)`

**Description:**  This predicate removes all the global variables in a group from the knowledge base.

**Arguments:**

*group*

The group key whose variables to remove.

**Examples:**  The following example removes all of the global variables in the Maintenance group from the knowledge base:
```
erase_globals('Maintenance')
```

**See Also:**  None.
**exec_program**
Launches a program.

**Synopsis:** `exec_program(_event, file_name, _format_string, _arg_list, watch_status)`

**Description:** This predicate launches a program. The program’s completion status can be monitored.

**Note:** Null arguments to exec_program might crash the event server when this predicate is run. In addition, ensure that all attributes passed to exec_program are instantiated.

**Arguments:**

_**arg_list**_.
A list of values (typically attributes of the event) to be supplied to the program in the form `[1, 2, 3]`. All of the attributes of the trigger event are also available to the program through environment variables; for example, the msg attribute value can be obtained from the $msg environment variable. See the *IBM Tivoli Enterprise Console Command and Task Reference* for additional information about environment variables available to running tasks and programs.

For every format specification in the format string, there must be a corresponding element in the argument list. The data types in the format string must be compatible with their corresponding values in the argument list. If there are no format specifications in the format string, the argument list must be an empty list, written as `[ ]`. The length of a formatted command line is limited to 256 characters.

_**event**_.
A pointer to the event that triggers running of the program. All attributes of this event are available to the program as environment variables. See the *IBM Tivoli Enterprise Console Command and Task Reference* for additional information about environment variables available to running tasks and programs.

_**format_string**_.
The format string for formatting arguments to the command. `%s` (STRING), `%d` (INTEGER), and `%ld` (INT32) format specifications can be defined in the format string for use with the corresponding values in the argument list.

_**file_name**_.
The path and file name of the program to run. Relative paths can be specified from the $BINDIR/TME/TEC directory.

_**watch_status**_.
Specifies whether program execution should be monitored. The `watch_status` argument can be ‘YES’ or ‘NO’. This argument must be enclosed in single quotation marks. If ‘YES’, the completion status command can be checked from the event console.

**Examples:** The following example shows a use of the predicate:

```plaintext
exec_program(_event,
     % Pass in the event pointer for access to
     % its environment variables.

    'scripts/send_notice',
    % Program path/name.
```
The `%s` format specifiers of the `_format_string` argument are bound to the `msg` and `severity` attributes of the event. The `send_notice` program is launched with four command line arguments, such as shown in the following example:

```bash
send_notice -m "Su to root failed for Joe" -s CRITICAL
```

Note that double quotation marks are used to delimit the value of the `msg` attribute so it is presented as a single argument to the `send_notice` command.

See Also: `exec_program_local`, `exec_task`, `exec_task_local`
**exec_program_local**
Launches a program on the local event server.

**Synopsis:**
`exec_program_local(_name, _event, file_name, format_string, _arg_list, watch_status)`

**Description:**
This predicate launches a program asynchronously on the local event server (local means the server where the rule engine is installed). When the program finishes, a TASK_COMPLETE event is generated if the `watch_status` argument is set to 'YES'. This event contains details about the program’s execution. The TASK_COMPLETE event class is defined in the tec.baroc file. A description of its attributes are as follows:

- **command**
  The name of the command to launch the program.

- **end_time**
  The time when the program finished.

- **execution_msg**
  Output from the program. This attribute contains a list of strings, each string representing a line of output from the program or script. This list is limited to 512 lines.

- **exit_status**
  The exit status set by the operating system for the program.

- **start_time**
  The time when the program started.

- **task_name**
  The name assigned to the program. It was assigned with the `_name` argument of the predicate.

- **task_number**
  An identifier for the executing program. These identifiers start at 1 and are incremented by 1 for each launch of a program.

- **task_status**
  The completion status for the program.

- **trigger_event_id**
  The identifier of the event that triggered the launch of the `exec_program_local` predicate.

Usually a pair of rules are created when using this predicate. The first rule launches the program. The second rule evaluates the results of the program when it is done and might take some action depending on the results.

**Arguments:**

- **_arg_list**
  A list of values (typically attributes of the event) to be supplied to the program in the form [1, 2, 3]. All of the attributes of the trigger event are also available to the program through environment variables; for example, the `msg` attribute value can be obtained from the `$msg` environment variable. See the IBM Tivoli Enterprise Console Command and Task Reference for additional information about environment variables available to running tasks and programs.

  For every format specification in the format string, there must be a corresponding element in the argument list. The data types in the format
string must be compatible with their corresponding values in the argument list. If there are no % format specifications in the format string, the argument list must be an empty list, written as [ ]. The length of a formatted command line is limited to 256 characters.

_event A pointer to the event that triggers running of the program. All attributes of this event are available to the program as environment variables. See the Tivoli Management Framework Reference Manual for additional information about environment variables available to running tasks and programs.

_format_string
The format string for formatting arguments to the command. %s (STRING), %d (INTEGER), and %ld (INT32) format specifications can be specified in the format string for use with the corresponding values in the argument list. If a format string is not specified, an empty _format_string argument must be specified in the form \"\" (two single quotation marks). The example in "exec_program" on page 142 shows the use of format strings.

_name The name to assign the program. It is used to identify the program in a TASK_COMPLETE event.

file_name
The path and file name of the program to run. Relative paths can be specified from the $BINDIR/TME/TEC directory.

watch_status
Specifies whether a TASK_COMPLETE event is to be generated. Valid values are:

'NO’ Do not generate a TASK_COMPLETE event when the program finishes. This argument must be enclosed in single quotation marks.

'YES’ Generate a TASK_COMPLETE event when the program finishes. This argument must be enclosed in single quotation marks.

Examples: The following example shows:

1. In the program_start rule, the ls (list) program is launched upon the reception of a TEC_DB event. The program is launched with the following characteristics:
   a. The program is given the name of list_tmpdir.
   b. There are no additional arguments for the program’s command line.
   c. A TASK_COMPLETE event is to be generated when the program finishes.

2. The program_result rule is triggered by the reception of a TASK_COMPLETE event with the task_name attribute set to list_tmpdir, which is the name of the program invoked in the previous rule.

3. The process_program_result action of the program_result rule does the following:
   a. Gets the value of the execution_msg attribute from the TASK_COMPLETE event and unifies that value with the _results variable. This attribute is a list of strings.
   b. _results is searched for an element with a value of OK.
   c. If an element with a value of OK is found in the list, the ok predicate is run. If it is not found, the not_ok predicate is run.

rule: program_start: {

  event: _event of_class 'TEC_DB'
  where [ ],
reception_action: start_it: (
    exec_program_local('lst_tmpdir',_event,'ls /tmp',
    '',[],'YES')
).

rule: program_result: (

    event: _event of_class 'TASK_COMPLETE'
    where [task_name: _task_name equals 'lst_tmpdir',
        % Test for program name. If passed, assign
        % value to variable.
        task_number: _task_num
        % Assign task_number attribute value to
        % variable.
    ],

    reception_action: process_program_result: (

        bo_get_slotval(_event,execution_msg,_results),
        % Get value of execution_msg attribute and assign to
        % variable. Attribute is a list type.

        member(_result_line,_results),
        (_result_line == 'OK' ->
            % Test each element for OK value.
            ok
            % OK value found in list. Run ok predicate.
            ; % Else.

            do_not_ok_thing
            % OK value not found in list. Run not_ok
            % predicate.)
    ).

See Also: exec_program, exec_task, exec_task_local
exec_task
Launches a task from a task library.

Synopsis: exec_task(_event, task_name, format_string, _arg_list, watch_status)

Description: This predicate launches a task from a task library. The task’s completion status can be monitored. Tasks provided by the Tivoli Enterprise Console product are described in the IBM Tivoli Enterprise Console Command and Task Reference.

Note: Null arguments to exec_task might crash the event server when this predicate is run. In addition, ensure that all attributes passed to exec_task are instantiated.

Arguments:
_arg_list
A list of values (typically attributes of the event) to be supplied to the program in the form [1, 2, 3]. All of the attributes of the trigger event are also available to the program through environment variables; for example, the msg attribute value can be obtained from the $msg environment variable. See the IBM Tivoli Enterprise Console Command and Task Reference for additional information about environment variables available to running tasks and programs.

For every format specification in the format string, there must be a corresponding element in the argument list. The data types in the format string must be compatible with their corresponding values in the argument list. If there are no format specifications in the format string, the argument list must be an empty list, written as [ ]. The length of a formatted command line is limited to 256 characters.

_event
A pointer to the event that triggers running of the program. All attributes of this event are available to the program as environment variables. See the Tivoli Management Framework Reference Manual for additional information about environment variables available to running tasks and programs.

_format_string
The format string for formatting arguments to the task. %s (STRING), %d (INTEGER), and %ld (INT32) format specifications can be specified in the format string for use with the corresponding values in the argument list. The format string contains the name of the task library, the host name where the task will run, and any command line arguments to the task in the following form:

-1 tasklibname -h hostname -a arg1 -a arg2...

The example in "exec_program" on page 142 shows the use of format strings.

Notes:
1. The –l, –h, and –a arguments in the format string are the same as those used in the Tivoli Management Framework wruntask command. See the Tivoli Management Framework Reference Manual for details.
2. The task name (–t TaskName) and pass environment variables (–E) wruntask command arguments are provided internally by the exec_task predicate.

task_name
Specifies the name of the task to run.
**watch_status**

Specifies whether task execution should be monitored. The `watch_status` argument can be 'YES' or 'NO'. This argument must be enclosed in single quotation marks. If 'YES', the completion status command can be checked from the event console.

**Examples:**

1. The following example shows that the Send_Email task from the T/EC Tasks library is launched on host stumpy. Two arguments are passed to the task: the administrator's name to appear in the To field of the note, and the administrator's e-mail address. Task completion will not be monitored. The `wruntask` command example shows how the task is actually launched with the arguments resolved.

   ```
   exec_task(_event, 'Send_Email', '-l "T/EC Tasks" -h "stumpy" -a "%s" -a "%s"', ['joe@company.com', 'joe@company.com'], 'NO')
   wruntask -t Send_Email -l "T/EC Tasks" -h "stumpy" -E -a "joe@company" -a "joe@company"
   ```

2. The following example shows how to run a task based on an event sent from Tivoli Distributed Monitoring, which is monitoring an application instance of an MS SQL database. The filtering criteria for the rule is an event of class MSSQLDatabase_LogSpacePercentUsedDB with a severity of CRITICAL. The value for the collection attribute contains the resource type being monitored. The resource type and host name are instantiated in variables for use in the `exec_task` call.

   ```
   rule: plain_rule1_42: (description:'ADSM incremental backup task',
   event: _ev1 of class within [MSSQLDatabase_LogSpacePercentUsedDB]
   where [severity: _ev1_severity
   collection: _ev1_collection,
   hostname: _ev1_hostname]
   ),
   reception_action: action0: (exec_task(_ev1, 'ADSMIncBackup', '-l MSSQLManagerTasks -h '@%s:%s'[, _ev1_collection,_ev1_hostname], 'YES')
   )
   ).
   ```

   The `exec_task` call resolves to the following command when an event is received for an MSSQLDatabase collection on host master@holon@holon:

   ```
   wruntask -t ADMIncBackup -l MSSQLManagerTasks \ 
   -h @MSSQLDatabase:master@holon@holon -E
   ```

**See Also:** exec_program, exec_program_local, exec_task_local
**exec_task_local**
Launches a task from a task library on the local event server.

**Synopsis:**  
exec_task_local( _name, _event, file_name, format_string, _arg_list,  
watch_status)

**Description:**  
This predicate launches a task asynchronously from a task library on  
the local server (local means the event server where the rule engine is installed).  
Tasks provided by the Tivoli Enterprise Console product are described in the IBM  
Tivoli Enterprise Console Command and Task Reference.

**Note:**  
This predicate can only be run on a managed node.  
When the program finishes, a TASK_COMPLETE event is generated if the  
watch_status argument is set to ‘YES’. This event contains details about the task’s  
execution. The TASK_COMPLETE event class is defined in the root.baroc file. A  
description of its attributes are as follows:

**command**  
The name of the command to launch the task.

**end_time**  
The time when the task finished.

**execution_msg**  
Output from the task. This attribute contains a list of strings, each string  
representing a line of output from the program or script. This list is limited  
to 512 lines.

**exit_status**  
The exit status set by the operating system for the task.

**start_time**  
The time when the task started.

**task_name**  
The name assigned to the task. It was assigned with the _name argument of  
the predicate.

**task_number**  
An identifier for the executing task. These identifiers start at 1 and are  
incremented by 1 for each launch of a task.

**task_status**  
The completion status for the task. These values are defined in the  
root.baroc file as RUNNING, SUCCESS, FAILURE, and UNKNOWN.

**trigger_event_id**  
The identifier of the event that triggered the launch of the exec_task_local  
predicate.

Usually a pair of rules are created when using this predicate. The first rule  
launches the task. The second rule evaluates the results of the task when it is done  
and might take some action depending on the results.

**Arguments:**

**_arg_list**  
A list values (typically attributes of the event) to be supplied to the  
program in the form [1, 2, 3]. All of the attributes of the trigger event are  
also available to the task through environment variables; for example, the  
msg attribute value can be obtained from the $msg environment variable.
See the *IBM Tivoli Enterprise Console Command and Task Reference* for additional information about environment variables available to running tasks and programs.

For every format specification in the format string, there must be a corresponding element in the argument list. The data types in the format string must be compatible with their corresponding values in the argument list. If there are no format specifications in the format string, the argument list must be an empty list, written as `[ ]`. The length of a formatted command line is limited to 256 characters.

**_event_** The pointer to the event that triggers running of the task. All attributes of this event are available to the task as environment variables. See the *Tivoli Management Framework Reference Manual* for additional information about environment variables available to running tasks and programs.

**_format_string_**

The format string for formatting arguments to the command. `%s` (STRING), `%d` (INTEGER), and `%ld` (INT32) format specifications can be defined in the format string for use with the corresponding values in the argument list. If a format string is not specified, an empty `_format_string` argument must be specified in the form “” (two single quotation marks). The format string contains the name of the task library, the host name where the task will run, and any command line arguments to the task in the following form:

`-l tasklibname -h hostname -a arg1 -a arg2...`

The example “exec_program” on page 142 shows the use of format strings.

**Notes:**

1. The –l, –h, and –a arguments in the format string are the same as those used in the Tivoli Management Framework `wruntask` command. See the *Tivoli Management Framework Reference Manual* for details.

2. The task name (`-t` *TaskName*) and pass environment variables (`-E`) `wruntask` command arguments are provided internally by the `exec_task_local` predicate.

**_name_** The name to assign the task. It is used to identify the task in a TASK_COMPLETE event.

**file_name**

The path and file name of the task to run. Relative paths can be specified from the `$BINDIR/TME/TEC` directory.

**watch_status**

Specifies whether a TASK_COMPLETE event is to be generated. Valid values are:

‘NO’ Do not generate a TASK_COMPLETE event when the task finishes. This argument must be enclosed in single quotation marks.

‘YES’ Generate a TASK_COMPLETE event when the task finishes. This argument must be enclosed in single quotation marks.

**Examples:** The following example shows:

1. In the task_start rule, the send_dbadmin task is launched upon the reception of a TEC_DB event. The program is launched with the following characteristics:

   - The task is given the name of send_dbadmin.
   - There are two arguments for the task’s command line.
• A TASK_COMPLETE event is to be generated when the task finishes

2. The task_result rule is triggered by the reception of a TASK_COMPLETE event with the task_name attribute set to send_dbadmin, which is the name of the task launched in the previous rule.

3. The process_task_result action of the task_result rule does the following:
   a. Gets the value of the execution_msg attribute from the TASK_COMPLETE event and unifies that value with the _results variable. This attribute is a list of strings.
   b. _results is searched for an element with a value of OK.
   c. If an element with a value of OK is found in the list, the ok predicate is run. If it is not found, the not_ok predicate is run.

See Also: exec_program, exec_program_local, exec_task
**first_causal_event**

Searches the event cache for the root cause event related to an effect event.

**Note:** Generally, the term *causal event* has been replaced by the term *cause event*, even though this command name has not been changed.

**Synopsis:**

```plaintext
first_causal_event(_effect_event, _cause_event)
```

—OR—

```plaintext
first_causal_event(_effect_event, _cause_event, time_before, time_after)
```

**Description:** This predicate is used to search the event cache for the root cause event in an event sequence defined with the `create_event_sequence` predicate. The event must also meet the criteria defined with the `create_event_sequence` predicate. For example, if events A, B, C, and D are defined as an event sequence in that order, and event D is event under analysis, this predicate will return the most recent instance of event A if it exists and meets the defined criteria, otherwise return event B if it exists, otherwise return event C if it exists, otherwise it fails.

If the `time_before` and `time_after` arguments are not specified, the event cache search time window defaults to 2 years (1 year before and 1 year after). You should limit a time window to the smallest reasonable window whenever possible for better performance.

**Arguments:**

- `_cause_event`
  A pointer to the root cause event found for the effect event. This argument must be free.

- `_effect_event`
  A pointer to the effect event whose cause event is being searched for. Typically the event under analysis.

- `time_after`
  The number of seconds after the effect event has been received. This argument is used to limit the event cache search to a time window.

- `time_before`
  The number of seconds before the effect event has been received. This argument is used to limit the event cache search to a time window.

**Examples:** The following example searches the event cache for a related cause event that has been previously received. If one is found, the effect event is acknowledged and linked to the cause event. This rule triggers on a superclass but it searches for a cause event. This design lets you create a single rule to process any number of events that are related.

```plaintext
rule: 'link_effect_to_cause':(
    event: _effect of_class 'EVENT',
    action: 'search_for_cause':(
        first_causal_event(_effect, _cause, 3600, 0),
        set_event_status(_effect, 'ACK'),
        link_effect_to_cause(_effect, _cause)
    )
).
```

**See Also:** `create_event_sequence` `first_effect_event` `first_related_event`
first_duplicate
Succeeds once for the first (most recent) duplicate event in the event cache that
satisfies the specified additional attribute and time window conditions.

Synopsis: first_duplicate(_event, event: _duplicate where attribute_conditions)
—OR—
first_duplicate(_event, event: _duplicate where attribute_conditions, _referenceEvent
–time_before –time_after)

Description: No class specification is required, since the duplicate events are
always of the same class. For additional information about duplicate events, see
“What is a duplicate event?” on page 72

If the –time_before and –time_after arguments are not specified, the event cache
search time window defaults to 2 years (1 year before and 1 year after). You should
limit a time window to the smallest reasonable window whenever possible for
better performance.

Arguments:
_event  A pointer to the event currently under analysis.
_referenceEvent
    A pointer to the reference event for the time window, typically the event
    under analysis.

event: _duplicate where attribute_conditions
    Specifies an event filter for querying the event cache. _duplicate is
    instantiated with a pointer to each duplicate event found. See ”Event
    filters” on page 60 for additional information.

–time_after
    The number of seconds after the reference event.

–time_before
    The number of seconds before the reference event.

Examples: The following example rule shows how to count the number of
duplicate NFS_NOT_RESPONDING events that are received and then drop them
so they’re not stored in the event database. This results in one event kept with its
repeat_count attribute updated each time a duplicate is received.

Note that this example doesn’t specify a time window argument, thus defaulting to
a 2 year window (1 year before and 1 year after). You should limit a time window
to the smallest reasonable window whenever possible for better performance.

rule: dup_nfs_not_resp:
    event: _event of_class 'NFS_NOT_RESPONDING',
    action: dup_and_drop_event:
        first_duplicate(_event, event: _dup_nfs_ev
                       where [status: 'outside'['CLOSED']]
        ),
        add_to_repeat_count(_dup_nfs_ev, 1),
        drop_received_event
    ).
See Also: all_duplicates
**first_effect_event**

Searches the event cache for the logically earliest effect event related to a cause event.

**Synopsis:** first_effect_event(_cause_event, _effect_event)

—OR—

first_effect_event(_cause_event, _effect_event, time_before, time_after)

**Description:** This predicate is used to search the event cache for the effect event related to a cause event in an event sequence defined with the create_event_sequence predicate.

If the time_before and time_after arguments are not specified, the event cache search time window defaults to 2 years (1 year before and 1 year after). You should limit a time window to the smallest reasonable window whenever possible for better performance.

**Arguments:**

_cause_event
A pointer to the cause event whose effect event is being searched for. Typically the event under analysis.

_effect_event
A pointer to the effect event found for the cause event. This argument must be free.

*time_after*
The number of seconds after the cause event has been received. This argument is used to limit the event cache search to a time window.

*time_before*
The number of seconds before the cause event has been received. This argument is used to limit the event cache search to a time window.

**Examples:** The following example searches the event cache for a related effect event that has been previously received. If one is found, the effect event is acknowledged and linked to its related cause event. This rule triggers on a superclass but it searches for a related effect event. This design lets you create a single rule to process any number of events that are related.

rule: 'link_cause_to_effect':

    event: _cause of_class 'EVENT',

    action: 'search_for_effect':
        first_effect_event(_cause, _effect, 3600, 0),
        set_event_status(_effect, 'ACK'),
        link_effect_to_cause(_effect, _cause)
    ).

**See Also:** create_event_sequence first_causal_event first_related_event
**first_instance**
Succeeds once for the first (most recent) event in the event cache that satisfies the specified class, attribute, and time window conditions.

**Synopsis:**  
\[
\text{first_instance(event: } _{\text{event of class}} \text{ class where } \text{attribute_conditions})
\]

—OR—

\[
\text{first_instance(event: } _{\text{event of class}} \text{ class where } \text{attribute_conditions, } _{\text{referenceEvent}} -\text{time_before} -\text{time_after})
\]

**Description:** Succeeds once for the first event that satisfies the specified class, attribute, and time window conditions.

**Arguments:**

\[
_\text{referenceEvent}
\]

A pointer to the reference event for the time window, typically the event under analysis.

\[
_\text{event: } _{\text{event of class}} \text{ class where } \text{attribute_conditions}
\]

Specifies an event filter. See “Event filters” on page 60 for additional information.

\[
-\text{time_after}
\]

The number of seconds after the reference event.

\[
-\text{time_before}
\]

The number of seconds before the reference event.

**Examples:** The following example shows a rule that:

1. Queries the event cache for the first instance of a universal_host event with the following additional conditions:
   - The status is not CLOSED.
   - The probe_arg attribute for the first instance of the event in the cache has the same value as the server attribute for the event under analysis, which is an NFS_No_Response event.
   - The event’s severity is CRITICAL.
   - The time window for searching is 20 minutes surrounding the event under analysis.

2. If an event meeting these conditions is found in the event cache, its severity is upgraded to FATAL.

```plaintext
rule: escalate: {
  description: 'escalate host down events when causing NFS problems',

  event: _{\text{event of class}} 'NFS_No_Response' where [ server: _{\text{server}}],

  action: 'increase_sev': {
    first_instance(event: _{\text{down_ev of class}} 'universal_host'
      where [status: outside ['CLOSED'], probe_arg: equals _{\text{server}}, severity: equals 'CRITICAL'],
      _event - 600 - 600 ),
```
set_event_severity(_down_ev, 'FATAL')
})
).

See Also: any_clear_target
**first_related_event**

Searches the event cache for the logically earliest event related to a reference event.

**Synopsis:**  

```lisp
first_related_event(_referenceEvent, _related_event, _relation)
```

—OR—

```lisp
first_related_event(_referenceEvent, _related_event, _relation, time_before, time_after)
```

**Description:** This predicate is used to search the event cache for the logically earliest cause or effect event related to the reference event. *Logically earliest* means as defined from left-to-right in an event sequence, with the logically earliest event starting from the left. If the found event is a cause event, the `_relation` argument is instantiated with the value of `c`. If the found event is an effect event, the `_relation` argument is instantiated with a value of `e`. For example, if events A, B, C, and D are defined with the `create_event_sequence` predicate as an event sequence in that order and the `first_related_event` predicate is called with an instance of event C as the reference event, the first instance of event A would be returned with the `_relation` argument instantiated with a value of `c` if it exists, otherwise event B would be returned with `_relation` set to `c` if it exists, otherwise event D would be returned with `_relation` set to `e` if it exists, otherwise the predicate fails.

This predicate should be used whenever correlation is needed to find cause events in the event cache, and then find effect events if a cause event is not found. Because this predicate only performs one search, it is more efficient than using the `first_causal_event` predicate followed by the `first_effect_event` predicate sequence of calls.

If the `time_before` and `time_after` arguments are not specified, the event cache search time window defaults to 2 years (1 year before and 1 year after). You should limit a time window to the smallest reasonable window whenever possible for better performance.

**Arguments:**

- `_referenceEvent`
  A pointer to the reference event whose logically earliest related event is being searched for.

- `_related_event`
  A pointer to the logically earliest related event found for the reference event. This argument must be free.

- `_relation`
  The relationship of the found event to the reference event. This argument must be free. Valid values are:
  - `c` A cause event to the reference event.
  - `e` An effect event to the reference event.

- `time_after`
  The number of seconds after the reference event has been received. This argument is used to limit the event cache search to a time window.

- `time_before`
  The number of seconds before the reference event has been received. This argument is used to limit the event cache search to a time window.
Examples: The following example searches the event cache for the logically earliest event related to the event under analysis. If one is found, the found event is acknowledged and linked to the reference event, either as a cause event or effect event, depending upon the returned value of the relation argument. This rule triggers on a superclass but it searches for a related event. This design lets you create a single rule to process any number of events that are related.

rule: 'link_effect_to_cause':

    event: _ev of_class 'EVENT',

    action: 'search_for_cause_or_effect':
    
        first_related_event(_ev, _related, _relation, 3600, 0),
        
        ( _relation == 'c',
        
            set_event_status(_ev, 'ACK'),
            link_effect_to_cause(_ev, _related)

        ;

        set_event_status(_related, 'ACK'),
        link_effect_to_cause(_related, _ev)

    )

).

See Also: create_event_sequence first_causal_event first_effect_event
**forward_event**

Forwards an event to an event server.

**Synopsis:**  `forward_event(_event)`

**Description:** This predicate forwards an event to an event server.

The predicate looks for the event server’s location in the tec_forward.conf file (located in the `rule_base_dir/TEC_RULES` directory). You must edit the tec_forward.conf file and change the value for the ServerLocation option to the host name of the system for the forwarded event.

**Note:** Because the forward_event predicate uses a non-Tivoli connection, you must specify the location of the server as an IP address or TCP/IP host name, regardless of whether there is a connection between the Tivoli regions.

The default setting in the tec_forward.conf file for the TestMode option is yes. This means that events are forwarded to a file. In order to actually forward an event to an event server, you must delete or comment out the TestMode option in the tec_forward.conf file.

**Arguments:**

`_event`  A pointer to the event to forward, typically the event under analysis.

**Examples:** The following example forwards events with a severity of CRITICAL or FATAL to the event server specified in the tec_forward.conf file:

```plaintext
rule: escalate: (  
    event: _evt of_class within [ 'EVENT' ]  
    where  
        [severity: within ['CRITICAL', 'FATAL'],  
    reception_action: action0: (  
        forward_event(_evt)  
    )  
).  
```

**See Also:** None.
**generate_event**
Generates an internal event.

**Synopsis:** `generate_event(event_class, list_of_event_attributes)`

**Description:** This predicate generates an event internally; that is, from within the event server instead of externally from a source such as an event adapter.

**Arguments:**

- `event_class`
  The event class for the generated event.

- `list_of_event_attributes`
  The attributes for the generated event. The attributes must be specified in a list using the following format:
  
  `[attribute1=value1, attribute2=value2,...]
  
**Examples:** The following example generates an event of class TradingDBDown with 4 attributes:

```vbnet
action:
  generate_event('TradingDBDown',
                 [source='SNMP',
                  origin=_origin,
                  hostname=_host,
                  msg='Trading DB host is down'])
)
```

**See Also:** None.
**get_attributes**
Retrieves event attribute values.

**Synopsis:** `get_attributes(_event, [ attribute_name=_attribute_value, ... ] )`

**Description:** This predicate retrieves the values of event attributes and instantiates variables with those values. The second argument is in list format.

**Arguments:**
- `_attribute_value`  
The variable to instantiate with the attribute value.
- `_event`  
The event from which to get the attribute values. Typically the event under analysis.
- `attribute_name`  
The name of the attribute whose value to retrieve.

**Examples:** The following example retrieves the hostname, severity, and status attribute values from the event under analysis and instantiates them in the `_hostname`, `_severity` and `_status` variables, respectively:

```plaintext
get_attributes(_event,[hostname=_hostname,
                        severity=_severity,
                        status=_status])
```

**See Also:** None.
**get_config_param**

Gets a rule engine configuration setting.

**Synopsis:**  `get_config_param(_name, _variable, default)`

**Description:** This predicate gets a rule engine configuration value defined in the $BINDIR/TME/TEC/.tec_config file and unifies it with a variable.

If the _name argument does not exist in the file, the default argument is unified with the _variable argument.

**Arguments:**

- **_name**  
The name of the configuration setting.

- **_variable**  
The variable to unify with the value of the configuration setting.

- **default**  
The value to unify with _variable if _name does not exist as a configuration setting in the file.

**Examples:**  The following example sets the _tec_rule_host variable to chair. If the tec_rule_host setting did not exist in the file, the variable would have been set to a value of not set. Some .tec_config file entries are shown first.

```plaintext
# .tec_config settings
#tec_rule_cache_size=10000
#tec_rule_cache_full_history=86400
#tec_rule_cache_non_closed_history=155520
#tec_rule_cache_clean_freq=3600
tec_rule_trace=YES
tec_rule_trace_file=/tmp/rules.trace
tec_rule_host=chair
tec_server_handle=5
get_config_param(_tec_rule_host, tec_rule_host, 'not set')
```

**See Also:**  None.
**get_global_grp**

Gets the value of all global variables in a group.

**Synopsis:**  
\[
\text{get_global_grp}(\text{group, key, value})
\]

**Description:** This predicate gets the value of all global variables in a group from the knowledge base. The predicate loops through the variables in the group and instantiates \_value for each variable found. \_value and \_key must be free.

**Arguments:**
- \_group  The group key for the variables.
- \_key  The key for the variables. Must be free.
- \_value  The value of the key. Must be free.

**Examples:** The following example gets all of the global variables for the Maintenance group.

\[
\text{get_global_grp('Maintenance', key, value)},
\]

**See Also:** [getGlobals][1], [getGlobalVar][2]
**get_global_var**

Gets a value of a global variable.

**Synopsis:**  
get_global_var(_group, _key, _value, _default)

**Description:**  
This predicate gets the value of one global variable from the knowledge base and unifies it with _value. If the variable has no value, it is set to _default and _default is unified with _value. _value must be free.

**Arguments:**

- **_default**
  The value for the variable if it currently has no value.

- **_group**
  The group key for the variable.

- **_key**
  The key for the variable.

- **_value**
  The value of the key. Must be free.

**Examples:**

The following example rule:

1. Gets the value for a global variable with a group key of Maintenance and a key value equal to the value of the origin attribute of the event under analysis. If there is no value for that key, it is initialized to a value of off.

2. A check is performed on the value of the global variable to see if the host is in maintenance mode.

3. If the check is true, the event under analysis is dropped and the rule set is exited.

```plaintext
rule:
check_maint_mode:
(
  event: _event of_class _event_class
  where [  
    origin: _origin 
  ],
  reception_action:  
  (  
    get_global_var('Maintenance', _origin, _maint_mode, 'off'),
    _maint_mode == 'on',
    drop_received_event,
    commit_rule
  )
).
```

**See Also:**  
get_globals, get_global_grp
**get_globals**
Gets all global variables.

**Synopsis:** `get_globals(_group, _key, _value)`

**Description:** This predicate returns the group key, key, and value for each global variable. The arguments must be free. The predicate loops through the global variables and instantiates the three arguments for each global variable found.

**Arguments:**
- `_group` The group key.
- `_key` The key.
- `_value` The value of the key.

**Examples:** The following example shows predicate usage:
```
global(_group, _key, _value)
```

**See Also:** `get_global_grp` `get_global_var`
**get_gm_time**
Gets the current time represented in Greenwich mean time (GMT).

**Synopsis:**  
`get_gm_time(_time_gm_struct)`

**Description:** This predicate gets the current time represented in GMT.  
 `_time_gm_struct` must be free.

**Arguments:**

- `_time_gm_struct`
  Represents a time structure in GMT. Do not confuse it with the data returned by the `get_time` predicate, in which the value for the `_time_epoch` argument is a number representing how many seconds have passed since an epoch.

**Examples:** The following example shows how to get the structure for the current time in GMT, convert the time to a string, and update the `time_string` attribute of the event with the string:

```prolog
get_gm_time(_time_gm_struct),
convert_ascii_time(_time_gm_struct, _time_string),
bo_set_slotval(_event, time_string, _time_string)
```

**See Also:**  
bo_set_slotval, convert_ascii_time, get_time, resolve_time
**get_local_time**
Gets the current local system time.

**Synopsis:**  
`get_local_time(_time_local_struct)`

**Description:**  
This predicate gets the current local system time. `_time_local_struct` must be free.

**Arguments:**

`_time_local_struct`
  
  Represents a time structure in local system time. Do not confuse it with the data returned by the `get_time` predicate, in which the value for the `_time_epoch` argument is a number representing how many seconds have passed since an epoch.

**Examples:** The following example shows how to get the structure for the current local system time, convert the time to a string, and update the `time_string` attribute of the event with the string:

```
get_local_time(_time_local_struct),
convert_ascii_time(_time_local_struct, _time_string),
bo_set_slotval(_event, time_string, _time_string)
```

**See Also:**  
[bo_set_slotval](#), [convert_ascii_time](#), [get_time](#), [resolve_time](#)
**get_time**
Gets the current time represented by an integer since the epoch, which is 00:00:00 Greenwich mean time (GMT) 01 Jan 1970 for most systems.

**Synopsis:** `get_time(_time_epoch)`

**Description:** This predicate gets the current time represented by an integer number of seconds since the epoch. `_time_epoch` must be free.

**Arguments:**

- `_time_epoch`
  Represents an epoch time number. Do not confuse it with the data returned by the `get_local_time` predicate, in which the value for the `_time_local_struct` argument is a time structure.

**Examples:** The following example shows how to get the epoch time number and then update the `time_epoch` attribute of the event with the number.

```prolog
get_time(_time_epoch),
bo_set_slotval(_event, time_epoch, _time_epoch)
```

**See Also:** `bo_set_slotval`
**global_exists**
Checks the existence of a global variable.

**Synopsis:**  `global_exists(group, key)`

**Description:**  This predicate checks that the global variable in group key `group` at key `key` exists. If the variable exists, the predicate succeeds.

**Arguments:**

- `group`  The group key for the variable to check.
- `key`  The key for the variable to check.

**Examples:**  The following example checks for the global variable whose key is the value of the origin attribute of the event under analysis and belongs to the Maintenance group:

```
global_exists('Maintenance', origin)
```

**See Also:**  None.
increment_slot
Adds a number to the value of the specified integer attribute.

Synopsis:  increment_slot(_event, _attribute_name, _by_value, _trigger)

Description:  This predicate adds a number to the value of the specified integer attribute.

Note:  Generally, the term slot has been replaced by the term attribute, even though this command name has not been changed.

Arguments:

_attribute_name
The attribute to change.

_by_value
The amount to add.

_event  A pointer to the event to change.

_trigger  Specifies whether change rules should be evaluated as a result of this attribute change. Valid values are: 'YES', yes, 'NO', or no.

Examples:  The following example shows predicate usage:
increment_slot(_event, host_down, 1, no)

See Also:  decrement_slot
**init_count**
Creates and initializes a counter.

**Synopsis:**  
`init_count(_key1, _key2, _value)`

**Description:**  
This predicate creates a counter identified by the values of `_key1` and `_key2`. It also initializes the counter to the value of `_value`. Typically, the initial value is set to 0.

This predicate is used in conjunction with the `check_and_increment_count` predicate, which is used to increment a count and compare it to a threshold value.

**Notes:**
1. If a counter is not created and initialized with the `init_count` predicate, it is created and initialized to 0 the first time the `check_and_increment_count` predicate is called to check the counter.
2. Once initialized, a counter continues counting until explicitly reinitialized with a new starting value.
3. You must reinitialize a counter that has reached its threshold if it is still needed for counting.

Counters are used to keep track of any arbitrary numeric value. The values of `_key1` and `_key2` can be set to easily identify the information being recorded. For example:

- To keep track of the number of times a particular event occurs on each host, the keys could be named using an `event_class,hostname` scheme; thereby creating a counter for each event and each host. For example, `perf_alert,orange`.
- To keep track of the number of times a particular failure occurs on a set of components, the keys could be named using a `failure,component` scheme; thereby creating a counter for each component and each failure. For example, `paper_jam,flr4rm23`.

If the event server stops, all counters are discarded.

**Arguments:**

- `_key1`  The primary key name for the counter. Must be instantiated.
- `_key2`  The secondary key name for the counter. Must be instantiated.
- `_value`  The value to initialize the counter. Must be instantiated.

**Examples:**
The following example counts the number of paper jams on a set of printers, based on receiving an event class of Printer_Jam. Printer counters are identified using a `failure,component` scheme. Printer_Jam events identify each printer in the hostname attribute.

Each counter is created, initialized to 0, and incremented to 1 the first time the `check_and_increment_count` predicate is called for a particular printer. Each subsequent call for that printer increments the count and then compares the count to the threshold value.

An administrator is notified when the number of paper jams on a printer reaches 5, and then the counter for that printer is reset to 0 using the `init_count` predicate. The administrator notification and reset of a counter is done in an ELSE clause of a Prolog statement because the `check_and_increment_count` predicate behavior is to fail when the count matches the threshold value.
rule: printer_jam: {
    event : _ev of_class 'Printer_Jam'
    where [hostname: _hn within ['flr4rm23',
        'flr3rm12',
        'flr1rm11',
        'flr6rm9'
    ],

    action: check_count: {
        (check_and_increment_count(printer_jam,_hn,5,_count)

        % ELSE clause follows

        exec_program(_ev,'scripts/notify.sh',
            'Printer failure on %s', [_hn], no),

        init_count(paper_jam,_hn,0)
    )
}).

See Also: check_and_increment_count
**init_event_activity**
Defines the reporting criteria for generating an event activity report.

**Synopsis:**  
`init_event_activity(file, event_exclusions, attribute_criteria, threshold)`

**Description:** This predicate defines the file for the report and defines the criteria for generating the report. An event activity report contains summary counts of the events in the event cache. You can configure the report to exclude particular events, filter on event attributes, and exclude counts that fall below a threshold value.

This predicate should be run in a rule triggered by a TEC_Start event at event server start-up time. This loads the predicate once, instead of every time it is needed.

**Arguments:**

*attribute_criteria*
The attributes whose summary counts to include in the report. The argument must be in list format; for example, `[source, hostname, severity]`.

A list element can be a single attribute or a nested list of multiple attributes; for example, `[hostname, severity]` can be one element. In the example of the `[hostname, severity]` nested list, a count of each severity is given for each host. The class keyword can be used in a nested list to count by event class name. For example, the `[class, hostname]` nested list provides a count of each host for each event class.

*event_exclusions*
The class names of the events whose information to exclude in the report. This argument must be in list format; for example, `['TEC_Heartbeat', 'TEC_Maintenance']`.

$file*
The path and file name where the report is written.

$threshold*
Any specification in the $attribute_criteria$ argument whose count is less than this value is not be shown in the report.

**Examples:**

1. The following example shows how to use the predicate:

   ```plaintext
   rep_freq is 20,
   init_event_activity(  
      '/tmp/event_activity',  
      % Report file  
      ['TEC_Heartbeat', 'TEC_Maintenance'],  
      % Do not report these events  
      [source,  
         hostname,  
         severity,  
         status,  
         [hostname,severity], % Multiple attribute reporting  
         [class,hostname] % Class reporting  
      ],  
      5 % Do not report counts less than this  
   ),
   ```
2. The initial timer for an event activity report must be started by a TEC_Tick event. The following example shows how to do this:

```plaintext
erule: configure_event_activity: 
    event: _event of_class 'TEC_Tick'
    where [msg: _msg equals 'Event Activity Report',
    duration: _reporting_frequency],

    reception_action: start_timer: 
        set_timer(_event,_reporting_frequency,_msg),
        commit_rule
    ).
```

3. The following example shows a fragment of an event activity report:

```
Event Activity For Server tkennedy

From: Thu Mar 02 14:14:02 2000.
To : Thu Mar 02 14:14:18 2000.

Reporting Frequency: 0 Minutes.

Total Events: 3332

Reporting Threshold: 5

======================================================================
Event Class Summary
======================================================================

Count  Class Name
-------------------------------------------------------------
849    TEC_Tick
848    TEC_DB
822    TEC_Notice
812    TEC_Error

======================================================================
Slot Summary
======================================================================

Count  Slot Criteria
-------------------------------------------------------------
3332    status=OPEN
590     severity=MINOR
574     severity=WARNING
564     severity=Critical
550     severity=UNKNOWN
544     severity=HARMLESS
510     severity=FATAL
12      hostname=midnight.austin.lab.tivoli.com
12      source=69.1.3.30
11      hostname=dhcp12-235.austin.lab.tivoli.com
11      source=69.1.12.235
11      hostname=stingray.austin.lab.tivoli.com
11      source=69.1.5.82
10      hostname=austin.lab.tivoli.com
10      source=69.1.1.6

See Also: print_event_activity
```
**ip_node_unreachable**
Determines if the event was sent from an unreachable subnet.

**Synopsis:**  ip_node_unreachable(_ipaddress, _event)

**Description:** This predicate tests to see if the given IP address is contained in the cache the event server maintains of unreachable IP addresses. If _ipaddress matches the subnet address and subnet mask of any of the subnets contained in the cache, the _event argument is set to the event handle for the corresponding TEC\_ITS\_SUBNET\_STATUS event.

This event can then be used to correlate the current event to the corresponding TEC\_ITS\_SUBNET event by using the link_effect_to_cause predicate. To check the success of the correlation, test that the _event argument is unified with an existing event, by using ground(_event). If it succeeds, there is a valid correlation.

This predicate requires that the NetView component be installed and that the netview.rls rule set is active.

**Arguments:**

_ipaddress_  
The unreachable IP address.

_event_ If the predicate is successful, this argument contains the event handle of the cached TEC\_ITS\_SUBNET\_STATUS event that matches the IP address. If no event is found, this argument is unchanged.
is_clearing_event
Tests whether an event has been defined as a clearing event with the 
create_clearing_event or create_event_sequence predicate.

Synopsis:  is_clearing_event(_event)

Description:  This predicate is used to test whether a rule or rule action should be 
run. If the event has been defined as a clearing event with the 
create_clearing_event or create_event_sequence predicate, and meets all of the 
appropriate conditions of the definition, the is_clearing_event predicate succeeds.

Arguments:

_event  A pointer to the event to test if it is a clearing event.

Examples:  The following example rule fragment tests whether the event under 
analysis is a clearing event. If the test passes, processing would continue with the 
next statement in the action. This rule triggers on the base event, so every 
incoming leaf event is tested.

rule: 'process_clearing_events':(

    event: _ev of_class 'EVENT',

    reception_action: 'check_for_clear':(
        is_clearing_event(_ev),

See Also:  create_clearing_event  create_event_sequence
**link_effect_to_cause**

Links an effect event to a cause event.

**Synopsis:**  
`link_effect_to_cause(_effect_event, _cause_event)`

**Description:** This predicate updates the cause_date_reception and cause_event_handle attributes of the effect event so that these attributes contain a reference to the cause event. The value of the date_reception attribute of the cause event is placed in cause_date_reception attribute and the value of event_handle attribute of the cause event is placed in the cause_event_handle attribute.

**Arguments:**

- `_cause_event_`  
  The cause event.

- `_effect_event_`  
  The effect event.

**Examples:** The following example links a universal_oserv event to a universal_host event if they are related, determined by their probe_arg attribute values. If they are related, the status attribute for the universal_oserv event is set to ACK.

```
rule: link_oserv_to_host: {
    event: _event of_class 'universal_oserv'
    where [probe_arg: _probe_arg, severity: equals 'WARNING'],

    action: 'link_host': {
        first_instance(event: _host_ev of_class
                       'universal_host'
                       where [severity: within ['CRITICAL', 'FATAL'],
                               probe_arg: equals _probe_arg,
                               status: outside ['CLOSED']
                       ]),
        set_event_status(_event,'ACK'),
        link_effect_to_cause(_event, _host_ev)
    }
},
```

**See Also:** [unlink_from_cause](#)
**load_globals**
Loads global variables from a file into the knowledge base.

**Synopsis:**  `load_globals(_file)`

**Description:**  This predicate loads all the global variables from a file into the knowledge base.

**Arguments:**

- `_file`  The path and file name that contains the variable definitions to load.

**Examples:**  The following example shows predicate usage:

```prolog
load_globals('/tmp/globalvars.txt')
```

**See Also:**  `save_globals`
**log_error**
Generates error messages to assist in rule development.

**Synopsis:** `log_error(format_string, variable_list, severity)`

**Description:** This predicate should be run from within a predicate you’ve created to help you debug rules. The more recent rule language predicates provided by IBM have this predicate embedded within them to help you debug problems with rules.

This predicate provides error messages the following ways:
- Written to a file
- TEC_Error events sent to the event serve.

Before calling the `log_error` predicate, do the following setup tasks:
- Define a rule that runs the `tell_err` built-in Prolog predicate. This predicate directs error messages to a specific file. You can call it from a rule that triggers on a TEC_Start event so it loads at event server start-up, or you can call it from within a rule that triggers on some other criteria. The `tell_err` predicate has the following format:
  ```prolog
tell_err('filename')
  ```
- Define the source location within the rule, so you have a point of reference from a generated error message. See [set_log_error_source on page 211](#) for additional information.

**Arguments:**

`_format`

The format specification for the output. The following format specifications are valid:

- `%c` Character.
- `%d` Integer printed in decimal notation.
- `%e` Real printed in exponential notation.
- `%f` Real printed in decimal notation.
- `%g` Real printed in its shortest form (decimal or exponential notation).
- `%o` Integer printed in octal notation, without sign and leading zero.
- `%s` String.
- `%u` Integer printed in unsigned decimal notation.
- `%x` Integer printed in hexadecimal notation, without sign and leading 0x.

You can supply more detailed conversion specifications between the `%` sign and the conversion character, as follows:

- `–` Left adjustment.
- `0` Zero padding to the left.
- `n` In cases of an integer or a string, `n` is the minimum length of the field.
- `n.m` In cases of a real, `n` is the minimum length of the field and `m` indicates the number of digits after the decimal point.
**severity**
Specifies the severity to assign the generated TEC_Error event.

**variable_list**
A list of variables whose values will appear in the message. There must be a matching format specification for each variable.

**Examples:** The following example shows a user-defined predicate named my_predicate that receives an argument (_data) that is actually passed in to be the argument of a user-defined predicate named check_data. The check_data predicate is the predicate to be debugged. The logic is as follows:

1. Define a source location named my_predicate for a point of reference from an error message. This is done with the set_log_error_source predicate.
2. Enable rule tracing of the check_data predicate with the trace_it predicate. To actually write the rule trace information to a file, the set_detailed_debugging predicate had to be run previously. The trace_it predicate just enables tracing.
3. If the check_data predicate fails, the log_error predicate is run and the Bad Data message along with the my_predicate source identifier are written to the error file. The tell_err predicate had to be run previously to define the location and name of the error file.

Additionally when the check_data predicate fails, a TEC_Error event is sent to the event server with a severity of CRITICAL, the Bad Data message, and the my_predicate source identifier. The message and the source identifier are assigned to the msg attribute of the event.

```prolog
my_predicate(_data):-
    set_log_error_source(my_predicate),
    ( trace_it(check_data),
      process_data(_data)
    ; log_error('Bad Data %s',[_data],'CRITICAL')
    )
```

**See Also:** set_log_error_source
place_change_request
Requests a change to an attribute value.

Synopsis: place_change_request(_event, _attributename, _newattributevalue)

Description: Change rules are triggered in response to the requested change. If there are no change rules in the rule base, the bo_set_slotval predicate would be a more efficient choice to change an attribute value, because processing resources are not used to check the rule base for change rules.

Arguments:

.attributename
The attribute to change.

.event  A pointer to the event containing the attribute to change.

.newattributevalue
The value to assign the updated attribute.

Examples: The following example requests to change the hostname attribute to a value of myhost:
place_change_request(_event, hostname, myhost)

See Also: bo_set_slotval, re_mark_as_modified
print_cache
Writes the event cache to a file.

Synopsis: print_cache(file_name)

—OR—

print_cache(file_name, event:_event of_class class where attribute_conditions)

Description: This predicate writes the event cache to a file. Two forms are provided:
• Without the event filter: writes the entire event cache
• With an event filter: writes certain events in the cache

Arguments:

event:_event of_class class where attribute_conditions
    Specifies an event filter for identifying particular events to write to the file.
    Do not use the same for the _event and class variables as those used in the
    event filter for the rule.

file_name
    The path and file name to which the event cache is written.

Examples:
1. The following example writes all of the events in the event cache to the /tmp/cache file.
   print_cache('/tmp/cache/')
2. The following example writes all events of class TEC_Start in the event cache to the /tmp/cache file.
   rule: print_cache: ( 
       event:_event of_class _class,
       reception_action: ( 
           print_cache('/tmp/cache', event:_cached_event of_class 'TEC_Start')
       )
   ).
3. The following example writes all events whose status attribute has a value of CLOSED in the event cache to the /tmp/cache file.
   rule: print_cache: ( 
       event:_event of_class _class,
       reception_action: ( 
           print_cache('/tmp/cache',
           event:_cached_event of_class _cached_class where
           [status: equals 'CLOSED'])
       )
   ).

See Also: print_class_tree print_event_activity
**print_class_tree**

Formats and writes an event class hierarchy tree from the active rule base to a file.

**Synopsis:**  
`print_class_tree(_file, _class)`

**Description:** This predicate formats and writes an event class hierarchy tree from the active rule base to a file, starting from a specified class as the root and continuing down to the leaf classes. It also prints the maximum depth and the total width of the tree, which is a representation of the general size of the tree.

**Arguments:**

- `_class` The name of the event class to start from in the class hierarchy.
- `_file` The path and file name to which the event class tree is written.

**Examples:** The following example formats and writes the entire event class tree of the active rule base (because the starting point is the base event class EVENT) to the `/tmp/class_tree` file.

`print_class_tree('/tmp/class_tree', 'EVENT')`

**See Also:** [print_cache](#), [print_event_activity](#)
**print_event_activity**
Writes the event activity report defined with the init_event_activity predicate.

**Synopsis:** print_event_activity

**Description:** This predicate writes the event activity report using the criteria set in the init_event_activity predicate. It is usually run from within a timer rule.

The frequency of when to write the report is controlled with the _duration argument of the set_timer predicate.

**Arguments:** None.

**Examples:**
1. The initial timer for an event activity report must be started by a TEC_Tick event so it can run indefinitely. The following example shows how to do this:
   
   ```
   rule: configure_event_activity: ( 
     event: _event of_class 'TEC_Tick'
       where [msg: _msg equals 'Event Activity Report', 
              duration: _rep_freq],
     reception_action: start_timer: ( 
       set_timer(_event,_rep_freq,_msg),
       commit_rule
     )
   ).
   ```

2. The following example shows a use of the print_activity_report predicate. An example report is shown in "init_event_activity" on page 174
   
   ```
   timer_rule: print_and_reset_event_activity: ( 
     event: _event of_class _class 
       where [],
     timer_info: equals 'Event Activity Report',
     timer_duration: _rep_freq,
     action: print_and_reset_event_activity: ( 
       print_event_activity,
       reset_event_activity,
       set_timer(_event,_rep_freq, 
                 'Event Activity Report')
     )
   ).
   ```

See Also: [init_event_activity](#)
**re_after_match**

Searches for a match in a string using a named regular expression and returns the substring located after the match as a result.

**Synopsis:** `re_after_match(_name, _string, _result)`

**Description:** This predicate searches for a match in `_string` using a named regular expression defined with the `re_create` predicate. The predicate succeeds if a match is found. The substring after the match is returned in `_result`.

Refer to Perl documentation for information about regular expression syntax and usage.

**Arguments:**

- `_name` The name of a regular expression defined with the `re_create` predicate.
- `_result` The substring located after the match.
- `_string` The string to search for a match.

**Examples:** The following example shows how to use the predicate:

```prolog
re_create(test,'a.*i')
% Create regular expression test.

re_after_match(test,'chair',_result)
% Search 'chair' using regular expression test.
% Return the substring after the match in _result.
% Succeeds, 'r' returned in _result.
```

**See Also:** `re_before_match`, `re_create`
re_before_match
Searches for a match in a string using a named regular expression and returns the substring located before the match as a result.

Synopsis:  re_before_match(_name, _string, _result)

Description: This predicate searches for a match in _string using a named regular expression defined with the re_create predicate. The predicate succeeds if a match is found. The substring before the match is returned in _result.

Refer to Perl documentation for information about regular expression syntax and usage.

Arguments:
_name The name of a regular expression defined with the re_create predicate.
_result The substring located before the match.
_string The string to search for a match.

Examples: The following example shows how to use the predicate:
re_create(test,'a.*r')
  % Create regular expression test.
re_before_match(test,'chair', _result)
  % Search 'chair' using regular expression test.
  % Return the substring before the match in _result.
  % Succeeds, 'ch' returned in _result.

See Also:  re_after_match  re_create
re_create
Defines a regular expression for use with other regular expression predicates.

Synopsis:  re_create(_name, _pattern)

Description:  This predicate defines a named regular expression that can be
referenced by other regular expression predicates. The predicate fails if the regular
expression is not valid. It should be run in a rule triggered by a TEC_Start event at
event server start-up time. This loads the predicate once, instead of every time it is
needed.

Refer to Perl documentation for information about regular expression syntax and
usage.

Arguments:
_name  The name that uniquely identifies the regular expression.
_pattern  The regular expression.

Examples:  The following example shows how to define a regular expression and
reference it from another regular expression predicate:
re_create(test,'h.*i')
% Create regular expression test.

re_search_string(test,'chair')
% Compare 'chair' to regular expression test.
% Succeeds, matches 'hai'.

See Also:  None.
re_mark_as_modified
Updates information for an event in the event database.

Synopsis: re_mark_as_modified(_event, _)

Description: This predicate is typically run after using the bo_set_slotval predicate to update event consoles and event database with the latest attribute values for an event.

Arguments:

_ Uninstantiated variable used internally by Prolog. Also referred to as the anonymous variable.

_event A pointer to the event to update. This should be the same event pointed to by the _event argument in bo_set_slotval.

Examples: The following example shows how to update the data for the event pointed to by _oldevent:
re_mark_as_modified(_oldevent, _)

See Also: bo_set_slotval, place_change_request
**re_match**

Searches for a match in a string using a named regular expression and returns a result.

**Synopsis:** `re_match(_name, _string, _index, _result)`

**Description:** This predicate searches for a match in `_string` using a named regular expression defined with the `re_create` predicate. The predicate succeeds if a match is found. The matched substring is returned in `_result`. The `_index` argument is used to specify which part of the matched substring to return.

Refer to Perl documentation for information about regular expression syntax and usage.

**Arguments:**

- `_index_` The part of the match to return in `_result`. A value of 0 returns the entire matching substring, a value of 1 indexes into the matched substring one position and returns the result, a value of 2 indexes into the substring two positions and returns the result, and so forth.

- `_name_` The name of a regular expression defined with the `re_create` predicate.

- `_result_` The matched substring, subject to the `_index` specification.

- `_string_` The string to search for a match.

**Examples:** The following example shows how to use the predicate:

```erlang
re_create(test,'a.*r')
% Create regular expression test.

re_match(test,'chair',0,_result)
% Search 'chair' using regular expression test.
% Return the entire result in _result.
% Succeeds, 'air' returned in _result.
```

**See Also:** `re_create`
**re_search_string**
Searches for a match in a string using a named regular expression.

**Synopsis:** `re_search_string(_name, _string)`

**Description:** This predicate searches for a match in `_string` using a named regular expression defined with the re_create predicate. The predicate succeeds if a match is found.

Refer to Perl documentation for information about regular expression syntax and usage.

**Arguments:**
- `_name` The name of a regular expression defined with the re_create predicate.
- `_string` The string to search for a match.

**Examples:** The following example shows how to use the predicate:

```prolog
re_create(test,'h.*i')
% Create regular expression test.
re_search_string(test,'chair')
% Search 'chair' using regular expression test.
% Succeeds, matches 'hai'.
```

**See Also:** re_create
**re_send_event_conf**
Sends an event to a remote event server.

**Synopsis:** `re_send_event_conf(_conf_file, _event)`

**Description:** This predicate sends an event to a remote event server defined in a configuration file. The configuration file must be located in the TEC_RULES subdirectory of the loaded rule base. The predicate references the configuration file by file name only, leaving off the .conf file name extension; for example, with a configuration file named host.conf, specify the value of host for the `_conf_file` argument. Each time a new configuration file is referenced by the predicate, its name is added to an internal configuration file table and its file handle is kept open. A maximum of 50 concurrent different configuration files are supported.

This predicate supports both connection and connectionless modes of operations. If events are to be forwarded to a remote event server frequently, ensure that the configuration file specifies connection_oriented for the ConnectionMode option. This communication prevents the event server from having to establish a communications channel each time an event is forwarded.

In addition, to ensure that the events intended for a remote event server are separated from events intended for another remote event server, the configuration file must specify the location and name of the cache file for events destined to a remote event server; for example, BufEvtPath=/etc/tivoli/orange.cache.

The following figure shows an example of a configuration file for use with the `re_send_event_conf` predicate. Configuration file options are described in the IBM Tivoli Enterprise Console Adapters Guide.

```
ServerLocation=orange.tivoli.com
TestMode=no
BufEvtPath=/etc/Tivoli/orange.cache
# ConnectionMode=connection_oriented
```

**Note:** If the configuration file is used in a rule base target, it must be distributed with the rule base target. This can be done by using the `–imptgtdata` option of the `wrb` command. See the IBM Tivoli Enterprise Console Command and Task Reference for complete details about the `wrb` command.

**Arguments:**

- `_conf_file`  
  The configuration file that defines remote event server information.

- `_event`  
  The event to be sent.

**Examples:** The following example sends the event under analysis to the remote event server specified in the configuration file host.conf:

```
re_send_event_conf('host', _event)
```

**See Also:** `forward_event`
**re_split_event_id**

Parses an element of the server_path event attribute.

**Synopsis:** `re_split_event_id(path_element, _host, _server_handle, _date_reception, _event_handle)`

**Description:** This predicate receives a list element from the server_path attribute as input, parses the element, and unifies (assigns) the parsed values with variables provided as arguments to the predicate. The server_path attribute is a list of elements that provides information about each event server that an event has passed through. Each element contains the information about one event server. The information is for each element is in the format of an event ID, which is described in the section "Event cache" on page 9. See the IBM Tivoli Enterprise Console Adapters Guide for additional information about the server_path attribute.

**Arguments:**

- `_date_reception`  
  The reception date of the event at the server is unified with this variable. This value was obtained from the date_reception attribute when the event was received at the server.

- `_event_handle`  
  The handle of the event at the server is unified with this variable. This value was obtained from the event_handle attribute when the event was received at the server.

- `_host`  
  The hostname of the server is unified with this variable. The value was obtained from the tec_rule_host configuration setting in the $BINDIR/TME/TEC/.tec_config file of the host where the server resides.

- `_path_element`  
  The list element of the server_path attribute to parse. Each element represents a server that the event has passed through.

- `_server_handle`  
  The handle of the server is unified with this variable. This value was obtained from the tec_server_handle configuration setting in the $BINDIR/TME/TEC/.tec_config file of the host where the server resides.

**Examples:** The following example iterates through each element in the server_path attribute and parses it:

```
bo_get_slotval(_event,server_path,_server_path),  
% Get the list for the server_path attribute.

member(_item,_server_path),  
% Get an element of the list.
% Because _item is free, the list will be traversed  
% and each element will be returned in succession.

  re_split_event_id(_item,_host,_server_handle,  
    _date_reception,_event_handle)  
% Parse each element into variables.
```

**See Also:** `bo_get_slotval`, `member`
**re_substitute**

Searches for a match in a string using a named regular expression, replaces the match, and returns the new string as a result.

**Synopsis:**  `re_substitute(_name, _string, _substitute, _result)`

**Description:** This predicate searches for a match in `_string` using a named regular expression defined with the `re_create` predicate. The predicate succeeds if a match is found. The value in `_substitute` replaces the match and the new string is returned in `_result`.

Refer to Perl documentation for information about regular expression syntax and usage.

**Arguments:**

- `_name`  The name of a regular expression defined with the `re_create` predicate.
- `_result`  The new string after substitution has been done.
- `_string`  The string to search for a match.
- `_substitute`  The value to replace the match with.

**Examples:** The following example shows how to use the predicate:

```prolog
re_create(test,'a.*w')
% Create regular expression test.
re_substitute(test,'hawk','oo',_result)
% Search 'hawk' using regular expression test.
% Return the new string in _result.
% Succeeds, 'hook' returned in _result.
```

**See Also:** `re_create`  `re_substitute_global`
**re_substitute_global**

Searches for all matches in a string using a named regular expression, replaces them, and returns the new string as a result.

**Synopsis:** `re_substitute_global(_name, _string, _substitute, _result)`

**Description:** This predicate searches for all matches in `_string` using a named regular expression defined with the `re_create` predicate. The predicate succeeds if at least one match is found. The value in `_substitute` replaces all occurrences of a match and the new string is returned in `_result`.

Refer to Perl documentation for information about regular expression syntax and usage.

**Arguments:**

- `_name` The name of a regular expression defined with the `re_create` predicate.
- `_result` The new string after substitution has been done.
- `_string` The string to search for a match.
- `_substitute` The value to replace matches with.

**Examples:** The following example shows how to use the predicate:

```prolog
re_create(test,'a.*w') % Create regular expression test.
re_substitute_global(test,'hawkhawkhawk','oo',_result) % Search 'hawk' using regular expression test.
% Return the new string in _result.
% Succeeds, 'hookhookhook' returned in _result.
```

**See Also:** `re_create` `re_substitute`
**redo_analysis**
Requests a reanalysis for an event.

**Synopsis:**  `redo_analysis(_event)`

**Description:** When correlating events, it might be necessary to re-evaluate the analysis of a previously received event. This predicate requests that the rule engine redo the analysis of regular rules for the specified event.

**Note:** It is possible to make the rule engine loop by using the redo_analysis predicate. For example if in the analysis of event A you ask to redo analysis of event B and vice-versa, the rule engine might enter an infinite loop.

**Arguments:**

_**event**  Specifies the event for which the analysis is to be redone.

**Examples:** The following example places a redo request on previously received INSTALLATION_FAILED events that might have been caused by the disk being full:

```prolog
rule: disk_full_check_install_failed: (
  description: 'look for installationfailed events for this host',
  event: _event of_class 'DISK_FULL'
    where [status: equals 'OPEN',
    hostname: _hostname ],

  action: ( all_instances(event: _install_ev
    of_class 'INSTALLATION_FAILED'
    where [target_host: equals _hostname],
    _event -600 -600 ),
    redo_analysis(_install_ev) )
).
```

**See Also:** None.
**remove_bslashes**
Converts back slashes to forward slashes in directory paths. In order to prevent backslashes from being interpreted as escape sequences, backslashes used as path separators must be specified as double backslashes.

**Synopsis:** `remove_bslashes(_path1, _path2)`

**Description:** This predicate converts the back slashes in the `_path1` argument to forward slashes and unifies the new path with the `_path2` argument.

**Arguments:**
- `_path1` The directory path to convert.
- `_path2` The converted directory path.

**Examples:** The following example converts back slashes in a directory path to forward slashes:

```prolog
% Assign value.
_path="\tivoli\data\repository',
% Convert back slashes.
% _new_path is unified with /tivoli/data/repository.
remove_bslashes(_path, _new_path)
```

**See Also:** None.
reset_event_activity
Resets the counts for all event reporting criteria to 0.

Synopsis: reset_event_activity

Description: This predicate resets the counts for all event reporting criteria to 0. It is usually run from within a timer rule following the print_event_activity predicate.

Arguments: None.

Examples: The following example shows a use of the predicate:

```prolog
timer_rule: reset_event_activity: {
  event: _event of_class _class
    where [ ],

  timer_info: equals 'Event Activity Report',
  timer_duration: _rep_freq,

  action: reset_activity: {
    print_event_activity,
    reset_event_activity,
    set_timer(_event,_rep_freq,'Event Activity Report')
  }
}
```

See Also: print_event_activity
**reset_global_grp**
Resets the value of all global variables in a group.

**Synopsis:**  \texttt{reset\_global\_grp(\_group,\_value)}

**Description:** This predicate changes the value of all global variables in a group in the knowledge base. The predicate loops through the variables in the group and for each variable found, changes its value to \_value.

**Arguments:**
- \_group  The group key for the variables.
- \_value  The new value for the variables.

**Examples:** The following example resets all of the global variables in the Maintenance group to off.
\texttt{reset\_global\_grp('Maintenance', 'off'),}

**See Also:** \texttt{set\_global\_var}
**resolve_time**
Retrieves the attributes of a time structure.

**Synopsis:** `resolve_time(_time_structure, _seconds, _minutes, _hours, _day_of_month, _month, _year, _day_of_week, _day_of_year, _daylight_saving)`

**Description:** This predicate retrieves attributes from a time structure represented by the `_time_structure` argument, and instantiates the remaining arguments to those attributes. `_time_structure` must be instantiated before calling resolve_time. The other arguments must be free. The values are in Greenwich mean time (GMT).

**Arguments:**

- `_day_of_month`
  Instantiated to an integer in the range 1–31.

- `_day_of_week`
  Instantiated to an integer in the range 0–6.

- `_day_of_year`
  Instantiated to an integer in the range 0–364.

- `_daylight_saving`
  Instantiated to an integer as reflected by the DST_ macros in `<sys/time.h>`.

  You can use this number to determine the type of daylight savings time style used on the current system. This could be useful if you need to manipulate time values returned by this predicate. The following example shows the values from a Solaris system:

  ```c
  #define DST_NONE 0 /* not on dst */
  #define DST_USA 1 /* USA style dst */
  #define DST_AUST 2 /* Australian style dst */
  #define DST_WET 3 /* Western European dst */
  #define DST_MET 4 /* Middle European dst */
  #define DST_EET 5 /* Eastern European dst */
  #define DST_CAN 6 /* Canada */
  #define DST_GB 7 /* Great Britain and Eire */
  #define DST_RUM 8 /* Rumania */
  #define DST_TUR 9 /* Turkey */
  #define DST_AUSTALT 10 /* Australian style with shift in 1986 */
  ```

- `_hours`
  Instantiated to an integer in the range 0–23.

- `_minutes`
  Instantiated to an integer in the range 0–59.

- `_month`
  Instantiated to an integer in the range 0–11.

- `_seconds`
  Instantiated to an integer in the range 0–59.

- `_time_structure`
  Represents a time structure. Do not confuse it with the data returned by the get_time predicate, in which the value for the `_time_epoch` argument is a number representing how many seconds have passed since an epoch.

- `_year`
  Instantiated to an integer in the range 00–99.

**Examples:** The following example shows how to get the structure for the current local system time, retrieve the attributes of the local system time structure, and update the month attribute of the event with the value of the `_month` argument.
get_local_time(_time_local_struct),
resolve_time(_time_local_struct, _seconds, _minutes,
    _hours,
    _day_of_month, _month, _year, _day_of_week,
    _day_of_year, _daylight_savings),
bo_set_slotval(_event, month, _month)

See Also: \texttt{bo_set_slotval}, \texttt{get_local_time}
**save_globals**
Writes all global variables from a group to a file.

**Synopsis:**  saveGlobals(_file, _group)

**Description:**  This predicate writes all the global variables from a group to a file.

**Arguments:**
_**file**  The path and file name to write the variables.
_**group**  The group key whose variables to write.

**Examples:**  The following example shows how to write the global variables in the Maintenance group to a file:
saveGlobals('/tmp/globalvars.txt', 'Maintenance')

**See Also:**  [load_globals](#)
**search_cache**
Performs a query of the event cache based on a named search defined with the create_cache_search_criteria predicate.

**Synopsis:** `search_cache(search_name, _referenceEvent, _maxEvents, _foundEvent)`

—OR—

`search_cache(search_name, _referenceEvent, _timeBefore, _timeAfter, _maxEvents, _foundEvent)`

**Description:** This predicate performs a query of the event cache. It is used in conjunction with the create_cache_search_criteria predicate, which defines a named search.

The second form of the predicate lets you specify a time window with the `_timeBefore` and `_timeAfter` arguments.

Succeeds once for each event that satisfies the search criteria.

You can use the get_attributes predicate to get the values for each found event’s attributes.

**Arguments:**

- `_foundEvent`  
  A pointer to a matching event.
- `_maxEvents`  
  The maximum number of events to return that meet the search criteria.
- `_referenceEvent`  
  A pointer to the reference event, typically the event under analysis.
- `search_name`  
  The name of the search criteria to use in the query. This name and its associated search criteria are defined with the create_cache_search_criteria predicate.
- `_timeAfter`  
  The number of seconds after the reference event.
- `_timeBefore`  
  The number of seconds before the reference event.

**Examples:** The following example uses the search named db_critical_search to find matching events within a 20 minute time window of the reference event. The search has been instructed to return no more than five matching events.

```
search_cache('db_critical_search',
            _refevent, 600,
            _timeBefore, 600,
            _timeAfter, 600,
            _maxEvents, 5,
            _found_event)
```

**See Also:** `create_cache_search_criteria`
**set_detailed_debugging**

Writes rule trace information to the rule trace file for predicates.

**Synopsis:**  set_detailed_debugging(on)

—OR—

set_detailed_debugging(off)

**Description:** This predicate toggles the writing of rule trace information for predicates that have rule tracing enabled with the trace_it predicate. You can run the set_detailed_debugging predicate from any rule.

**Arguments:**

- **off**  Specifies to not write rule trace information.
- **on**  Specifies to write rule trace information.

**Examples:** The following example shows a user-defined predicate named my_predicate that receives an argument (_data) that is actually passed in as the argument to a user-defined predicate named check_data. The check_data predicate is the predicate to be debugged. The logic is as follows:

1. Enable writing of the rule trace information to a file with the set_detailed_debugging predicate.
2. Define a source location named my_predicate for a point of reference from an error message. This is done with the set_log_error_source predicate.
3. Enable rule tracing of the check_data predicate with the trace_it predicate.
4. If the check_data predicate fails, the log_error predicate is run and the Bad Data message along with the my_predicate source identifier are written to the error file. The tell_err predicate had to be run previously to define the location and name of the error file.

   Additionally when the check_data predicate fails, a TEC_Error event is sent to the event server with a severity of CRITICAL, the Bad Data message, and the my_predicate source identifier. The message and the source identifier are assigned to the msg attribute of the event.

Regardless of whether the predicate succeeds or fails, trace information is written to the rule trace file.

```prolog
set_detailed_debugging(on),
my_predicate(_data):-
  set_log_error_source(my_predicate),
  (  
    trace_it(check_data),
    process_data(_data)
  ;
    log_error('Bad Data %%s',[_data],'CRITICAL')
  )
```

**See Also:** None.
set_event_administrator
Sets the administrator for an event.

Synopsis:  set_event_administrator(_event, new_administrator)

Description: This predicate sets the value for the administrator attribute for the specified event.

Note: This predicate directly modifies the value of the attribute without issuing an internal change request that goes through the change rules. To trigger change rules, call the place_change_request predicate following the set_event_administrator call.

Arguments:

_event A pointer to the event for which the administrator is to be set.

new_administrator Specifies the new administrator for the event.

Examples: The following example shows predicate usage:

set_event_administrator(_event, bjones)

See Also: change_event_administrator, place_change_request
**set_event_message**
Sets the msg attribute of an event.

**Synopsis:** `set_event_message(_event, _format, [_value])`

**Description:** The format specification is similar to that for the `sprintf()` function in the C programming language.

**Note:** This predicate directly modifies the value of the attribute without issuing an internal change request that goes through the change rules. To trigger change rules, call the `place_change_request` predicate following the `set_event_message` call.

**Arguments:**
- `_event` A pointer to the event containing the msg attribute to assign the value.
- `_format` The format specification for the msg attribute value. The following conversion specifications are valid:
  - `%c` Character.
  - `%d` Integer printed in decimal notation.
  - `%e` Real printed in exponential notation.
  - `%f` Real printed in decimal notation.
  - `%g` Real printed in its shortest form (decimal or exponential notation).
  - `%o` Integer printed in octal notation, without sign and leading zero.
  - `%s` String.
  - `%u` Integer printed in unsigned decimal notation.
  - `%x` Integer printed in hexadecimal notation, without sign and leading 0x.

You can specify more detailed conversion specifications between the `%` sign and the conversion character, as follows:
- Left adjustment.
- 0 Zero padding to the left.
- `n` In cases of an integer or a string, `n` is the minimum length of the field.
- `n.m` In cases of a real, `n` is the minimum length of the field and `m` indicates the number of digits after the decimal point.

- `_value` The text to be formatted for the msg attribute. This argument is in list format.

**Examples:** The following example shows various uses of the predicate:
- `integer is 123`,
- `real is 12.3`,
- `_string = 'Hello, World'`,
- `% Assign values.

`set_event_message(_event, '%s', [_string]),`
- % msg attribute assigned 'Hello, World'.
set_event_message(_event, '%%20s', [string]),
  % msg attribute assigned 'Hello, World'.

set_event_message(_event, '%%-20s', [string]),
  % msg attribute assigned 'Hello, World'.

set_event_message(_event, 'Integer in decimal notation: %%d', [integer]),
  % msg attribute assigned 'Integer in decimal notation: 123'.

set_event_message(_event, 'Integer in decimal notation with field width: %%10d', [integer]),
  % msg attribute assigned 'Integer in decimal notation with field width: 123'.

set_event_message(_event, 'Integer in decimal notation with leading zeros: %%010d', [integer]),
  % msg attribute assigned 'Integer in decimal notation with leading zeros: 0000000123'.

set_event_message(_event, 'Integer in octal notation: %%o', [integer]),
  % msg attribute assigned 'Integer in octal notation: 173'.

set_event_message(_event, 'Integer in hexadecimal notation: %%x', [integer]),
  % msg attribute assigned 'Integer in hexadecimal notation: 7b'.

set_event_message(_event, 'Real in decimal notation: %%f', [real]),
  % msg attribute assigned 'Real in decimal notation: 12.300000'.

set_event_message(_event, 'Real in decimal notation with field width: %%3.2f', [real]),
  % msg attribute assigned 'Real in decimal notation with field width: 12.30'.

set_event_message(_event, 'Real in real notation: %%f', [real]),
  % msg attribute assigned 'Real in real notation: 12.300000'.

set_event_message(_event, 'Real in exponential notation: %%e', [real]),
  % msg attribute assigned 'Real in exponential notation: 1.230000e+01'.

set_event_message(_event, 'Real in its shortest form: %%g', [real]),
  % msg attribute assigned 'Real in its shortest form: 12.3'.

See Also: [place_change_request]
**set_event_severity**
Sets the severity of an event.

**Synopsis:**  *set_event_severity(_event, new_severity)*

**Description:** This predicate sets the severity of the specified event.

**Note:** This predicate directly modifies the value of the attribute without issuing an internal change request that goes through the change rules. To trigger change rules, call the place_change_request predicate following the *set_event_severity* call.

**Arguments:**
- `_event`  A pointer to the event for which the severity is to be set.
- `new_severity`  The new event severity.

**Examples:** The following example shows predicate usage:
*set_event_severity(_event, 'CRITICAL')*

**See Also:** [change_event_severity](#), [place_change_request](#)
**set_event_status**
Sets the status of an event.

**Synopsis:** `set_event_status(_event, new_status)`

**Description:** This predicate sets the status attribute of the specified event.

**Notes:**
1. This predicate directly modifies the value of the applicable event attribute without issuing an internal change request that goes through the change rules. To trigger change rules, call the place_change_request predicate following the `set_event_status` call.
2. If the status attribute was set to a value of CLOSED by this predicate, the duration attribute is not modified to provide the age. The value remains at 0. See “duration” on page 8 for additional information.

**Arguments:**

- `_event_` A pointer to the event for which the status is to be set.
- `new_status` The new value to assign the status attribute. Valid values for the status attribute are described on page 8.

**Notes:**
1. You cannot change a status of CLOSED.
2. A change from ACK to OPEN status is not valid for the `new_status` argument.

**Examples:** The following example shows how to set the status attribute of the event under analysis to ACK:

```
set_event_status(_event, 'ACK')
```

**See Also:** `change_event_status`, `place_change_request`
**set_global_var**
Sets the value of a global variable.

**Synopsis:** `set_global_var(_group, _key, _value)`

**Description:** This predicate sets the value of one global variable in the knowledge base. To set a variable to a list, use the `[ ]` notation.

**Arguments:**

- `_group_` The group key for the variable.
- `_key_` The key for the variable.
- `_value_` The value to set.

**Examples:** The following example shows various uses of the predicate:

```prolog
set_global_var('My group key', _key, 'My value')
set_global_var('My group key', _key, ['a', 'b', 'c'])
set_global_var('Maintenance', _origin, 'on')
```

**See Also:** `reset_global_grp`
**set_log_error_source**
Defines a source identifier for a point of reference from an error message generated by the log_error predicate.

**Synopsis:** `set_log_error_source(source_location)`

**Description:** This predicate defines a source location within a rule action so you have a point of reference from a generated error message to help you debug the rule. For the argument, specify the name of a rule action or other significant identifier (for example, a predicate name).

**Arguments:**

*source_location*
A string identifying a meaningful location in a rule.

**Examples:** The following example shows a user-defined predicate named `my_predicate` that receives an argument (_data) that is actually passed in to be the argument of a user-defined predicate named `check_data`. The `check_data` predicate is the predicate to be debugged. The logic is as follows:

1. Define a source location named `my_predicate` for a point of reference from an error message. This is done with the `set_log_error_source` predicate.
2. Enable rule tracing of the `check_data` predicate with the `trace_it` predicate. To actually write the rule trace information to a file, the `set_detailed_debugging` predicate had to be run previously. The `trace_it` predicate just enables tracing.
3. If the `check_data` predicate fails, the `log_error` predicate is run and the Bad Data message along with the `my_predicate` source identifier are written to the error file. The `tell_err` predicate had to be run previously to define the location and name of the error file.

Additionally when the `check_data` predicate fails, a TEC_Error event is sent to the event server with a severity of CRITICAL, the Bad Data message, and the `my_predicate` source identifier. The message and the source identifier are assigned to the msg attribute of the event.

```prolog
my_predicate(_data):-
    set_log_error_source(my_predicate),
    ( trace_it(check_data),
      process_data(_data)
    ;
      log_error('Bad Data %s',\[\_data\],\'CRITICAL\')
    )
)```

**See Also:** `log_error`
**set_timer**

Sets a timer on an event.

**Synopsis:**  
```
set_timer(_event, timer_duration, timer_info)
```

**Description:**  
This predicate sets a timer on a received event. When the timer expires, the rule engine finishes processing the current event and then triggers timer rules for the specified event. In essence, the expiration of a timer is a transaction requesting the execution of timer rules on a specified event.

The maximum number of active timers that can be placed on events with this predicate is 1000.

An event of class TEC_Tick always exists in the event cache; that is, it is never aged out of the event cache. You can search for this event in the cache and use it to start a timer, knowing that it will always be there.

**Arguments:**

- `_event` A pointer to the event on which the timer is being set.
- `timer_duration` The duration (in seconds) of the timer. It can also be used for event filtering in a timer rule.
- `timer_info` The timer information. This argument can be anything, such as an integer, a string, or a structured item. It can be used for event filtering in a timer rule.

**Examples:**  
The first rule in the following example initially sets the timer for event activity report generation. The second rule is evaluated whenever the timer expires for event activity reporting. The action in the second rule prints the event activity report, resets counters for the next event activity report, and sets a new timer for generation of the next event activity report.

```
rule: {
    event: _event of_class 'TEC_Start'
    where [ ],
    reception_action: {
        first_instance(event:_ev of_class 'TEC_Tick' where []),
        set_timer(_event, 600, 'Event Activity Report')
    },
    timer_rule: reset_print_activity: {
        event: _event of_class _class
        where [ ],
        timer_info: equals 'Event Activity Report',
        timer_duration: _rep_freq,
        action: reset_print_activity: {
            print_event_activity,
            reset_event_activity,
            set_timer(_event, _rep_freq, 'Event Activity Report'
        }
    },
}
```

**See Also:** None.
**trace_it**
Enables tracing of user-defined predicates.

**Synopsis:**  \texttt{trace_it(predicate\_name)}

**Description:** This predicate designates which user-defined predicates to trace.

**Note:** Do not use this predicate to enable tracing for anything but predicates you have created.

**Arguments:**

\textit{predicate\_name}

The name of the predicate to trace.

**Examples:** The following example shows a user-defined predicate named \texttt{my\_predicate} that receives an argument \texttt{(_data)} that is actually passed in to be the argument of a user-defined predicate named \texttt{check\_data}. The \texttt{check\_data} predicate is the predicate to be debugged. The logic is as follows:

1. Define a source location named \texttt{my\_predicate} for a point of reference from an error message. This is done with the \texttt{set\_log\_error\_source} predicate.
2. Enable tracing of the \texttt{check\_data} predicate with the \texttt{trace\_it} predicate. To actually write the rule trace information to a file, the \texttt{set\_detailed\_debugging} predicate had to be run previously. The \texttt{trace\_it} predicate just enables tracing.
3. If the \texttt{check\_data} predicate fails, the \texttt{log\_error} predicate is run and the Bad Data message along with the \texttt{my\_predicate} source identifier are written to the error file. The \texttt{tell\_err} predicate had to be run previously to define the location and name of the error file.

   Additionally when the \texttt{check\_data} predicate fails, a \texttt{TEC\_Error} event is sent to the event server with a severity of CRITICAL, the Bad Data message, and the \texttt{my\_predicate} source identifier. The message and the source identifier are assigned to the \texttt{msg} attribute of the event.

\begin{verbatim}
my\_predicate(_data):-
    set\_log\_error\_source(my\_predicate),
    (trace\_it(check\_data),
     process\_data(_data)
    ; log\_error('Bad Data %s',[_data],'CRITICAL')
    )
)

See Also: \texttt{set\_detailed\_debugging}
\end{verbatim}
unlink_from_cause
Unlinks an effect event from a cause event.

Synopsis:  unlink_from_cause(_effect_event)

Description:  This predicate updates the cause_date_reception and
cause_event_handle attributes of the effect event to a value of 0, breaking the link
between the two events.

Arguments:

_effect_event
  The event to be unlinked.

Examples:  The following example shows predicate usage:
unlink_from_cause(_oserv_down_event)

See Also:  link_effect_to_cause
**update_event_activity**
Captures event information for reporting by the print_event_activity_report predicate.

**Synopsis:**  update_event_activity(_event)

**Description:**  This predicate captures information from an event and stores it internally for later reporting. It is typically called in a rule that runs on every event class.

**Arguments:**

_event  A pointer to the reference event, which is typically the event under analysis.

**Examples:**  The following example shows a use of the predicate:

```prolog
rule: update_event_activity: {

  event: _event of_class _class
  where [],

  reception_action: update_activity: (  
    % recorded(event_activity, active),
    % Line above used with im.rls (intermediate mgr. rules)

    update_event_activity(_event)
  )
}.
```

**See Also:**  [print_event_activity](#)
Chapter 5. Correlation examples

This chapter describes processing guidelines and provides some scenarios of how to correlate events with IBM Tivoli Enterprise Console rules. It assumes analysis of events has already been done. Analysis of the events in your environment is a key step in determining how to develop a rule base. See page 3 in Chapter 1, “Rule development fundamentals” for additional information.

The rules, BAROC files, and event relationship diagrams for the examples shown in this chapter are available on the event server host in the $BINDIR/TME/TEC/samples/correlation directory.

Event sequences shown in flowcharts in this chapter are represented as follows:

![Flowchart symbols]

- Root cause event
- Clearing event
- Effect event

Recording causal relationships between events

Once enterprise-significant events have been identified, the best way to record causal relationships between them is to review all the events from a single component, divide them into logically-related groups, and create flowcharts showing the sequence from one event to the next. These flowcharts should also include any clearing events that signify a return to normal state for the component. For example, the following figure shows the relationships between events from an APC uninterruptible power supply that might be generated due to a loss of electrical power from the utility company.

![Relationship flowchart]
The flowchart shows that the upsOnBattery event is generated by the uninterruptible power supply as soon as it detects a loss of line power and engages the battery backup. This is the first notification of a problem because there is no event leading into the upsOnBattery event.

If the power is not restored, a lowBattery event is generated, indicating the uninterruptible power supply battery level is getting low. A low battery condition can eventually deteriorate and lead to the generation of a upsDischarged event. A upsDischarged event signals that the battery has been run down and the uninterruptible power supply attempts a smooth shutdown of the machine.

Note: Even though a monitoring system might not necessarily receive all of the effect events in a sequence, the logical relationships are still valid. It is entirely possible to receive a cause event and then an effect event from far downstream in the sequence, without receiving any of the intervening events. The events should still be defined as a single logical sequence even if some of the events are not always received.

For example, if event A leads to B, B leads to C, and C leads to D, these should be defined as one logical sequence. If you only receive events A and D, the single sequence definition is sufficient. You do not need to define multiple sequences like A to B, A to C, A to D, B to C, B to D, and so forth. The sequence definition handles the logical relationships among the events in the sequence.

The powerRestored, returnFromLowBattery, and dischargeCleared events are clearing events. They indicate that a condition no longer exists; that is, the machine has returned to its typical state. In the flowchart, the clearing events point to their effect events; for example, a powerRestored event clears a upsOnBattery event. Not all events have corresponding clearing events in a system.

The IBM Tivoli Enterprise Console product provides predicates to define event relationships such as those shown in the flowchart and correlate those events as they are received by the event server. These predicates and their usage are described in subsequent sections of this chapter.

Special cases when recording causal relationships
The following special cases can arise when recording causal relationships between events. Predicates are provided to accommodate these special cases.

- A component sends multiple events of the same class but with different meaning determined by the value of a specific attribute.
- An event is not part of a sequence of events, yet it has a clearing event.

Multiple events of the same class but different meanings
The following flowchart shows an example of this special case. Compaq Insight Manager physical drive status events are all of class cpqTape3PhyDrvStatusChange. The value of the cpqTapePhyDrvCondition attribute determines the status, using the following values: Degraded, Failed, and OK. The clearing event contains the OK value. Other products such as Distributed Monitoring and those that generate SNMP traps similarly use attribute values with
the same class to indicate status.

An event is not part of a sequence
The following flowcharts show two examples of this special case. With this special case there is no sequence of problem events as there is in the flowchart on page 218. A clearing event can clear one or more problem events, but the problem events are not part of a sequence. For example, event A can clear events B, C, and D, which do not form an event sequence.

Cisco router events (shown in the figure on the left) are generated with different classes for problem and clearing events. Compaq Insight Manager drive array events (shown in the figure on the right side) are generated with the same class but use different values in the cpqDa3PhyDrvStatus attribute for problem and clearing events.

Using causal relationship information
An event’s position in a sequence is used to describe it and can determine the type of processing it should receive.

Cause events
A cause event precedes another event in a sequence. It is most likely the cause of an event that arrives later, assuming they are related to the same component, such as a host, router, printer, and so on.
In addition to routine processing for events (for example, duplicate detection), processing for a root cause event can include automatically creating a trouble ticket and searching for any effect events that might have been received before the root cause event, perhaps because of polling intervals or network delays. If effect events are found, they should be linked to their cause event, acknowledged, and have their severity reduced to indicate that they are the effects of a known problem. If a trouble ticket is created by an effect event that was received, the trouble ticket should be automatically updated with the new cause information.

In the flowchart on page 218, the upsBatteryOn event is a cause event to the lowBattery and upsDischarged events. Both the upsOnBattery and lowBattery events are cause events to the upsDischarged event.

The upsBatteryOn event is the first event, because it is the first one in the chain of events. Because it is the first event, it is commonly referred to as the root cause event. All subsequent events result from this event.

Effect events

An effect event is a symptom of some other problem. In the flowchart on page 218, the upsDischarged event is an effect event to the lowBattery and upsOnBattery events.

The root cause of a problem might not always be known when an effect event is received; for example, a monitoring system might detect an effect to the component before the root cause and send the effect event first. In this situation, the effect event must be considered as the root cause until the true root cause is known.

In addition to routine processing for events (for example, duplicate detection), processing for an effect event should include a search for the root cause event that has been received in the sequence. If a cause event is found, the effect event should be linked to it, acknowledged, and have its severity reduced to indicate that it is an effect of a known problem. If the cause event created a trouble ticket, the trouble ticket should be automatically updated with effect information.

Continuing with the uninterruptible power supply example on page 218, processing for the upsDischarged event should search the event cache for a upsOnBattery event from the same component. If a upsOnBattery event is not found, a search should be made for a lowBattery event from the same component. If neither of these events are found, the upsDischarged event must be considered as the root cause for now (the battery might be faulty or depleted and need replacing).

Clearing events

A clearing event indicates the return to a typical state, thus canceling a problem state. In the flowchart on page 218, the powerRestored event is a clearing event for the upsOnBattery event, and the returnFromLowBattery event is a clearing event for the lowBattery event.

When a clearing event is received, processing should close any related problem events, close the clearing event itself, and close any related trouble ticket. It is a good idea to also update the administrator attribute of the clearing event to indicate it was automatically closed by a rule.
Although extremely uncommon, sometimes a clearing event is received before corresponding problem events. This situation can occur if the monitoring system detects a solution to a problem or if network delays exist. In uncommon situations such as this, the correlation predicates can search the event cache for a corresponding clearing event when a problem event is received (called a reverse lookup), and thus the problem event can be closed without further processing. Under almost all circumstances though, a clearing event will not arrive before a corresponding problem event. Because reverse-lookup processing consumes a large amount of resources, do not use it unless absolutely necessary.

Processing events based on causal relationships

Putting all of the cause information together results in basic guidelines for processing events based on their role or position in an event sequence. Cause events are typically managed identically (regardless of event class) for a given component, and often across many different parts of an enterprise. The event processing guidelines for causal relationships are categorized as follows:

- Processing cause events
- Processing effect events
- Processing clearing events

Processing cause events

The following list describes typical processing for cause events:

1. **Duplicate event detection and escalation**: If a duplicate event exists in the event cache within a time window, increment its repeat count, drop the current event, and exit processing. If the number of duplicates within the time window exceeds a threshold, increase the cached event’s severity, notify an administrator, or do both actions, and exit processing.

2. **Search for a prior clearing event**: In the extremely uncommon occurrence of a clearing event being received before its problem events, search the event cache for the clearing event as soon as a related problem event is received. If a clearing event is found in the cache, close the problem event immediately and exit processing.

   **Note**: Under almost all circumstances, a clearing event will not arrive before a corresponding problem event. Because reverse-lookup processing consumes a large amount of resources, do not use it unless absolutely necessary.

3. **Search for effect events**: If duplicate or clearing events are not found in the event cache, search for related effect events that could have been received before the cause event. If effect events are found, link them to the cause event, update the trouble ticket (if one exists), and exit processing.

4. **Perform notification and open a trouble ticket**: If this step is reached, the current event is the first notification of a problem. This should trigger a notification to an administrator, create a trouble ticket, and run a script if necessary.

Processing effect events

The following list describes the order of processing for effect events:

1. **Duplicate event detection and escalation**: Same guideline as step 1 in “Processing cause events”.

2. **Search for a prior clearing event**: Same guidelines as step 3 in “Processing cause events”.

Chapter 5. Correlation examples  221
3. **Search for prior cause events**: In most circumstances, a cause event is received before related effect events; thereby the cause is known when a related effect event is received. Search the event cache for the root cause event, and if it is found, link the effect event to it, update the trouble ticket (if one exists), and exit processing.

4. **Search for effect events**: If a previously received cause event is not found in the event cache, the current event is assumed to be the root cause event (pending further information). Search for related effect events that could have been received before the cause event. If related effect events are found, link them to the cause event, update the trouble ticket (if one exists), and exit processing.

5. **Perform notification and open a trouble ticket**: Same guideline as step 4 in “Processing cause events” on page 221.

**Processing clearing events**

The following describes the processing for clearing events:

**Search for related problem events**: Search the event cache for related problem events. Close each related problem event. The clearing event can also trigger administrator notification or trouble ticket updating if required. The clearing event can be closed after the related problem events are closed, or it can remain open pending the arrival of additional problem events.

---

**Event correlation rule language predicates**

Beginning with IBM Tivoli Enterprise Console Version 3.7, a number of rule language predicates have been provided with the following goals in mind:

- To make it easier and more efficient to perform event correlation based on causal relationships, as described in previous sections of this chapter. For example, there are predicates such as the following:
  - To create descriptions of event relationships like those relationships described previously. These predicates are loaded into the knowledge base of the rule engine when the event server is started.
  - That use the descriptions of event relationships stored in the predicates previously mentioned to search the event cache for related events when new events are received.

- To reduce the number of rules that must be developed so one rule can manage many events. This can be accomplished because:
  - Processing for an event is often determined by its role in an event sequence rather than its specific event class.
  - Event relationship information is stored in the knowledge base of the rule engine and available when the event server is started.

With these two goals in mind, the correlation predicates are designed to record and retrieve information by event class. This means that one rule can process any number of different event classes provided those classes require the same type of processing. The predicates determine the event class of the event under analysis and retrieve the appropriate information automatically. The number of rules to develop is based on the number of unique event processing behaviors needed (one rule for each behavior), rather than by the number of events in the environment.

The following steps are required to use the correlation predicates:
1. Create the event relationship information to load into the knowledge base of the rule engine. These predicates are loaded at event server start-up when a TEC_Start event triggers the rule that calls the predicates that define the relationships.

2. Create the rules that use the event relationship information previously loaded to correlate incoming events for appropriate processing.

**Defining event relationship information**

Keep the following in mind when defining event relationship information:

- Event cache searches for correlated events are based on event sequences and clearing events defined with the create_event_sequence and create_clearing_event predicates, respectively.
- Some monitoring sources use a single event class with a changing attribute value that determines the type of event; for example, Distributed Monitoring and Compaq Insight Manager use this method, as previously described in this chapter.
- The correlation predicates store event relationship information the following ways:
  - Clearing events are indexed by their class names so that when a clearing event is received, its information can be found quickly.
  - Problem event information can include information about the events that clear them, for purposes of reverse lookup, which is an extremely uncommon situation described on page 221.
- These predicates are loaded at event server start-up when a TEC_Start event triggers the rule that calls the predicates which define the relationships.

See "create_event_sequence" on page 132 and "create_clearing_event" on page 126 for detailed information and examples for using these predicates to define event relationship information.

**How event criteria is evaluated**

The correlation predicates permit specification of attribute conditions for correlation purposes. These conditions can be absolute requirements on attribute values, or attribute-match conditions that require the values of certain attributes to match between two events. It is important to understand how these attribute conditions are evaluated by the correlation search predicates.

When an event is submitted with a correlation search predicate (for example, the first_causal_event predicate), its absolute conditions are evaluated first to ensure that the event qualifies as a correlation event. If the absolute conditions are not met, the predicate fails. If the absolute conditions are met, the event cache is searched for related events based on event class. Once an event of the appropriate class is found, the found event’s absolute conditions are checked to ensure that it qualifies as a correlation event, and its attribute-match conditions are checked against the original event to ensure the two events can be correctly correlated.

With clearing events the situation is a little more complex because there are more possibilities. If a clearing event is defined within an event sequence (using the create_event_sequence predicate), the attribute conditions for the target event are typically defined with the sequence and those conditions are used.

If a clearing event is defined with the create_clearing_event predicate (for defining clearing events not defined with the create_event_sequence predicate), the conditions for both the clearing event and target events can be specified. Under
typical circumstances, a clearing event is received and the event cache is searched for its targets using the any_clear_target or all_clear_targets predicate. As explained previously, only the absolute conditions for the clearing event are checked to ensure that it qualifies for correlation. When a target event is found, the target event’s absolute and its attribute-match conditions are checked to ensure that the two events can be correlated. For this reason, under typical circumstances the attribute conditions for the clearing event contains only absolute conditions, and the attribute conditions for the target event contains at least one attribute-match condition.

The only time attribute conditions for a clearing event should include attribute-match conditions is if a reverse lookup is used with the any_clearing_event predicate, to search the event cache for a previously-received clearing event when one of its targets is received. Under this extremely uncommon circumstance, the absolute criteria of the problem event is evaluated immediately and the clearing event’s absolute and attribute-match conditions are evaluated when the clearing event is found. When performing a reverse lookup, the attribute-match criteria for the clearing and target events should be the same. The absolute criteria can be different.

**Using event relationship information to find related events**

Keep the following items in mind when using event relationship information to search for related events in the event cache:

- Event cache searches for correlated events are based on event sequences and clearing events defined with the create_event_sequence and create_clearing_event predicates.
- Most of the correlation search predicates offer two versions: one that lets you specify a time window for limiting searches, and the other that does not let you specify a time window but defaults to two years (1 year before and 1 year after the reception of an event). For best performance, you should limit event cache searches to the smallest reasonable time window.
- These predicates are loaded at event server start-up when a TEC_Start event triggers the rule from which they are called.

The following table lists and describes the correlation predicates for searching the event cache for correlating related events.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_clear_targets</td>
<td>Returns all events in the cache that the specified clearing event clears.</td>
</tr>
<tr>
<td>any_clear_target</td>
<td>Returns the first event in the cache that the specified clearing event clears.</td>
</tr>
<tr>
<td>any_clearing_event</td>
<td>Returns the first event in the cache that clears an event.</td>
</tr>
<tr>
<td>first_causal_event</td>
<td>Searches the event cache for the root cause event related to an effect event.</td>
</tr>
<tr>
<td>first_effect_event</td>
<td>Searches the event cache for the effect event related to a cause event.</td>
</tr>
<tr>
<td>first_related_event</td>
<td>Searches the event cache for the logically earliest event related to a reference event.</td>
</tr>
<tr>
<td>is_clearing_event</td>
<td>Tests whether an event has been defined as a clearing event with the create_clearing_event or create_event_sequence predicate.</td>
</tr>
</tbody>
</table>
Putting it all together

This section provides examples of rules that implement the topics previously described in this chapter. For additional details about each of the correlation predicates shown in this section, see the particular predicate in the Alphabetic listing of rule language predicates beginning on page 77.

Creating the event relationship information

The following rule triggers on a TEC_Start event (the rule spans two pages). A TEC_Start event is generated when the event server starts up. This rule defines all of the correlation information used in the preceding examples in this chapter and loads it into the rule engine’s knowledge base.

```
rule: 'create_event_sequences':
{
  event: ev of_class 'TEC_Start',

  reception_action: 'create_sequences':{
    create_event_sequence(
      ['upsOnBattery',
       'lowBattery',
       'upsDischarged',
       'universal_host'],

      ['hostname', ['status', 'outside', ['CLOSED']]],
      clears('powerRestored', [], ['upsOnBattery'], []),
      clears('returnFromLowBattery', [], ['lowBattery'], []),
      clears('dischargeCleared', [], ['upsDischarged'], []),
      clears('universal_host',
        [ ['severity', equals, 'HARMLESS'] ],
        'universal_host',
        []),
      attr_condition('universal_host',
        ['severity', equals, 'FATAL']),
      attr_exception('hostname', 'universal_host',
        'probe_arg')
    ),

    create_event_sequence(
      ['cpqTape3PhyDrvStatusChange'],
      ['hostname', ['status', 'outside', ['CLOSED']]],
      attr_sequence(
        'cpqTape3PhyDrvStatusChange',
        'cpqTapePhyDrvCondition'=['Degraded', 'Failed']),
      clears(  
        'cpqTape3PhyDrvStatusChange',
        [ ['cpqTapePhyDrvCondition', equals, 'OK'] ],
        [ 'cpqTape3PhyDrvStatusChange' ]
      ),
    ),

    create_clearing_event(
      'CiscoLinkUp', [ ], ['CiscoLinkDown', ['origin'], no],
    ),

    create_clearing_event(
      'cpqDa3PhyDrvStatusChange',
      [ ['cpqDaPhyDrvStatus', equals, 'OK'] ],
      ['cpqDa3PhyDrvStatusChange'],
      ['hostname', ['cpqDaPhyDrvStatus', not equals, 'OK'] ],
      no),
  }
}
```
Rules for processing the events

The two rules shown in this section handle all of the events defined in the rule in “Creating the event relationship information” on page 225, as well as any other events that require the same processing.

process_problem_events rule: The process_problem_events rule handles all problem events (non-clearing events). It triggers on every leaf class because it specifies the base class in the event filter. (This can be changed to restrict the rule to trigger on events from a specific component if necessary.) The rule logic is as follows:

1. The check_for_clear action checks that the incoming event is not a clearing event. If it is, processing skips to the next rule. If it is not a clearing event, a search of the event cache is conducted for a related clearing event that has been previously received. If a previously received clearing event is found, the event under analysis is closed, the administrator attribute is updated to reflect that a rule closed the event, and processing exits.

   Note: Searching of the event cache for a previously received clearing event that arrived before any of its related problem events utilizes a large amount of system resources. This processing should only be used in those extremely uncommon environments where a monitor or network delays might cause clearing events to arrive at the event server before their related problem events.

2. The duplicate_detect action searches the event cache for duplicate events that have been received within one hour. If a duplicate is found, its repeat_count attribute is incremented, the event under analysis is dropped, and processing exits.

3. The check_for_prior_cause action searches the event cache for the root cause related event that has been received within the last hour. If the related cause event is found, the event under analysis is linked to it as a related effect event, the status of the effect event is set to acknowledged, the existing trouble ticket is updated, and processing exits. This and the next action assume a script exists to update trouble tickets.

4. The check_for_effect action searches the event cache for the first effect event that has been received within the last hour. (First effect event meaning the logically earliest as defined event in an event sequence, not order of arrival at the event server). If an effect event is found, it is linked to the event under analysis as a related effect event, its status is set to acknowledged, the existing trouble ticket is updated, and processing exits.

5. The open_trouble_ticket action is run only if none of the other actions succeed. If this action runs, it means the event under analysis is the first notification of a problem. Therefore, a trouble ticket is created and an administrator is notified. This action assumes that the operator has created the appropriate script in scripts/create_trouble_ticket.sh.

Some points worth noting about this rule are:

- Although the check_for_prior_cause and check_for_effect actions search for only the root cause and first effect events, respectively, all related incoming events are
linked appropriately regardless of the order received at the event server, provided this processing is performed on every incoming event. That is, every effect event is linked to a cause event.

- Because the rule conducts event cache searches for both cause and effect events appropriately, all the actions are reception actions. No events are submitted for redo analysis. Redo analysis is inefficient.

```rust
class: 'process_problem_events':(
  event: _ev of_class 'EVENT',

  reception_action: 'check_for_clear':(
    is_clearing_event(_ev),
    commit_action
  ;
    (any_clearing_event(_ev, _clr, 3600, 0),
     set_event_status(_ev, 'CLOSED'),
     change_event_administrator(_ev,
      'Event Processing Rule')
    ;
    commit_set
  )
  ),

  reception_action: 'duplicate_detect':(
    first_duplicate(_ev, _dup_ev where [
      status: outside ['CLOSED']]
    ,
    _ev -3600 -0),

    add_to_repeat_count(_dup_ev, 1),
    drop_received_event,
    commit_set
  ),

  reception_action: 'check_for_prior_cause':(
    prior_causal_event(_ev, _cause, 3600, 0),
    link_effect_to_cause(_ev, _cause),
    set_event_severity(_ev, 'ACK'),
    exec_program(_ev, 'scripts/update_trouble_ticket.sh',
     '%ld', [_cause], no),
    commit_set
  ),

  reception_action: 'check_for_effect':(
    first_effect_event(_ev, _effect, 3600, 0),
    link_effect_to_cause(_effect, _ev),
    set_event_severity(_effect, 'ACK'),
    exec_program(_ev, 'scripts/update_trouble_ticket.sh',
     '%ld', [_effect], no),
    commit_set
  ),

  reception_action: 'open_trouble_ticket':(
    exec_program(_ev, 'scripts/create_trouble_ticket.sh',
     '%ld', [_event], no),
    exec_program(_ev, 'scripts/notify_admin.sh',
     'admin_group', [], no),
    commit_set
  )
).
**process_clearing_event rule:** The process_clearing_event rule handles all clearing events in the environment. It must follow the process_problem_events rule, either in the same rule set or a subsequently-loaded rule set.

If an incoming event is a problem event (that is, not a clearing event), it encounters one of the commit_set predicates in the process_problem_events rule, and the process_clearing_event rule does not run.

If an incoming event is a clearing event, the check_for_clear action in the process_problem_events rule causes processing to skip the rest of that rule and resume processing with this rule. Only clearing events are processed by this rule.

This rule triggers on every leaf class because it specifies the base class in the event filter. (This can be changed to restrict the rule to trigger on events from a specific component if necessary.)

The rule logic searches the event cache for all related events this clearing event can clear that were received within the last hour. If related events are found, their severity is set to CLOSED, the administrator attribute is updated to reflect that a rule closed the events, and processing stops.

```plaintext
rule: 'process_clearing_event':(
    event: _clr_ev of_class 'EVENT',
    reception_action: 'clear_all_targets'(isclearing_event(_ev),
      all_clear_targets(_clr_ev, _target, 3600, 0),
      set_event_status(_target, 'CLOSED'),
      change_event_administrator(_target, 'Clearing Event Rule')),
    reception_action: 'exit' {
      commit_set
    }
).
```

**Alternatives:** There are many different ways to develop rules. The rules shown in this section are alternatives to the two rules previously described. They perform much of the same processing but are more compact and efficient.

The process_clearing_event rule is triggered only by clearing events. It performs remaining processing just like the rule in the "process_clearing_event rule". If the event is not a clearing event, this rule fails.

The process_problem_events rule is triggered only if the process_clearing_event rule did not run. It performs remaining processing just as the rule in "process_problem_events rule" on page 226 except it does not check for a clearing event when a problem event is received. It also uses the first_related_event predicate instead of the first_causal_event and first_effect_event predicates. This is more efficient because only one search of the event cache is performed.

**Note:** These rules span two pages.
set_event_status(_target, 'CLOSED'),
change_event_administrator(_target, 'Clearing Event Rule')
;
commit_set
)
).

rule: 'process_problem_events':{
  event: _ev of_class 'EVENT',
  reception_action: 'duplicate_detect'(
    first_duplicate(_ev, event: _dup_ev
      where [status:outside ['CLOSED']],
      _ev -3600 -0),
    add_to_repeat_count(_dup_ev, 1),
    drop_received_event,
    commit_set
  ),
  reception_action: 'check_for_related':(
    first_related_event(_ev, _related, _type, 3600, 0),
    _type == 'c',
    set_event_status(_ev, 'ACK'),
    link_effect_to_cause(_ev, _related)
    ;
    set_event_status(_related, 'ACK'),
    link_effect_to_cause(_related, _ev)
  ),
  exec_program(_ev, 'scripts/update_trouble_ticket.sh', '%ld', [_related], no),
  commit_set
  ),
  reception_action: 'open_trouble_ticket':(
    exec_program(_ev, 'scripts/create_trouble_ticket.sh', '', [], no),
    exec_program(_ev, 'scripts/notify_admin.sh', 'admin_group', [], no),
    commit_set
  )
}).

Chapter 5. Correlation examples  229
Chapter 6. Testing, tracing, and profiling rules

The IBM Tivoli Enterprise Console product provides the following functions to help you test, debug, and analyze rules:

- Commands and programs to test rules using simulated events
- A tracing function and profiling function to analyze the execution of rules

Testing rules

You can test rules using:

- The \texttt{wpostemsg}, \texttt{postemsg}, \texttt{wpostzmsg}, and \texttt{postzmsg} commands
- The tec\_agent\_demo program
- The SendEvents program

Sending events with commands

You can test rules with the \texttt{wpostemsg} and \texttt{postemsg} command-line commands. These commands send events whose characteristics are specified as arguments. The \texttt{wpostemsg} command simulates an event from a TME adapter and uses Tivoli Management Framework services for communication with the event server. The \texttt{postemsg} command simulates an event from a non-TME adapter and does not use Tivoli Management Framework services for its communication with the event server. See the \textit{Tivoli Enterprise Console Reference Manual} for additional information about these commands.

Sending events with the tec\_agent\_demo program

The tec\_agent\_demo program sends events you create within event files. Each event file contains only one event. The events to send are specified in a control file, which the program uses to prompt you before sending each event. You can create test scenarios for your rules by grouping events in specific control files and then specifying the control file as an argument to the program. The program and the files it uses must be run on the event server host.

Configuration prerequisites for using the tec\_agent\_demo program

The tec\_agent\_demo program reads a control file that lists the names of files containing the events to send. The control file must be named events\_list. The following example shows the contents of a control file.

\begin{verbatim}
TEC_Start
NT_NAV_start
NT_NAV_stop
NT_Perf_Alert
TEC_Stop
\end{verbatim}

Each event must be contained within a single file, referred to as an event file. The control file and event files can be in the same directory or different directories. If they're in different directories, you must specify the paths to the event files listed in the control file; for example, /test/TEC_Start.

The example on page 232 shows the contents of the NT_Perf_Alert event file. The other event files are formatted similarly but contain different events for this particular test scenario. The following is the syntax for event files:
You can specify as many or as few attributes as you need for an event. Default values provided by the event server are filled in where appropriate by the event server.

The delimiter between the event class name and each attribute is a semi-colon (;).

You can use white space in an event file as needed for readability.

Single line comments can be inserted following a number sign (#) character.

Each event file must end with the END keyword.

```
NT_Performance_Alert;
hostname=mfoster; origin=146.84.39.103; category=0;
eventType=Information;sid=N/A;sub_source=PerfMon; id=2000;
msg="\MFOSTER ; Object: Processor ;
Counter: % Processor Time ; Instance: 0 ; Parent: ;
Value: 13.586 ; Trigger: > 1.000';
date='Apr 29 14:36:34 2000';
sub_origin=mfoster;computer=\MFOSTER;
END
```

See “Creating event files from events in the reception log” on page 233 for information about an easy way to create event files.

**Using the tec_agent_demo program**

To send events:

1. From a bash or UNIX shell command line, enter the following command after initializing the Tivoli environment (for example, after issuing the `setup_env.sh` command)
   ```
   export TEC_BIN_DIR=$BINDIR/TME/TEC
   ```

2. Enter the following command to send the events listed in the control file:
   ```
   $TEC_BIN_DIR/tec_agent_demo -data /control_file_dir
   ```
   The `control_file_dir` is the directory where the control file is located. The control file must be named `events_list`. The program displays each event that it is ready to send.

3. Press Enter to send each event. The event is sent to the event server for processing. To exit the simulator program before all of the events listed in the control file are sent, press Ctrl+c.

**Sending events with the SendEvents program**

The SendEvents program sends events that have been retrieved from the reception log and written to a specific directory structure. The events are sent using a time interval. The interval can be used to do the following:

- Send an event every x number of seconds
- Send the events at the same interval as they were received originally

The SendEvents program expects the same directory structure as shown on page 234, including the `events_list` and `time_list` files. Use the procedure in “Creating event files from events in the reception log” on page 233 to create the appropriate directory structure. You can specify which directory to write the events retrieved from the reception log.

The SendEvents program and the files that it uses must be run on the event server host.

To send events, do the following:
1. Ensure that your Tivoli environment is initialized (for example, the `setup_env.sh` command has been issued).

2. From a bash or UNIX shell command line, enter the following command. The arguments are described after the example.

   ```bash
   $BINDIR/TME/TEC/contrib/SendEvents.pl -d directory -t time
   
   -d directory  
   The directory where event folders, events_list file, and time_list file are located. This argument is required.
   
   -t time  The time interval, in seconds, to send the events. This argument is optional. If it is not specified, the events are sent using the same interval as they were received.
   
Creating event files from events in the reception log

A program is available to parse the events in the reception log for use with the tec_agent_demo or the SendEvents programs. This procedure can be a quick way for you to create events for testing rules.

To create events for testing from those in the reception log:

1. Ensure your Tivoli environment is initialized (for example, the `setup_env.sh` command has been issued).

2. From a bash or UNIX shell command line, enter the following command. The directory argument specifies the directory to write the event folders (which contain event files), events_list file, and time_list file.

   ```bash
   wtdumprl | $BINDIR/TME/TEC/contrib/ParseEvents.pl -d directory
   
   The event files are generated in a directory structure like the following figure. For this example, the preceding command was issued with the `-d /test3`
Each event file is within an event folder. For example, the event0001 file is within the event0001 folder. The numbering sequence is internally generated by the ParseEvents.pl program and begins with the oldest event; that is, the event in the event0001 file was received before the event in the event0002 file.

Once you have captured the events in event files, you can rename them, modify them, and move them for use with the tec_agent_demo program. Or, you can use the directory structure as it is with the SendEvents program.

### Tracing rules

To trace rules, you must first specify a trace file using the `wsetesvrcfg –t` command or the Server Parameters window (see the *IBM Tivoli Enterprise Console Command and Task Reference* for more information.)
Next, compile the rule base with tracing enabled. This can be done from the command line with the *wrb –comprules –trace* command, with a trace directive specified in a rule or rule set, or from the rule builder by selecting *Trace Rules* in the Compile Rule Base dialog. When compiling is completed, load the rule base. Ensure you select *Trace Rules* in the Event Server Parameters dialog before stopping and restarting the event server to begin logging of the trace output.

When they run, rules generate trace output in a log file. You can examine the trace file to analyze and debug rule execution. The name and location of the trace file is set in the Event Server Parameters dialog. The default trace file is /tmp/rules.trace. The *tail –f* command is useful for viewing the trace file as events are received.

**Note:** Tracing should be disabled when a rule base is compiled for the production environment, because it uses system resources.

**Trace granularity**

The following levels of granularity are supported for rule tracing:

**All rules**

All rules within a rule base are traced when tracing is enabled by the *wrb –comprules –trace* command or by selecting the *Trace Rules* option when compiling a rule base with the rule builder.

**Rule set**

A particular rule set is traced by inserting a trace directive into the rule set. Insert it at the top before the first rule.

**Rule**

A particular rule is traced by inserting a trace directive into the rule. Insert it before the rule’s event filter.

**Tracing a rule set**

To trace a rule set, insert the trace directive into the rule set. Insert it at the top before the first rule. The following example shows how to enable tracing in a rule set:

```
directive: trace  % Start trace.
rule: rule1: (   % End rule1.
  ...   % End rule2.
rule: rule2: (   % End rule2.
  ...   % End rule3.
rule: rule3: (   % End rule3.
  ...   % End rule set.
  % End trace.
```

**Tracing a rule**

To trace a particular rule, insert the trace directive into the rule. Insert it before the rule’s event filter. The following example shows how to enable tracing for a particular rule:

```
rule: test_rule: (   directive: trace,
  event: _evt of_class within [ 'NT_NAV' ] where [ ],
```

Chapter 6. Testing, tracing, and profiling rules  235
reception_action: action0:(
    drop_received_event
)
).

Trace information

Trace information is written to the trace file for the following:

Rules When a rule is entered and exited. Only a rule whose event filter matches
the event under analysis is traced.

Attribute conditions
When an attribute condition of an event filter is entered and exited.

Actions
When a rule action is entered and exited.

Rule language predicates
When entering and exiting a rule language predicate, and when a predicate
fails.

Prolog predicates
When a predicate written in Prolog within a rule action is entered and
exited. Prolog predicate tracing is limited to one depth level; nested Prolog
calls are not traced. (To trace nested calls, include the directive
set_detailed_debugging(on) in the startup rule of the rule set.)

Rule tracing example

The following rule illustrates rule debugging. Assume that tracing is enabled and
the rule is contained in the rule set my_first_set:

rule: my_first_rule: (  
    description: 'Simplerule',  
    event: _ev of_class 'HIGH_CPU_USAGE'
        where [usage: _usage,
               hostname: equals 'my_server'],

    action: auto_ack: (  
        set_event_status(_ev, 'ACK'),
        set_event_administrator(_ev, 'john')
    ),

    action: page_administrator: (  
        exec_program(_ev,
            send_cpu_usage_to_pager,
            'john my_server %d',
            _usage],
            'NO'
    )
)
).

Tracing a rule event filter

The following trace line indicates that the my_first_set rule set has been entered:

[117] => rule set my_first_set

Note: Each trace line begins with the trace line number enclosed in brackets.
The following trace line indicates that the rule my_first_rule has been entered:

[118] => rule my_first_rule
event : 0x2c0b88 of_class HIGH_CPU_USAGE
Note: The incoming event must match the specified event class for the event to be traced.

The following trace lines indicate that the attribute conditions of the rule’s event filter are being checked. These trace lines indicate that the event satisfies the attribute conditions:

```
[119] call condition
[120] call usage: _125
[121] exit usage: 95
[122] call hostname: _126
[123] exit hostname: my_server
[124] call hostname: my_server equals my_server
[125] exit hostname: my_server equals my_server
[126] exit condition
```

Lines 120–123 indicate that the value of two attributes are being retrieved (usage and hostname), while lines 124 and 125 indicate that a retrieved attribute value for hostname is being compared to a literal value (my_server) for equality.

Variables are traced by their assigned internal number, not their name. The internal number is preceded by an underscore character (_). A variable is always used when retrieving an attribute value, even if you don’t store the value in a named variable.

The following trace lines indicate that a HIGH_CPU_USAGE event for a host other than my_server (another_host) was received. This event did not match the attribute conditions specified in the event filter:

```
[119] call condition
[120] call usage: _125
[121] exit usage: 95
[122] call hostname: _126
[123] exit hostname: another_host
[124] call hostname: another_host equals my_server
[125] fail hostname: another_host equals my_server
[126] fail condition
```

When a failure occurs, no further attribute condition testing is performed, and thus no further tracing of attribute conditions occurs for this rule.

Note: Rule statement execution is run sequentially. Because the first comparison that fails causes rule evaluation to stop, performing attribute comparisons before attribute assignments results in better rule engine performance.

**Tracing rule actions**

If an event satisfies all of the conditions specified in the event filter, the rule actions are run and traced. Actions are traced at the following two levels:

- Tracing is performed when entering or exiting an action.
- Tracing is performed when entering or exiting a predicate or an event filter in a predicate.

The following is an example of the trace file output for two actions (auto_ack and page_administrator). The auto_ack action contains two predicates (set_event_status and set_event_administrator). The page_administrator action contains the exec_program predicate.

```
[127] call action auto_ack
[128] call set_event_status(0x20b88, ACK)
[129] exit set_event_status(0x20b88, ACK)
[130] call set_event_administrator(0x20b88, john)
```

Chapter 6. Testing, tracing, and profiling rules   237
[131] exit set_event_administrator(0x2c0b88, john)
[132] exit action auto_ack
[133] call action page_administrator
[134] call exec_program(0x2c0b88, send_cpu_usage_to_pager, 'john my_server %d', [95], NO)
[135] exit exec_program(0x2c0b88, send_cpu_usage_to_pager, 'john my_server %d', [95], NO)
[136] exit action page_administrator

Note: Tracing does not include program or task execution status, like the send_cpu_usage_to_pager program in the preceding example. A predicate that calls a program or task, in this example the exec_program, can succeed even if the program or task fails.

As mentioned previously, variables are initially traced by their assigned internal number, not by their name. Once a variable has received a value, the value is used instead of the number (see trace line 134, which shows the value of my_server used instead of _126).

If a predicate called from within an action fails, the failure is reported in the trace file output and any remaining predicates in the action are not traced, because they are not run. A predicate can fail for any number of reasons, one of the more common being incorrect argument values.

Tracing multiple solution event filters in predicates
When a redo request is performed on a rule that contains an action that searches the event cache for all duplicates or all occurrences of an event, possible additional solutions are examined. If the event satisfies the event filter criteria of the predicate, a new solution for the event results.

If the action failed, it indicates that no other possibility successfully exited, that no other event satisfying the event filter criteria for the predicate was found, or that the predicate call failed.

The following action is the basis for the information that follows:
action: ( all_instances(event: _nfs_ev of_class 'NFS_SERVER_NOT_RESPONDING' where [server: equals 'Pascal']),

set_event_status(nfs_ev, 'CLOSED')
)

Assume two NFS_SERVER_NOT_RESPONDING events for the server Pascal had were received and are in the event cache.

Lines 139 through 144 contain the trace information for the first event found by the all_instances predicate:
[139] call reception_action action_1
[140] call all_instances(event : _366 of_class NFS_SERVER_NOT_RESPONDING where [server: equals Pascal], 0x2c0b88-600-600)
[141] exit all_instances(event : 0x2c0ae0 of_class NFS_SERVER_NOT_RESPONDING where [server: equals Pascal], 0x2c0b88-600-600)
[142] call set_event_status(0x2c0ae0, CLOSED)
[143] exit set_event_status(0x2c0ae0, CLOSED)
[144] exit reception_action action_1
Lines 145 through 150 contain the trace information for the second event found by the all_instances predicate:

[145] redo reception_action action_1
[146] redo all_instances
[147] exit all_instances(event : 0x2eabf0 of_class NFS_SERVER_NOT_RESPONDING where [server:equals Pascal],0x2c0b88-600-600)
[148] call set_event_status(0x2c0ae0, CLOSED)
[149] exit set_event_status(0x2cae0, CLOSED)
[150] exit reception_action action_1

Lines 151 through 154 contain the trace information generated when no additional events are found by the all_instances predicate:

[151] redo reception_action action_1
[152] redo all_instances
[153] fail all_instances(event : _366 of_class NFS_SERVER_NOT_RESPONDING where [server:equals Pascal],0x2c0b88-600-600)
[154] fail reception_action action_1

Profiling rules

Profiling generates a report that contains rule execution information. You can profile an entire rule base, rule sets, or particular rules. A report contains the following information for each rule being profiled:

• The amount of time (in seconds) spent by the rule to process the last event that triggered the rule
• The number of events processed by the rule
• The amount of time (in seconds) all events spent in the rule for processing
• The throughput of events for the rule, expressed as the number of events per second

Note: Profiling should be disabled when a rule base is compiled for the production environment, because it uses system resources.

The following figure shows an example of a profile report with one rule profiled:

```
---------------------------------------------
  Timing Summary
---------------------------------------------
test_rls:
  Time for last Event: 7.000000000000001e-02
  Event Count: 2
  Total Time: 4.799999999999998e-01
  Events per second: 4.166666666666669e+00

```

To profile rules, you must compile the rule base with profiling enabled. This can be done from the command line with the `wrb -comprules -profile` command or with the profile directive specified in a rule set or rule.

After recompiling the rule base with profiling enabled, stop and restart the event server to begin the profiling. The profile report is appended to the `$DBDIR/tec/profile` file when you shut down the event server. Because a profile report is always appended to the same file it can become quite large if you never delete it or delete entries within it, so check it periodically.
Profile granularity

The following levels of granularity are supported for rule profiling:

All rules
All rules within a rule base are profiled when profiling is enabled by the `wrb --comprules --profile` command.

Rule set
A particular rule set is profiled by inserting a profile directive into the rule set, at the top before the first rule.

Rule
A particular rule is profiled by inserting a profile directive into the rule, before the rule’s event filter.

Profiling a rule set
To profile an entire rule set, insert the directive before the first rule in the rule set. The following example shows how to enable profiling in a rule set:

```
directive: profile % Start profiling.
rule: rule1: (  
  ... 
  ). % End rule1.
rule: rule2: (  
  ... 
  ). % End rule2.
rule: rule3: (  
  ... 
  ). % End rule3.
  % End rule set.
% End profiling.
```

Profiling a rule
To profile a particular rule, insert the profile directive into the rule, before the rule’s event filter. To suppress profiling of a particular rule, insert the profile_off directive at the same location. The following example shows how to enable profiling for a particular rule:

```
rule: test_rule: (  
  directive: profile,  
  event: _evt of_class within [ 'NT_NAV' ] where [ ],  
  reception_action: action0:{  
    drop_received_event
  }
).
```

Measuring event processing performance

You can measure event processing performance on the event server by generating a report of event arrival and processing during a user-defined sample period. This report also includes the overall count of events received and processed. You can also specify a time interval to be used for periodically calculating throughput during the sample period.

Events received by the event server are inserted into the reception log with a state of QUEUED. When processing by the event server is complete, the state of the event in the reception log is updated to PROCESSED. To create a report of this processing activity, add the following two parameters to the .tec_config configuration file:
where \( report\_period \) is an integer specifying the rate at which processing rates are printed, and \( sample\_period \) is an integer specifying the time window for which event arrival and processing rates are computed. After you add these parameters, stop and restart the event server.

If either of these parameters is specified in the configuration file, the output is printed to the tec_reception trace file. The following example shows the output produced (with time stamps removed):

```
Event Throughput Statistics
Reporting Interval is 2 seconds
Sample Interval is 60 seconds
Actual Period 8 seconds
Events Received: 592
Event Arrival Rate: 74.000000 events/second
Events Processed: 700
Event Processing Rate: 87.500000 events/second

Total Events Waiting: 0
Total Events Received: 6604
Total Events Processed: 5666
Processing Backlog: 938
```

The output is displayed in two sections. The first section displays the following information:

- The reporting and sample intervals specified in the configuration file
- The current cumulative time within the sample period
- The count of events received and the arrival rate
- The count of events processed and the processing rate (this is the event server processing throughput, in events per second)

The second section shows event-related statistics computed since the server was started, including the number of received, processed, and waiting events, as well as the current server backlog. (The backlog is the difference between the received and processed events.)
Chapter 7. Creating rules with the graphical rule builder

IBM Tivoli Enterprise Console rules are written using rule language predicates and Prolog built-in predicates. The IBM Tivoli Enterprise Console rule builder is a GUI that non-programmers with no knowledge of the rule language, Prolog, or of the rule engine can use to develop rules based on natural language.

Notes:
1. Manipulating rule bases with the rule builder does not provide functionality for managing rule base targets or rule packs, or for profiling rules.
2. Rule bases created with the `wrb –crtrb` command are distributed rule bases and have a default rule base target named EventServer. Any rule set imported into the rule base using the rule builder must then be imported into the EventServer rule base target, or any other rule base target, using the `wrb –imptgtrule` command in order for the rule set to be loaded by the rule engine.
3. The rule builder cannot be used to edit rule set files that are created with a text editor.
4. Rule set files that are created with the rule builder must not be manually edited.
5. You can use the `upgrade_gui.sh` command to convert rules created by the version 3.6.2 and earlier rule builder to the rule syntax supported by the version 3.7 and later version of the rule compiler. The converted rules take advantage of features implemented by the newer version of the compiler and are easier to read. If you convert rules with this command, you can no longer edit them with the rule builder—you must use a text editor. See the IBM Tivoli Enterprise Console Command and Task Reference for details about the `upgrade_gui.sh` command.

Overview of the rule builder

The rule builder eliminates the need to know the underlying rule language or Prolog. The rule builder uses a layered approach to bridge the gap between the high-level concepts that are presented to you and the actual mechanics involved in implementing these concepts. You can use the rule builder GUI to describe a rule in a way that resembles the rule in, natural language. This description is then provided to an internal rule generator. The rule generator generates the actual rules in a mixture of rule language predicates and Prolog statements.

Using the rule builder, you specify what should be done, without having to specify how. For example, you can specify that an event should be closed 60 seconds after it has been acknowledged. When the rule set is saved, the rule generator produces the actual code that is required to implement the rule, in this case by setting a timer and creating a timer rule in rule syntax.

Graphical rule builder considerations for international environments

The GUI rule builder does not support entering UTF-8 data and the rule compiler does not support compiling non-UTF-8 data. If you need to create a rule with the rule builder that contains non-English data, you can enter the non-English data in the rule builder in local encoding. Then, convert the rule set file into UTF-8 data with the `so-v` command before compiling the rule base. See the Tivoli Management Framework Reference Manual for additional information about the `wiconv` command.
Using the rule builder

To create a new rule base, or to add new rule sets to an existing rule base using the rule builder, use the process illustrated in the following figure. The callouts in the figure correspond to headings for tasks shown in the table after the figure.

1. **Start the Rule Builder**
2. **Create New Rule Base**
3. **Open Rule Base**
4. **Create New Rule Set**
5. **Name and Save Rule Set**
6. **Open Rule Set**

See “Creating rules in international environments” on page 33 and “BAROC files in international environments” on page 44 for information about valid UTF-8 data in rules and BAROC files, respectively.
Chapter 7. Creating rules with the graphical rule builder  245
After rule sets are added or edited and the rule base is saved, the rule base must be compiled, loaded, and made active before use by the event server.

If you modify any of the BAROC files in the rule base, you only need to stop and restart the event server after compiling and loading a rule base.

Note: In this section, to view a pull-down menu, place the cursor over the item of interest and click the left mouse button. To view a pop-up menu, place the cursor over the item of interest and click the right mouse button. See the Tivoli Management Framework User's Guide for instructions on how to navigate in the Tivoli desktop.

The following table lists the context and authorization role required to use the rule builder:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Context</th>
<th>Tivoli Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the rule builder</td>
<td>Event server</td>
<td>senior</td>
</tr>
</tbody>
</table>

### Starting the rule builder
To start the rule builder, right click on the Event Server icon on the Tivoli desktop. You cannot start the rule builder from the command line. The Event Server icon looks similar to the following example:

![Event Server Icon](image)

To start the rule builder, select Rule Bases from the Event Server pop-up menu to display the window. In this example, the loaded rule base (represented by the arrow) is the Default rule base, which is shipped with the product. You cannot edit the Default rule base. The Test rule base is a user-defined rule base. The loaded
rule base is represented by the icon on the left in the following figure:

![Rule Base Icon](image)

**Editing an existing rule base**

To edit an existing rule base, select **Edit Rules** from the pop-up menu of the rule base to edit. The following window is displayed.

![Rule Base Window](image)

The rule sets that exist for this rule base are displayed in the order that they were imported. If the rule set was created with the rule builder, the value in the Editable column is yes. Only rule set files with a value of yes can be edited with the rule builder.

**Creating a new rule set**

Use the following procedure to create a new rule set. For more information about rule sets, see “Rule sets and rule packs” on page 15.
1. From the T/EC Rule Base window, select **Rule Set->New Ruleset** to display the T/EC Rule Base window with a new rule set structure named **new_set**.

2. Highlight **new_set** in the Set Name text field at the bottom of the window.

3. Enter a name for the new rule set and press Enter.
   The new_set entry is replaced with the new rule set name and the entry remains selected in the Rule Sets scrolling list.

   ![Image of T/EC Rule Base window with new_set highlighted]

**Note:** Any text field that has a return character symbol requires a carriage return entered from the keyboard for the changes to take effect.

**Editing a rule set**

You can edit a rule set to create new rules in a new or existing rule set file, or to modify existing rules in an existing rule set file.
To edit a rule set, select **Rule Set->Edit Ruleset** in the TEC Rule Base window to display the Rule Set window. You are now ready to create a rule.

### Creating a simple rule

A simple rule is a rule that does not do any correlation with other events in the event cache, except that a simple rule can be used to filter duplicate events.

Simple rules pertain to only one event instance; however, the rule can contain multiple event classes, each of which would cause evaluation of the rule. For example, you might want all events related to `login` or `su` command attempts to be automatically assigned to a security administrator. There are several of these event classes, including `Logfile_Login` and its subclasses, `Logfile_Su` and its subclasses, and `Logfile_Passwd`. If the rule contains these superclasses, by default any leaf-node event of one of these classes causes the rule to be evaluated. If you also want the rule evaluated for non-leaf node events (containing event superclasses like `Logfile_Login` and `Logfile_Su` mentioned previously), you can specify the `fire_on_non_leaf` directive (see “fire_on_non_leaf” on page 69 for additional information).

**Note:** Once a rule is saved, the rule type (simple or compound) is fixed and cannot be changed later.

The parts of a simple rule are as follows:

- Description
- Event class
- Conditions
- Actions

Use the following procedure to create a simple rule:

2. Enter a brief descriptive comment about the rule in the Description text field. This description is used to identify the rule in other dialogs.

3. Use the following procedure to enter the event class. Event classes are described in Chapter 2, “Event class concepts”, on page 35.
   a. From the Simple Rule Window, click **Event Class** to display the Select Class window.
   b. Select a class by doing one of the following:
• Select the appropriate event classes from the Available Class(es) scrolling list and click the left arrow button to move the selected event classes to the Selected Class(es) scrolling list.

• In any of the windows where there is a text field, you can enter a partial word or substring and it is matched, if possible, in the scrolling list. For example, suppose you are writing a rule for all events related to the su command, whether successes or failures. If you enter the string su, all classes that contain su are selected. These include Logfile_Su, Root_Login_Success, Su_Success, Root_Login_Success_From, and Su_Failure. The pattern matching is not case sensitive.

c. To move an event class from the Selected Class(es) scrolling list to the Available Class(es) scrolling list, select the event class and click the right arrow button.

d. Click OK to apply your changes and to display the updated Simple Rule window. The event class filter condition is added.

![Simple Rule: New Simple Rule](image)

The rule that is being created is shown in the Rule Synopsis field.

4. Use the following procedure to enter the conditions. Attribute conditions are described in “Attribute conditions” on page 63. Using conditions, you can further restrict the events for which this rule is applicable by specifying values for certain attributes for the given event class. For example, using the event class of Host_Down, selecting a value for the hostname attribute causes this rule to evaluate to true for only Host_Down events whose host name is in the given list.

Event class definitions are found in the BAROC files for a rule base.
a. To specify the conditions that must exist in order for the rule to evaluate to true, click **Conditions** to display the Condition in Rule window.

![Condition in Rule: New Simple Rule](image)

b. Select the preferred attribute name from the Available Attribute(s) scrolling list.

c. Select one of the following options from the Relation menu:

   - **in list**: One or more values must be matched.
   - **none**: There must be no value.
   - **not in list**: None of the values must be matched.

d. Type the value in the Edit Value text field. Press the Enter key after each entry to move the value into the Attribute Value scrolling list. For the in list and not in list options, one or more values can be specified.

e. Click **Add** after each attribute, its relation, and all preferred values have been entered. The condition is shown in the Conditions Synopsis field. If the condition statement does not appear in this window, the attribute condition was not added.
f. To delete an attribute, select it from the Available Attribute(s) scrolling list, then click Delete. The selected attribute is removed from the Conditions Synopsis field.

g. To edit or re-enter an existing attribute, select the name from the Available Attribute(s) scrolling list. The current values are displayed. Re-enter any values that you want to change.

h. After entering all preferred attributes, click OK to display the updated Simple Rule window. The rule you are creating is shown in the Rule Synopsis field.

5. Enter the actions. Actions define a set of actions to take when an event meets the rule conditions. Use the following procedure to enter actions:

a. Click Actions in the Simple Rule window to display the Actions in Rule window.

<table>
<thead>
<tr>
<th>Rule Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>When to Run</td>
<td>Determines when the actions are started.</td>
</tr>
</tbody>
</table>
### Rule Action Description

| Action(s) | Specifies what to do when the event filter conditions are met. |

**Note:** You must specify when to run the action before you can specify any actions.

b. There are a number of activation points in the life cycle of an event at which an action can be run. To add an activation point, use the left-most Add option menu and select the preferred clause. The choices are as follows:

<table>
<thead>
<tr>
<th>When to Run</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>When event is received</td>
<td>Upon receipt of the event into the rule engine.</td>
</tr>
<tr>
<td>After event is received</td>
<td>(N) minutes after the event is received.</td>
</tr>
<tr>
<td>When severity is upgraded</td>
<td>After the severity attribute of the event is changed, either internally by a rule or from running the Change_Severity task from an event console.</td>
</tr>
<tr>
<td>When severity is downgraded</td>
<td>After the severity attribute of the event is changed, either internally by a rule or from running the Change_Severity task from an event console.</td>
</tr>
<tr>
<td>When event is acknowledged</td>
<td>Immediately after the event is acknowledged, either internally by a rule or from an event console.</td>
</tr>
<tr>
<td>After event is acknowledged</td>
<td>(N) minutes after the event is acknowledged.</td>
</tr>
<tr>
<td>When event is closed</td>
<td>Immediately after the event is closed, either internally by a rule or from an event console.</td>
</tr>
<tr>
<td>When frequency exceeds a limit</td>
<td>When the number of events received is greater than a specified limit.</td>
</tr>
</tbody>
</table>

c. Once an activation point is added, a new option menu is displayed under the When to Run heading. Select this option menu to view all of the conditions that have been selected.

d. Use the Change option menu to modify or replace an existing condition. Actions associated with this clause are preserved. For example, if you have an action that reads *When the severity is upgraded, launch a command,*
you can change the activation point to **When the event is acknowledged** and the **launch a command** action is still selected.

To remove a condition, select the appropriate clause in the When to Run list and click **Delete**. All actions associated with this condition are also removed.

e. Select the action(s) to perform from the right-most Add option menu. The choices are shown in the following table:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set severity</td>
<td>Specifies a new value for the severity attribute. Severity can be UNKNOWN, HARMLESS, WARNING, MINOR, CRITICAL, and FATAL.</td>
</tr>
<tr>
<td>Set status</td>
<td>Specifies a new value for the status attribute. Status can be OPEN, RESPONSE, ACK, or CLOSED.</td>
</tr>
<tr>
<td>Set message</td>
<td>Provides a text field to set a value for the msg attribute in the event. This can be an informational message or it can contain the value of another attribute.</td>
</tr>
<tr>
<td>Forward Event</td>
<td>Sends an event to a different event server. There must be a ServerLocation option specified in the tec_forward.conf file in the TEC_RULES subdirectory of the rule base. See the description of &quot;forward_event&quot; on page 160 for additional information about the tec_forward.conf file.</td>
</tr>
<tr>
<td>Drop Duplicate Event</td>
<td>Checks for the existence of a duplicate event in the event cache having a status other than CLOSED within a time window. If one exists, the repeat_count attribute of the existing event increments by one and the newly received event is deleted.</td>
</tr>
<tr>
<td>Launch a Task</td>
<td>Causes a task from a specified task library to be performed. A Task Selection window similar to the example below is displayed when you choose this action.</td>
</tr>
</tbody>
</table>

In the Task Selection window, select the task library to use by double-clicking on the task library name in the Libraries scrolling list. Then, highlight the task you want to run in the Tasks list and click **Select task**. Next, click **OK**. You are returned to the Actions in Rule window.

By default, tasks are run on the same node as the event server.

Proper access to a task library is necessary to use this action.
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch a Command</td>
<td>Causes a system command or shell script to run. A Select a Program window similar to the example below is displayed when you select this action. In the Select a Program window, enter the full or relative path to the command by typing the Path Name in the text field or by using the file browser. The default search path is $BINDIR/TME/TEC. Commands are run on the same node as the event server.</td>
</tr>
</tbody>
</table>

f. Once an action is added, a new scrolling list is displayed under the Action(s) heading. Click the up or down arrows to rearrange the list of actions. Select an action by clicking it. The following shows the portion of the window where the new scrolling list is displayed:
g. If a command or task takes parameters, click **Edit Arguments** to display the Edit Arguments window.

h. Type the format string into the Format string text field. Use quotation marks to force a string with white space (such as the value of the msg or date attribute) to be read as a single parameter. If the Format string text field is left blank, the rule builder treats each selected attribute as a single parameter. The command line parameters for a task are the same as for the `wruntask` command. Tasks and commands run by default on the node where the event server is running. See the *Tivoli Management Framework Reference Manual* for additional information about the `wruntask` command.

**Note:** If you select to launch a task and do not specify a format string in the optional Edit Arguments window for the task (that is, leave the Format string text field blank), the default arguments `-h event_server_hostname -l TaskLibraryName` are used. If you add any text to the Format string field, these two parameters must also be provided.

i. After entering all necessary activation points and actions, click **OK**. The values are stored and the changes are displayed in the Simple Rule window.

---

Chapter 7. Creating rules with the graphical rule builder 257
j. Click **OK** to display the Rule Set window and close the Simple Rule window.

Creating a compound rule

Using compound rules, you can specify a causal relationship between two event classes.

A compound rule pertains to multiple events of exactly two event classes. These events must have a cause and effect relationship—either one event causes another to be generated or one event causes another to be closed—and this relationship needs to be specified in the rule. For example, if a Host_Down event is generated for an NFS file server, it is likely to cause NFS_Server_No_Response events to be generated by the server’s clients. Once a relationship is specified, rules that apply to either the cause or effect events can be generated.

**Note:** Once a rule has been saved, the rule type (simple or compound) is fixed and can not be changed later.

There are three parts of a compound rule that need to be entered:

- Description
- Event classes
- Correlation

Use the following procedure to create a compound rule:
1. From the Rule Set menu, select Rule->New Rule->Compound to display the Compound Rule: New Compound Rule window.

Type a brief descriptive comment about the rule in the Description text field. This description is used to identify the rule in other dialogs.

2. Use the following procedure to enter the event classes. Event classes are described in Chapter 2, “Event class concepts”, on page 35.

a. Click Event Classes to display the Select Class window and select the classes for the events to correlate. By default, the first two event classes from the list of classes for the rule base are in the Selected Class(es) scrolling list.
b. If there are unneeded event classes in the Selected Class(es) scrolling list, select the unneeded classes and click the right arrow button to move them to the Available Class(es) window. Moved classes are displayed at the bottom of the list.

c. To select the two event classes for this rule, select them in the Available Class(es) scrolling list. Click the left arrow button to move these to the Selected Class(es) scrolling list.

d. Click OK to apply your changes and display the Compound Rule window. The rule builder alerts you if you selected two or more event classes. The rule that is being created is shown in the Rule Synopsis field.

3. Enter the correlation that provides the relationship between the two event classes.

Once the correlation relationship is established, the two events have a fixed position in the correlation dialog. If you listed the effect event first and the cause event second, use the is-caused-by or is-canceled-by correlation type relation. If you listed the cause event first and the effect event second, use the causes or cancels relation.

Rules that are written to handle multiple-linked events process all links regardless of the order of event arrival. For example, an NFS_No_Response event is caused by a Host_Unreachable event. This Host_Unreachable event is closed by a Host_Up event. When the Host_Up event is received, the NFS_No_ Response status is set to closed as the Host_Unreachable status is changed. The rule builder properly associates these events regardless of the time of receipt of these events in the event cache, except if the events have been removed from the cache due to aging.

To always update an event with the latest values, you can specify the same class on both sides of the cancels (or is canceled by) relation. This is particularly useful for Distributed Monitoring events.
a. To enter the correlation, click **Correlation** to display the Correlation in Rule window.

![Correlation in Rule: New Compound Rule](image)

b. Select one of the following options from the Correlation Type menu:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>causes</td>
<td>This links the event on the right to the event on the left. The values of the two attributes, date_reception and event_handle, from the cause event (in the screen example this is the Host_Unreachable event) are written in the cause_date_reception and cause_event_handle attributes of the effect event (in the example, this is the NFS_No_Response). Whenever either status (effect event or cause event) is changed, the other is updated.</td>
</tr>
<tr>
<td>is caused by</td>
<td>This links the event on the left to the event on the right. The values of the two attributes, date_reception and event_handle, from the cause event are written in the cause_date_reception and cause_event_handle attributes of the effect event. The status of the cause event is written to the status of the effect event. Any changes to the status of the cause event are automatically reflected in the status of the effect event.</td>
</tr>
<tr>
<td>cancels</td>
<td>The reception of the event on the left closes the event on the right.</td>
</tr>
<tr>
<td>is canceled by</td>
<td>The reception of the event on the right closes the event on the left.</td>
</tr>
</tbody>
</table>

c. A default time window of five minutes is provided. To replace the default value, select it, type the new value, and click the Enter key. Delete any entry in the text field if you do not want to limit the event search to a specific time window.

**Note:** While specifying a time window results in a more efficient search of the event cache and is generally recommended, in some instances you might want to eliminate the time window. For example, suppose there is a Host_Up event that should close any Host_Unreachable events for the same host name. It might be reasonable not to specify a time window, since the Host_Up event signals that this host is now in working order.

d. Specify additional values to identify in the Comparison dialog. The Comparison dialog restricts rule evaluation to only those events that are members of the specified event classes and whose attributes meet the specified criteria.
Select an attribute name for the first event from the scrolling list on the left to display the attribute name in the text window. The event class for the event is shown directly above the attribute text window.

Select an attribute name for the second event from the scrolling list on the right to display the attribute name in the text window. The event class for the event is shown directly above the attribute text window.

Click **Add** to apply the changes and show the updated part of the rule in the Correlation Synopsis field.

Repeat steps e, f, and g for each comparison that is needed.

Click **OK** to apply these changes and display the updated Compound Rule window. The rule being created is shown in the Rule Synopsis field.

Click **OK** to apply the changes and return to the Rule Set window. The new rule is added to the list.

### Saving a rule set

Use the following procedure to save a rule set:
1. From the Rule Set window, select **Rule Set->Close** to display the TEC Rule Base window for the rule base.

![Rule Set window](image)

2. Select **Rule Base->Save** to save the rule set file. When saving the rule base, the rules that were created using the rule builder are generated.

3. Select **Rule Base->Close** to display the Confirm window.

![Confirm window](image)

4. Click **Yes** to save the rule set and return to the Event Server Rule Bases window.

![Event Server Rule Bases window](image)

**Compiling a rule base**

After a rule set has been updated and stored in the rule base, the rule base must be compiled to make the new rules accessible to the event server. Use the following procedure to compile a rule base:
1. In the Event Server Rule Bases window, select **Compile** from the appropriate rule base pop-up menu to display the Compile Rule Base window.

The Trace Rules option controls the rules tracing option. See “Tracing rules” on page 234 for additional information.
2. Click **Compile**. Output similar to the following is displayed.
3. Click **Close** to display the Event Server Rule Bases window.

![Event Server Rule Bases Window](image1)

4. To make the modified rule base accessible to the event server, select **Load** from the appropriate rule base’s pop-up menu to display the Load Rule Base window.

![Load Rule Base Window](image2)

5. Select **Load and activate the rule base**

6. Click **Load & Close**.

**Stopping and restarting the event server**

You only need to stop and restart the event server after compiling and loading a rule base if you modify any of the BAROC files in the rule base. Use the following procedure if you need to restart the event server:

1. From the Tivoli desktop, select **Shut Down** from the Event Server icon pop-up menu.
2. After receiving a confirmation message that the event server has shut down, select **Start-up** from the Event Server icon pop-up menu on the Tivoli desktop.

---

**Rule builder examples**

This section provides some examples showing how to use the rule builder to turn policy statements into rules.

### Defining an escalation policy

**Problem**

Your network operations center has a target of resolving critical problems reported on the production SAP R/3 system within 15 minutes and minor problems within one hour. If they cannot meet these targets, then they have to call in Level 2 support to assist them. You want to automate this escalation to the second level and send an e-mail when the time limits expire. You are using the Tivoli AMS Module for R/3 to monitor the R/3 production instance.

A concise statement of the operational policy is the following:

> For all events coming from the production R/3 system (SAP instance=PRD), escalate critical events to Level 2 if they are not resolved within 15 minutes and escalate minor events to Level 2 if they are not resolved within one hour.

**Solution**

**Get event information:** Getting at the information about the event is fairly easy. Each attribute or attribute of the event is passed as an environment variable to the response task (for example, the IP address where the event originated is passed as $origin, the name of the host is passed as $hostname, and so forth). The *IBM Tivoli Enterprise Console Command and Task Reference* describes the tasks shipped by IBM and the environment variables available to tasks.
In addition to all of the attribute values, two additional environment variables are set to allow tasks to be written generically to respond to a large number of diverse events. These variables are as follows:

$EVENT\_CLASS
Set to the class of the event.

$SLOTS
A list of all attributes by name.

For a Distributed Monitoring event that is checking the percent of disk space used, these variables are set to the following:

$EVENT\_CLASS=Sentry2\_0\_diskusedpct
$SLOTS=origin sub\_origin source sub\_source...

**Build the rules:** After you write and test the e-mail task, you can build the rules. The first part of the policy statement is as follows:

*For all events coming from the production R/3 system (SAP instance = PRD), escalate critical events to Level 2 if they are not resolved within 15 minutes...*

You can identify the production R/3 system in one of two ways as follows:

- List the IP addresses or host names of the machines that make up this instance and check for origin or host name in that list.
- Take advantage of the fact that the AMS Module for R/3 places the instance ID in the sub\_source attribute of the incoming event.

The events that interest you in this scenario are from Distributed Monitoring and from the R/3 event adapter. Distributed Monitoring events inherit from the base event class Sentry2\_0\_Base and R/3 event adapter events inherit from the base event class SAP\_Alert. Both of these event classes can be specified in a single rule.

The additional conditions to check for are for the severity and the status. You can determine the status of these conditions by checking the Conditions Synopsis field.
in the Condition in Rule dialog.

![Condition in Rule: New Simple Rule](image)

The action portion of this policy is to e-mail an administrator 15 minutes after the event is received. The e-mail action is implemented as a task and is selected from the available task libraries. Use the Actions in Rule dialog to set these conditions.

![Actions in Rule: New Simple Rule](image)

The completed rule for this portion of the policy looks like the following:

Event Class:  
[Sentry2_0_Base, SAP_Alert]

Conditions:  
sub_source in [PRD]  
status not in [CLOSED]  
severity in [CRITICAL, FATAL]

Actions:  
15 minutes after event is received  
send e-mail
The condition is checked when the event is received to set the timer and again when the timer goes off, so the action only happens if the status is still not equal to CLOSED.

The second part of the policy is implemented in the same way.

…and escalate minor events to Level 2 if they are not resolved within one hour.

Event Class:
[Sentry2_0_Base, SAP_Alert]
Conditions:
- status outside [CLOSED]
- severity in [WARNING]
Actions:
- 60 minutes after event is received
  - e-mail administrator
The rule set called Escalation_policy contains these two rules.

Monitoring for potential security breaches on UNIX endpoints

**Problem**
You want to monitor for certain events on UNIX endpoints that are essentially harmless unless they occur in a pattern of a certain frequency. In this particular case, you want to check for failures when a user attempts to log in as another user with the `su` command on your critical servers. A failure occurs when the user does not know the required password to switch to the new user identity. If this happens occasionally, you do not want the event console operators to even see it. However, if it happens more than 5 times in 10 minutes, you want the operator to know about it and an e-mail automatically sent to the security administrator.

A concise statement of the operational policy might look like the following:

> Automatically close all incoming Su_Failure events unless you get more than 5 of them from the same node within 10 minutes. In that case, upgrade the severity to a warning level, display it to the operator, and e-mail the security administrator.

**Solution**

**Modify default severity:** The UNIX logfile adapter is pre-configured to monitor the logfile `/usr/adm/sulog` file, which is where the switch user program writes messages for both successes and failures. By default, the adapter generates an Su_Failure event with a severity of WARNING whenever a user attempts to switch their user ID and does not correctly enter the required password.

In this case, you want to regard the Su_Failure event as a harmless event. To accomplish this, you must change the default severity of this event from WARNING to HARMLESS by modifying the `tecad_logfile.baroc` file in the TEC_CLASSES subdirectory of the rule base.

While you are looking at the BAROC file for this event, make a note of the attributes used for duplicate detection of this event (those with the `dup_detect` facet set to YES). These attributes are used for the frequency check in the rules you will build.

**Create the rules:** Once you have modified the default severity from WARNING to HARMLESS, you are ready to create the rules for enforcing your operational policy. The first part of the operational policy is expressed as follows:
Automatically close all incoming Su_Failure events...

This is the first rule to build for implementing this policy:

Event Class:  
Su_Failure

Conditions:

Actions:
When event is received
Set Status to [CLOSED]

This rule looks at all incoming events and immediately changes their status to CLOSED. If the console operators have the event console configured to not show CLOSED events, then they never see the incoming event.

The second part of the policy statement is as follows:

...unless you get more than 5 of them that are from the same node within 10 minutes. In that case, upgrade the severity to a warning level...

To implement this portion of the policy, add another action to the existing rule.

Event Class:  
Su_Failure

Conditions:

Actions:
When event is... received
Set Status to [CLOSED]
When event occurs 5 times within 10 minutes
Set Severity to [WARNING]

This action changes the fifth event within a 10-minute period that matches the dup_detect criteria to a WARNING level event. The attributes used for duplicate detection include the host where the event occurred, the user ID that was attempting to switch logins, the user ID that was being switched to, and the terminal identifier where the command was attempted. The five failed attempts must match on all of these attributes for this action to run.
The last portion of the policy statement then implements the recommended response based on the fact that the event has been upgraded from a harmless event to a warning event:

...and display it to the operator and e-mail the security administrator.

Adding another action to the existing rule implements this portion of the policy. This action sets the status to OPEN so that it displays at an event console and uses the task execution feature of the rules engine to launch the task to e-mail an administrator. The task can be configured with the e-mail address and the message to send.

Event Class:
   [Su_Failure]
Conditions:
   When event is received
       Set Status to [CLOSED]
   When event occurs 5 times within 10 minutes
       Set Severity to [WARNING]
   When severity is upgraded
       Set Status to [OPEN]
       e-mail administrator

The completed rule now implements the stated policy.

You can further define this policy to only include attempts to switch to particular privileged user accounts or only include critical servers by adding conditions to the rule. For example, if you only care about users attempting to switch to user ID root on corporate database servers, you can add the following conditions to the rule:

Event Class:
   [Su_Failure]
Conditions:
   hostname in [dataserv1, dataserv2]
Using Distributed Monitoring as a passive monitor

Problem
You are using Distributed Monitoring to monitor performance and availability of your systems. You want to show the latest value of the respective monitors to operators so that they do not become overburdened with out-of-date information. Additionally, you want to only show harmless events for a short period of time. This gives operators a visual queue that the condition has been resolved, but does not clutter their screens with these events.

Here is a statement of an operational policy that describes a potential use of Distributed Monitoring as a passive monitor for disk utilization:

*The operator console should be updated with the latest value of all Distributed Monitoring monitors; as a new event comes in, it should replace the old event. Once the Distributed Monitoring monitor returns to the normal range, the event should be automatically closed after five minutes so that it does not clutter the console.*

Solution

Configure the distributed monitoring monitors: The first step to implement this policy is configuring the Distributed Monitoring monitors to send the necessary events. For the purpose of this example, the available swap space is the threshold being monitored. Your policy might be to establish thresholds as follows:
<table>
<thead>
<tr>
<th>Severity</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>Less than 10 MB</td>
</tr>
<tr>
<td>Warning</td>
<td>Less than 20 MB</td>
</tr>
<tr>
<td>Reset to Normal</td>
<td>Greater than 30 MB</td>
</tr>
</tbody>
</table>

Note that some space is provided between the warning threshold and the reset threshold to ensure that we do not just keep alternating between the two values. The Distributed Monitoring configuration involves setting monitor thresholds that send IBM Tivoli Enterprise Console events at the above thresholds.

**Create the rules:** Now you can build the rules that implement the operational policy for Distributed Monitoring monitors. The first part of the policy is as follows:

*The operator console should be updated with the latest value of all Distributed Monitoring monitors; as a new event comes in, it should replace the old event.*

This describes a correlation between two events where one event is cancelled by another. In this case, the event doing the canceling happens to be of the same class as the one being canceled. In other cases it could be a different class (for example, a host_down event can be canceled by a host_up event). In order to make this rule apply across all of the events generated by Distributed Monitoring, you want to write it on the base event for all Distributed Monitoring events. That event class is named Sentry2_0_Base. All other event classes used in Distributed Monitoring are derived from this event class, so any rules written on the base class applies to all Distributed Monitoring events.

In the case where both events are of the same class, care needs to be taken to ensure that the incoming event does not cancel itself. You can do this by making sure that part of the correlation comparison includes a check on the event status. Putting this comparison in the rule makes it look like the following:

[Sentry2_0_Base] is canceled by [Sentry2_0_Base] within a 1440 minute period if:

[Sentry2_0_Base].status equals [Sentry2_0_Base].status

The time window is set to 24 hours, but any time window can be selected. If the status is not checked, the effect is that the old events are canceled by the incoming event and the act of closing an old event runs the action again. This then closes the incoming event, which is clearly not the preferred behavior. By including the status check, the incoming (open) event does not match against and old event once it has been closed. This is a good basis for starting any correlation rule that uses the is-canceled-by relationship.

You can develop the remainder of the rule by adding those attributes that are specific to the event class to identify duplicate events. The attributes that are of interest for determining duplicate Distributed Monitoring events are listed in the following table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>IP address of system being monitored</td>
</tr>
<tr>
<td>sub_origin</td>
<td>Name of the machine being monitored</td>
</tr>
<tr>
<td>source</td>
<td>SENTRY</td>
</tr>
</tbody>
</table>
### Attribute | Meaning
--- | ---
sub_source | Policy region containing the Distributed Monitoring profile
collection | Name of the Distributed Monitoring profile
monitor | Name of the individual monitor
probe_arg | Arguments passed to the monitor program

You define the correlation between two events by first selecting the type of the correlation. In this case, it is an is-cancelled-by correlation as opposed to a causes or is-caused-by-correlation. Additionally, you must enter a time window and a set of attributes on which to match.
The completed rule, taking into account all of these attributes, is shown in the Rule Synopsis pane in the following figure.

![Rule Synopsis](image)

The effect of this rule is that whenever a new Distributed Monitoring event comes in, it will immediately close out any similar events that were received within the last 24 hours. This keeps the operator up to date with the latest status.

The second part of the policy statement is as follows:

*Once Distributed Monitoring monitor returns to the normal range, the event should be automatically closed after five minutes so that it does not clutter the console.*

This is easy to implement using the rule generator. This is done by creating a simple rule that runs on the same event class (Sentry2_0_Base) five minutes after the event is received. The completed rule is shown in the Rule Synopsis pane in...
the following figure.

The completed rule set for the Distributed Monitoring policy includes the following two rules:

**Linking related events**

**Problem**

If a node goes down, you get multiple events from the NetView component. One event comes, indicating that the node is down, but additional events will also come in indicating that each of the interfaces to that node is down. You want to correlate the interface down events with the node down event and then automatically close them when the node comes back up.

A simple statement of this policy is as follows:

*Interface down events can be caused by Node down situations on the same node. If an interface down event can be attributed to a node down situation, then do so and close it automatically when the node comes back up.*
Solution
Something that makes this problem potentially difficult to solve is the fact that the events can be received in any order. The node down situation could be detected before the interface down event or vice versa and the rules must be able to handle either order. Fortunately, the rule generator is written to handle this automatically.

The rule for this situation is:

\[[\text{TEC\_ITS\_NODE\_STATUS}]\text{ causes }[\text{TEC\_ITS\_INTERFACE\_STATUS}]\]
within a 10 minute period if:
\[[\text{TEC\_ITS\_NODE\_STATUS}].nodestatus equals[\text{TEC\_ITS\_INTERFACE\_STATUS}].ifstatus\]
\[[\text{TEC\_ITS\_NODE\_STATUS}].hostname equals [\text{TEC\_ITS\_INTERFACE\_STATUS}].hostname\]

The code that is generated from this rule checks incoming events of both classes against the event cache for matching events within the last 10 minutes and assigns the cause and effect relationship. The effect of that relationship is that whenever the cause event is acknowledged or closed, all effect events have their status changed accordingly.
Part 2. Correlating events at the gateway or adapter
Chapter 8. Writing rules for the state correlation engine

The state correlation engine is a high-speed, rule-based event filtering and correlation service that runs at the Tivoli Enterprise Console gateway or adapter. Because the state correlation engine processes events before forwarding them to the event server, it can be used to reduce event traffic by discarding unwanted events or consolidating multiple events into summary events. This can significantly reduce workload on the event server by processing events closer to the source; resources on the event server can then be devoted to correlation that involves events coming from multiple gateways or adapters.

Correlation and filtering of events by the state correlation engine is controlled by rules. However, it is important to note that the state correlation engine is entirely distinct from the Prolog-based rule engine that runs on the event server. The following table summarizes some of the important differences.

Table 1. Differences between Prolog and state correlation rule engines

<table>
<thead>
<tr>
<th>Prolog rule engine</th>
<th>State correlation engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs at the event server</td>
<td>Runs at the gateway or adapter</td>
</tr>
<tr>
<td>Can correlate events from multiple gateways</td>
<td>Can correlate only events flowing through a single gateway, or from a single source</td>
</tr>
<tr>
<td>Uses rules written in Prolog-based rule language</td>
<td>Uses rules written in XML and actions implemented as Java classes</td>
</tr>
<tr>
<td>Can perform sophisticated correlation and establish causal links between events</td>
<td>Can perform a limited set of state-based correlation functions</td>
</tr>
<tr>
<td>Has access to historical events in the event cache</td>
<td>Works only with current events</td>
</tr>
</tbody>
</table>

State correlation rules

Correlation is achieved with state-based and stateless rules. You specify these rules by using XML syntax, defined by the supplied DTD file, tecsce.dtd. The location of the default XML file is $BINDIR/TME/TEC/default_sm/tecroot.xml. Sample XML rule files are provided in the /EIFSDK/samples/state_correlation directory on the IBM Tivoli Enterprise Console Non-TME Installations CD.

Most rules are defined using state machines. A state machine gathers and summarizes information about a particular set of related events. It is composed of states, transitions, summaries, and other characteristics, such as expiration timers and control flags.

There are five state-based rule types: duplicate, threshold, passthrough, resetOnMatch, and collector, all based on state machines. Each state machine looks for a trigger event to start it. There is one stateless rule type: match.

A state-based rule operates on a sequence of events arriving at the state correlation engine, whereas a stateless rule operates on a single, current event. A rule can contain the following elements:

- Predicates for matching events relevant to that rule
- Actions that run after the rule triggers
Attributes, such as a threshold limit

There are six rule types: duplicate, match, collector, threshold, passthrough, or resetOnMatch. Each rule analyzes incoming events based on predicates you specify; depending on this analysis, each received event is either discarded by the rule or forwarded to the actions specified in the rule. (If no action is specified, the events are forwarded to the gateway.)

**Duplicate**
The first event is forwarded, and all duplicate events are discarded.

**Match**  All the events that match the rule are forwarded.

**Collector**
Events are collected for the time specified in the rule and then all events are sent individually to the server.

**Threshold**
A single event is sent to the server when the threshold is reached.

**Passthrough**
The events are sent to the server only if a specific set of subsequent events arrives during a time window.

**Reset On Match**
The events are sent to the server only if a specific set of subsequent events does not arrive during a time window.

**Note:** An event is analyzed according to the time it arrives at the state correlation engine, not the time it originates. In some circumstances, this can affect the behavior of time-based rules, such as thresholding rules. For example, if the connection between an event source and the gateway is lost, events are cached at the adapter until the connection becomes available again. When the connection is restored, all of the cached events are sent at once, which might trigger a threshold rule unexpectedly.

### Predicates
A predicate in the predicate library consists of a Boolean operator and zero or more arguments. Each argument can be a predicate returning the following types:

<table>
<thead>
<tr>
<th>Predicate Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean value</td>
<td>Equality</td>
</tr>
<tr>
<td>Function returning a value</td>
<td>Addition</td>
</tr>
<tr>
<td>Event attribute</td>
<td>&amp;hostname</td>
</tr>
<tr>
<td>Constant</td>
<td>The string foobar</td>
</tr>
</tbody>
</table>

See [Chapter 9, “State correlation reference”, on page 305](#) for more information.

### Actions
Rules can call actions to process events. Actions are implemented as Java classes, and instances of the required action classes are instantiated during initialization of the state correlation engine. (See “Writing custom actions” on page 296 for more information.) The four standard actions for state correlation are the Discard action, the TECSummary action, the Forward action, and the SendTECEvent action. These actions support a common, optional Boolean attribute, named singleInstance. If this attribute is equal to false, the action is not shared among different rules. Thus,
one instance of the action is created for every rule that triggers it. This is the default behavior. If the attribute is true, a single instance of the action is created and shared among all rules that trigger it.

Note: In addition to the four standard actions, you can also use custom actions that you develop. See “Writing custom actions” on page 296 for more information.

Discard
The Discard action explicitly discards an event when a rule is triggered. Thus, the event is not forwarded. This action has no arguments. The following XML fragment shows an example with the Discard action (see Chapter 9, “State correlation reference”, on page 305 for more information about XML rule syntax):

    <rule id="root.match_discard_tec_notice">
        <eventType>TEC_Notice</eventType>
        <match>
            <predicate>
                <![CDATA[
                    # always succeeds
                    true
                ]]>  
            </predicate>
        </match>
        <triggerActions>
            <action function="Discard" singleInstance="true"/>
        </triggerActions>
    </rule>

TECSummary
The TECSummary action processes a set of matching events and summarizes them in the form of a single summary event, which can then be forwarded to the event server. The summary event is the last event sent to the action by the rule, with the repeat_count attribute incremented to indicate the total number of events being summarized. (If any of the received events already have repeat_count attributes, these values are preserved by adding them to the final repeat_count value.)

In addition to setting the repeat_count attribute, the TECSummary action can also optionally set the value of the msg attribute of the summary event. The value used is determined by the optional msg parameter specified by the XML rule.

For example, suppose the following events are received:

EVENT;repeat_count=3;msg=event1;
EVENT;repeat_count=5;msg=event2;
EVENT;msg=event3;

The generated summary has the following repeat_count:

repeat_count = 3 + 5 + 1 = 9

Only the repeat_count and msg attributes of the summary event are changed by the TECSummary action. All other attribute values are preserved as they were received by the state correlation engine.

The following XML fragment shows how to configure the TECSummary action:

    <rule id="root.duplicate_tec_db">
        <eventType>TEC_DB</eventType>
        <duplicate timeInterval="10000">
            <cloneable attributeSet="sql_code"/>
            <predicate>
                <![CDATA[
                    # always succeeds
                    true
                ]]>  
            </predicate>
        </duplicate>
        <triggerActions>
            <action function="Discard" singleInstance="true"/>
        </triggerActions>
    </rule>
Forward

The Forward action is used to forward an event to another rule. By using the Forward action, you can link rules together so that a single event is processed serially by multiple rules. This can be useful if you want to filter or summarize events with one rule before analyzing them with another, or if you want to use one rule to summarize the events generated by several other rules. The parameters of the Forward action specify the destination the event is to be forwarded to. For example, the following code forwards an event to a rule called rule2:

```xml
<action function="Forward" >
  <parameters>
    <![CDATA[
      RULES:rule2
    ]]>
  </parameters>
</action>
```

You can also use the Forward action to forward a single event to more than one rule. To do this, specify multiple rules, separated by commas:

```xml
<action function="Forward" >
  <parameters>
    <![CDATA[
      RULES:rule2, rule3, rule4
    ]]>
  </parameters>
</action>
```

If the destination action is meant to receive events only through the Forward action, you can indicate this by specifying the attribute `directAccess="false"` on the RULE element. If you specify this attribute, a rule cannot receive events directly from the state correlation engine; instead, only events forwarded by other rules can be received.

In addition to forwarding the event to the specified destination, the Forward action also forwards it to the next action in the current rule, if there is one.

**Note:** If you use the Forward action to link rules together, make sure the sequence of rules does not cause an infinite loop.

The following XML example shows how the Forward action might be used to link multiple rules. In this example, a collector rule collects the events forwarded by two different threshold rules.

```xml
<?xml version="1.0"?>
<!DOCTYPE rules SYSTEM "rule.dtd">
<rules predicateLib="ZCE">

# If we reach this point then
# the sql_code is already duplicated
# because it is used as a cloneable
# parameter.
true

</rule>
```
The SendTECEvent action is used to finalize the processing of events that have been modified by custom actions before sending them back to the gateway. (See "Updating events" on page 300 for more information.)

The following XML rule uses a custom action, EventModificationSample, to escalate the severity of any incoming TEC_Notice event. It then uses the SendTECEvent action to update the modified event and forward it to the gateway.
### Attributes common to all rules

The following are attributes common to all rules:

**id**
Specifies the identifier for each rule. It must be unique within the correlation engine where it is registered. Periods are treated as directories. For example, if you have a rule with the identifier test.threshold, you cannot have another rule with the identifier test.threshold.1.

**eventType**
Specifies an event class to which this rule applies. A rule can include multiple eventType elements, each one specifying a different event class; the rule will process events of any matching class. If a rule includes no eventType elements, it processes events of any type. This can affect performance, so whenever possible, use eventType elements to limit rule processing to relevant event classes.

*Note:* The state correlation engine does not have access to the BAROC files defining event classes and therefore cannot analyze events according to inheritance hierarchies. You must include a separate eventType element for each specific event class you want to process, regardless of inheritance relationship. Subclasses of specified classes are not automatically processed.

### Matching rules

Matching rules are stateless. They perform passive filtering on the attribute values of an incoming event. A matching rule consists of a single predicate; if the predicate evaluates to true, the trigger actions, which are specified in the rule, run. The following is an example of the rule:

```xml
<!-- Discard all heartbeat events for my host that have msg="please match me". -->
<rule id="test.match">
    <eventType>TEC_Heartbeat</eventType>
    <match>
        <predicate>
            <![CDATA[
                &msg == "please match me" &&
                &hostname == "hostname1"
            ]]>         
        </predicate>
    </match>
</rule>
```
Duplicates rules
The duplicates rule blocks the forwarding of duplicate events within a time interval. It requires these arguments:

- A time interval during which state correlation blocks duplicates of the trigger event. You control the interval with the `timeInterval` attribute, specified in milliseconds. The trigger event is the first event detected by the duplicates rule and is the only one that is not actually discarded.
- A predicate that is used in detecting the trigger event.

Figure 1 shows the state transitions for the duplicate rule:

```
S1
^ 1
| 2
v
S2
```

Threshold rules
The threshold rule looks for \( n \) occurrences of an event within a time interval. When the threshold is reached, it sends events to the defined actions. The threshold rule requires the following parameters:

- One of the sending modes specified by the `triggerMode` attribute:
  - `firstEvent` Sends the first event received during the time window.
  - `lastEvent` Sends the last \((n^{th})\) event received during the time window.
**allEvents**
Sends all events 1 through \( n \), This is the default mode.

**forwardEvents**
Sends all events after the \( n \)th until it resets.

- A time interval during which the threshold has to be reached. You control the interval with the timeInterval attribute, specified in milliseconds.
- The time interval mode that indicates whether the time interval is fixed or sliding. The attribute timeIntervalMode=fixedWindow | slideWindow interval controls the mode. The default value is fixedWindow.
- The number of events to match, specified by the thresholdCount attribute.
- A trigger predicate that is used in detecting 1 through \( n \) events.

Figure 2 and Figure 3 show the operation of the threshold rule with timeIntervalMode=fixedWindow specified.

**Figure 2. State transitions for the basic threshold rule**

**Figure 2** shows the state machine for the modes firstEvent, lastEvent, and allEvents. Transition 1 occurs when state correlation detects the trigger event (trigger predicate matches). Transition 2 takes place when a subsequent incoming event matches the predicate. When the time interval expires, transition 3 occurs and the state machine resets. Transition 4 resets the state machine after the threshold is reached. When the state SN is reached, either the first event, the last event, or all \( n \) events are sent before resetting.

**Figure 3. State transitions for the threshold rule using forwardEvents**
In forwardEvents mode (Figure 3 on page 290), the threshold rule sends all events matching the predicate after the threshold is reached, until the time interval expires.

If you specify a fixed window time interval (timeIntervalMode=fixedWindow), the timer starts when the first matching event arrives and closes after the time interval expires. In order for the threshold to be triggered, \( n \) matching events must be received within that time window. If the threshold is not reached within the specified time window, the rule resets, and the next subsequent matching event starts a new time window.

If you specify a sliding window time interval (timeIntervalMode=slideWindow), a separate timer is effectively started when each matching event arrives, resulting in overlapping virtual time windows. In this case, the threshold is triggered if \( n \) matching events arrive within any time window.

The difference between a fixed window and a sliding window for a time interval of \( t \) seconds might be summarized as follows:

- With a fixed window, the threshold is triggered if \( n \) events arrive within \( t \) seconds of the first event.
- With a sliding window, the threshold is triggered if \( n \) events arrive within \( t \) seconds of each other.

The following is an example of a threshold rule with a sliding time window:

```xml
<rule id="test.threshold">
  <eventType>Node_Down</eventType>
  <threshold thresholdCount="5" timeInterval="60000" timeIntervalMode="slideWindow" triggerMode="allEvents">
    <predicate>
      <![CDATA[
        (&msg == "node down") &&
        (isMemberOf(&hostname, [ 192.168./16 ]))
      ]]> 
    </predicate>
  </threshold>
</rule>
```

Threshold rules can also define complex aggregate values, instead of a simple count of events. Use the aggregate configuration tag to define this rule. You can construct an aggregate value similar to the definition of a predicate. But instead of a simple true or false result, define a progressive value using the functions listed in Chapter 9, “State correlation reference”, on page 305. Threshold rules with aggregate values trigger only when the aggregate value is equal or greater than the thresholdCount value. The following example sets a thresholdCount value based on a percentage, but works with an attribute value that is specified as a fractional value between 0 and 1. By using an aggregate, the rule converts the fractional value into a percentage:

```xml
<rule id="test.aggregate_threshold">
  <eventType>Temperature_Variation</eventType>
  <threshold thresholdCount="100" timeInterval="2000" triggerMode="allEvents" timeIntervalMode="fixedWindow">
    <predicate>
      <![CDATA[
        (&value < 0.5) &&
        (isMemberOf(&host, [ 192.168./16 ]))
      ]]> 
    </predicate>
  </threshold>
</rule>
```
Collector rules

The collector rule gathers events that match the given predicate for a specified period of time. The rule triggers when the timer expires and sends all collected events to the defined actions. The collector rule requires these arguments:

- A time interval during which matching events are collected. You control the interval with the timeInterval attribute, specified in milliseconds.
- A predicate, which is part of filtering the relevant events to add to the collection.

Figure 4 shows the state transitions for the collector rule:

![State transitions for the collector rule]

In Figure 4, S1 is the initial state. Transition 1 occurs when there is a match on an incoming event; the initial event is not sent but collected. A timer is set to the specified interval. Before the timer expires, all incoming events that match the predicate are collected (transition 2). Transition 3 occurs when the time interval expires, and the state machine resets. At this time, all collected events are sent. The following is an example of the rule:

```xml
<!-- Collects 10 seconds of Server_Down events for my database. -->
<rule id="test.collector">
  <eventType>Server_Down</eventType>
  <collector timeInterval="10000">
    <predicate>
      <![CDATA[
        $servername == "my_database"
      ]]>
    </predicate>
  </collector>
</rule>
```

Passthrough rules

The passthrough rule forwards the trigger event only if a specific set of events arrives within a specified time interval. If the required events arrive before the timer expires (optionally is a specific sequence), the trigger event is forwarded; if they do not arrive, the timer resets and the trigger event is not forwarded.

The passthrough rule requires the following arguments:
- A Boolean value (randomOrder) indicating whether the required events can arrive in any order or must arrive in the order specified. If randomOrder is equal to yes, the events can arrive in any order.
- A time interval, after which the state machine will reset.
- A trigger predicate, defining the trigger event. This is the event that initializes the state machine and is forwarded if the required subsequent events arrive within the time interval.
- One or more predicates specifying the required subsequent events.

Figure 5 shows the state transitions for the passthrough rule when the required events must arrive in sequence (randomOrder=no).

![Figure 5. State transitions for the passthrough rule (randomOrder=no)](image)

In Figure 5, S1 is the initial state. Transition 1 occurs when the trigger event is detected; the transition stores the event and starts the timer. Transition 2 occurs when an incoming event matches the first predicate in the required sequence; similarly, transition 3 takes place when an incoming event matches the second predicate in the sequence. When state S4 is reached, the rule forwards the trigger event and resets to the initial state S1 (transition 5). Transition 4 occurs when the time interval expires, resetting the rule to the initial state without forwarding the trigger event.

Figure 6 on page 294 shows the state transitions for the passthrough rule when the required events can arrive in any order (randomOrder=yes).
In Figure 6, the transitions are the same as in Figure 5 on page 293. In this case, however, the final state is S5, after which the state machine resets.

**Reset on match rules**

The reset on match rule forwards the trigger event only if a specific set of events does not arrive within a specified time interval. If the required events arrive before the timer expires (optionally in a specific sequence), the trigger event is not forwarded; if they do not arrive, the timer resets and the trigger event is forwarded.

The passthrough rule requires the following arguments:
- A Boolean value (randomOrder) indicating whether the required events can arrive in any order or must arrive in the order specified. If randomOrder is equal to yes, the events can arrive in any order.
- A time interval, after which the state machine will reset.
- A trigger predicate, defining the trigger event. This is the event that initializes the state machine and is forwarded if the required subsequent events do not arrive within the time interval.
- One or more predicates specifying the subsequent events required to prevent forwarding of the trigger event.

Figure 7 on page 295 shows the state transitions for the reset on match rule when the required events must arrive in sequence (randomOrder=no).
In Figure 7, S1 is the initial state. Transition 1 occurs when the trigger event is detected; the transition stores the event and starts the timer. Transition 2 occurs when an incoming event matches the first predicate in the required sequence; similarly, transition 3 takes place when an incoming event matches the second predicate in the sequence. When state S4 is reached, the rule resets to the initial state S1 (transition 5). Transition 4 occurs when the time interval expires, causing the rule to forward the trigger event and then reset to the initial state.

Figure 8 shows the state transitions for the reset on match rule when the required events can arrive in any order (randomOrder=yes).

In Figure 8, the transitions are the same as in Figure 7. In this case, however, the final state is S5, after which the state machine resets without forwarding the trigger event.
Cloning state machines

You can clone any state-based rule by using the `clonable` tag. If state correlation clones a rule when the trigger event occurs, state correlation creates another instance of the rule. Such a rule is useful in handling multiple event sequences without the need to write many rules.

For example, if a rule is cloned based on the hostname attribute, a new instance of the rule is created for each different hostname value (assuming all other filter predicates match). This makes it possible to write a single rule that can be applied to multiple machines without having to write a separate rule for each.

The following example uses cloning to detect a simultaneous security attack coming from several sources (same attack, different sources).

```xml
<rule id="test.cloneable_threshold">
  <eventType>Node_Down</eventType>
  <threshold thresholdCount="5" timeInterval="60000" triggerMode="allEvents">
    <clonable attributeSet="hostname"/>
    <predicate>
      <![CDATA[
        (&msg == "node down")
      ]]>
    </predicate>
  </threshold>
</rule>
```

Cloneable attributes are part of the event filters defined for the state machine. Thus, events must have the value for the cloneable attribute defined to trigger the state machine. For example, you define a state machine with the following XML element:

```xml
<cloneable attributeSet="hostname"/>
```

**Note:** A null attribute value is not treated as another value for cloning purposes. In the above example, if an event arrives that does not have the hostname attribute, the rule is not triggered.

Writing custom actions

In addition to the standard actions (Discard, TECSummary, Forward, and SendTECEvent), your rules can also use custom actions you write using Java code. By writing custom actions, you can perform more sophisticated event processing, including modification of event attributes.

Each action is implemented as a Java class. When the state correlation engine starts, it creates instances of all of the action classes required by the rules, using any parameters specified by the rules. If an action is declared as shared, only a single instance of each action is created, and this same instance is used by all rules that call that action. If an action is not shared, a separate instance is created for each rule that uses a particular action class. (If you need to call an action using different parameters in different rules, the action cannot be shared.)
Event objects

The state correlation engine works with Java objects that represent events. When an event arrives at the gateway or is generated by the adapter (depending on where state correlation is running), a Java object is created containing all of the attribute data from the event. This object, an instance of class com.tivoli.zce.engine.Event, is sent to the state correlation engine. If persistence is enabled, the state correlation engine then writes a record of the event to a persistent store on the gateway machine. The persistent store is a recovery mechanism used to ensure that no events are lost if the gateway shuts down while events are processed by the state correlation engine. When the gateway is restarted, any unsent events recorded in the persistent store are immediately sent to the gateway.

Note: The persistent store used by the state correlation engine is separate from event buffer used by the gateway or adapter, although both are configured by the BufferEvents and BufEvtPath keywords in the gateway profile or adapter configuration file. See IBM Tivoli Enterprise Console User's Guide for more information about configuring event buffering.

As initially created, the Java Event object contains two copies of the event:

- A working copy, which can be directly accessed using the methods of the event object. Actions can use this working copy to make changes to event attributes during processing.
- An internal snapshot of the state of the event as it was received by the state correlation engine. This field is initially equivalent to the record written to the persistent store, and it is not dynamically updated to match changes to the working copy.

The event object is then processed by the state correlation rules and any actions called by those rules. This processing might include discarding the event, changing the event attribute values, or creating new events. When the rules finish processing, the event is sent to the gateway (unless the event has been discarded).

The event data sent to the gateway is based on the content of the internal snapshot field at the time rule processing is completed. Because this field is not automatically updated during event processing, any rule that changes the attribute data of an event must update the snapshot before exiting; otherwise, changes made to the event data are lost. The simplest way to do this is to always use SendTECEvent as the last action in your rule; you can also update the snapshot within a custom action. See [Updating events on page 300] for more information.

If persistence is enabled, when the event is forwarded to the gateway, the record of the event in the persistent store is marked to indicate that the event has been sent. This prevents the event from being sent again the next time the gateway is started. If the event is discarded within a custom action, you must also make sure the action removes the event from the persistent store. See [Discarding events on page 301] for more information.

Action structure

An action is implemented as a Java class and must be included in the com.tivoli.zce.actions.libs package in order to be found by the state correlation engine. This class must implement the necessary interfaces to support event handling within state correlation rules. An action class should extend the
com.tivoli.zce.actions.DefaultActionHandler class, which provides the necessary interfaces and some default behavior. An action must implement the following three methods:

- **processEvent()**. This method is called by a rule, or by the preceding action within a rule, and takes as its parameter a single Event object. This method is used in cases where the action is processing a single event.

- **processEvents()**. This method is similar to the **processEvent()** method, but takes as its parameter a single EventList object, which contains an array of multiple events. Both methods must be implemented, because an action can be called with either a single event or an event list. The **processEvents()** method can parse the list and then call the **processEvent()** method for each one.

- **doParse()**. This method is called by the state correlation engine during initialization, after the action class is instantiated. The **doParse()** method parses the parameters specified in the XML rule that calls the action; these parameters govern the behavior of the action instance for all rules that use it. (Note that if an action is not shared, multiple instances might be created with different parameters.)

When an event is received by the action (through either the **processEvent()** method or the **processEvents()** method), the action can call the methods of the event object to retrieve or change the event attribute data. (See “Working with events” on page 299 for more information.) Finally, assuming the event is not discarded, the action must send the event to the next step in processing. This can either be to another action, or if the current action is the last one within a rule, back to the state correlation engine. The basic method for doing this is the **forward()** method, but under some circumstances you might need to use a different method (see “Updating events” on page 300 for more information).

Figure 9 on page 299 summarizes the general structure of an action class.
Working with events

An event object (an instance of com.tivoli.zce.engine.Event) contains the event attribute information as a set of name-value pairs stored in a hash table. The Event class provides methods you can use to work with these attributes. The methods include:

hasAttribute()

The hasAttribute() method takes a single string as a parameter and returns a Boolean value indicating whether the event contains an attribute with the specified name. For example, event.hasAttribute("HOSTNAME") would return true if event has an attribute called HOSTNAME. (Note that attribute names are case sensitive.)

g getString()

The getString() method takes a single string as a parameter and returns a string containing the value of the attribute with the specified name. For example, event.getString("SEVERITY") returns the current value of the SEVERITY attribute of event.

Note: The Event class implements other type-specific methods for retrieving values of other types. However, all Tivoli Enterprise Console event attributes are represented as strings, so only the getString() should be used to retrieve Tivoli Enterprise Console event attribute values.

putItem()

The putItem() method takes as its parameters a string key and a value.
This method sets the value of the attribute whose name is equal to the specified key string. For example, `event.putItem("ORIGIN","SCE")` sets the value of the ORIGIN attribute to the string "SCE". If the specified key does not match an existing attribute, a new attribute is added. If the attribute value contains spaces or special characters, enclose it within nested single quotes to ensure correct parsing by the event server.

**Note:** The state correlation engine does not have access to the BAROC files defining event classes, and it does not attempt to validate the attribute names specified with the `putItem` method. You must ensure that any attributes you add using this method are valid for the event class; an event containing attributes that are not valid will be discarded by the event server. If you are using custom actions, you should check your event server logs periodically to ensure that events are not being discarded for this reason.

In order to facilitate working with event attributes, the `ITecEventAttributes` interface defines constants representing many of the common attribute names used by base event classes. If your action class implements this interface, you can use these constants rather than string literals to specify attribute names.

### Updating events

Rule processing works on copies of events represented as Java objects. When rule processing is finished, the state correlation engine uses the value of the internal snapshot field to update the attribute values of the event before sending it to the gateway. This field contains a snapshot of the event attribute data at a particular time and is not dynamically updated to reflect changes to the attribute hash table. Therefore, it is important to update the snapshot in order to capture the modifications made by custom actions. An event snapshot must be updated in either of the following situations:

- If the event has been modified by any action in the rule, the snapshot must be updated at some point after the last modification and before the rule exits. This can be done either by the same action that modified the event or by a subsequent action.
- If the action created the event, it must also update the snapshot before forwarding the event to the next action.

There are several ways you can handle updating the snapshot field:

- You can use the `TecEventUtil.updateTecEvent()` method to explicitly update the event snapshot before calling `actionHandler.forward()`. This method takes as its parameters two event objects: the event to be updated, and the event that was originally passed in to the action. (These are the same unless the event being updated is a new one.) The following example updates the snapshot of `myEvent` before forwarding it to the next action:
  ```java
  updateTecEvent(myEvent,myEvent);
  forward(myEvent);
  ```

- You can use the `TecEventUtil.forwardTecEvent()` method to update the event snapshot and forward it to the next action in a single step. Using this method instead of the `actionHandler.forward()` method ensures that the snapshot is always current; however, updating the snapshot unnecessarily can affect performance, so use this method only when the snapshot must be updated. If the event has not been modified, or if the action is never used as the last one within a rule, it is safe (and more efficient) to use the `forward()` method instead. The `forwardTecEvent()` method takes three parameters: the action handler doing...
the forwarding, the event being modified, and the original event passed in to the
action. The following example updates the snapshot of myEvent and forwards it
to the next action:

TecEventUtil.forwardTecEvent(this, myEvent, myEvent);

- You can include the standard SendTECEvent action as the last action in the rule.
  This action uses the TecEventUtil.forwardTecEvent() method to update and
  forward any event it receives, ensuring that all changes made by earlier actions
  are reflected in the forwarded event.

Note: SendTECEvent is useful for updating the snapshot only when a
preexisting event has been changed by a custom action. If your action
creates a new event, it must also update the snapshot before forwarding
the event, even if it is not the last action in the rule. If you forward a new
event without updating it first, an error results.

Discarding events

You can discard an event within a custom action by simply not forwarding the
event to the next action. An event that is not forwarded receives no further
processing and is not sent to the event server after the rule exits.

However, if you discard an event, you must also remove it from the persistent
store in order to ensure that it is not unintentionally sent to the event server the
next time the gateway restarts. To remove an event from the persistent store, use
the PHandler.removeEvent_termAction() method, specifying either a single event
or an event list as a parameter.

Action class examples

The examples in this section show several different simple custom action classes.

Event modification example

This example escalates the severity of all incoming events. When an event is
received, either individually or as part of an event list, the action reads the current
value of the SEVERITY attribute. If SEVERITY is equal to WARNING, MINOR, or
CRITICAL, the action increments it to the next highest level. If SEVERITY is equal
to FATAL, it is unchanged.

After changing the value of the SEVERITY attribute, the action then updates the
event snapshot and forwards the event to the next action.

Because this action does not use any parameters, the doParse() method does not
need to do anything, but must still be present.

// Actions must be part of the com.tivoli.zce.actions.libs package
package com.tivoli.zce.actions.libs;

// State correlation imports
import com.tivoli.zce.IRule;
import com.tivoli.zce.ParserException;
import com.tivoli.zce.CorrelatorException;
import com.tivoli.zce.engine.EventList;
import com.tivoli.zce.engine.Event;
import com.tivoli.zce.actions.DefaultActionHandler;
import com.tivoli.zce.actions.TecEventUtil;

public class EventModificationSample extends DefaultActionHandler implements ITecEventAttributes{
    /**
     * Processes a list of events by parsing the list and calling processEvent()
     * for each
     */
*/
public void processEvents(EventList eventList) throws Exception
{
    if (eventList != null)
    {
        try
        {
            for (int i = 0; i < eventList.size(); i++)
            {
                processEvent(eventList.eventAt(i));
            }
        }
        catch (Exception x)
        {
            x.printStackTrace();
        }
    }
}

/******************************************************************************
 * Processes a single event. If the value of the SEVERITY attribute is lower  
 * than FATAL, it is incremented to the next level. The SEVERITY_LIST array is   
 * defined by the ITecEventAttributes interface and contains the six standard  
 * severity values: UNKNOWN, HARMLESS, WARNING, MINOR, CRITICAL, and FATAL.      
 ******************************************************************************/
public void processEvent(Event event) throws Exception
{
    String severity = null;
    if (event != null)
    {
        if (event.hasAttribute(SEVERITY))
        {
            severity = event.getString(SEVERITY);
            int i = 0;
            while (i < SEVERITY_LIST.length && !(severity.equals(SEVERITY_LIST[i])))
            {
                i++;
            }
            if (i < SEVERITY_LIST.length - 1)
            {
                severity = SEVERITY_LIST[i + 1];
            }
        }
        else
        {
            severity = SEVERITY_LIST[0];
        }
    }
    event.putItem(SEVERITY, severity);
    // Update the event snapshot and forward to the next action
    TecEventUtil.forwardTecEvent(this, event, event);
}

/******************************************************************************
 * Parses parameters for the action (none in this case)                         
 ******************************************************************************/
public Object doParse(IRule rule, String paramSz) throws ParserException
{
    return null;
}
}

**Event discard and creation example**
This example discards each incoming event and creates a new event of a different class, using attribute information from the original event. When an event is received, either individually or as part of an event list, the action reads the value
of the HOSTNAME attribute (if the event has one). It then creates a new TEC_Notice event and sets the HOSTNAME attribute of the new event to the same value as that of the original event (or to a default value). The new event is then updated and sent to the next action.

The action then records the new event in the persistent store, and removes the original event from the persistent store. Because the original event is not forwarded to the next action, it receives no further processing and is discarded.

```java
package com.tivoli.zce.actions.libs;

// State correlation imports
import com.tivoli.zce.IRule;
import com.tivoli.zce.ParserException;
import com.tivoli.zce.CorrelatorException;
import com.tivoli.zce.engine.EventList;
import com.tivoli.zce.engine.Event;
import com.tivoli.zce.actions.DefaultActionHandler;
import com.tivoli.zce.persistence.PHandler;
import com.tivoli.zce.actions.TecEventUtil;

public class CreateNewEventSample extends DefaultActionHandler implements ITecEventAttributes {
    // A default host name to use if the event does not contain one
    private final String DEFAULT_HOSTNAME = "'no host'";

    public void processEvents(EventList eventList) throws Exception {
        if (eventList != null) {
            try {
                for (int i = 0; i < eventList.size(); i++) {
                    processEvent(eventList.eventAt(i));
                }
            } catch (Exception x) {
                x.printStackTrace();
            }
        }
    }

    public void processEvent(Event event) throws Exception {
        Event newEvent = new Event();
        String hostname = DEFAULT_HOSTNAME;
        if (event != null) {
            if (event.hasAttribute(HOSTNAME)) {
                hostname = event.getString(HOSTNAME);
            }
        }
    }
}
```
// create new event and set HOSTNAME attribute equal to HOSTNAME of
// original event, or to default value
newEvent.type = "TEC_Notify";
newEvent.putItem( HOSTNAME, hostname);
newEvent.putItem( MSG, "Generated by State Correlation");
newEvent.putItem( ORIGIN, "SCE");

// Update the snapshot and forward the new event to the next action
TecEventUtil.forwardTecEvent( this, newEvent, event);

// Add the new event to the persistent store
PHandler.addEvent( newEvent);

// Remove the original event from the persistent store. Because it
// not forwarded, the event is discarded.
PHandler.removeEvent_termAction(event);
}
Chapter 9. State correlation reference

This appendix contains the following topics:

- Rule syntax
- Predicate library
- State correlation messages

Note: Rules in state correlation are not the same as IBM Tivoli Enterprise Console rules.
For general information about state correlation, see Chapter 8, “Writing rules for the state correlation engine”, on page 283.

Rule syntax

State correlation rules are defined using XML syntax version 1.0. Rule definitions must be compliant with the document type definition (DTD) provided in the $BINDIR/TME/TEC/default_sm/tecsce.dtd file.

The XML default encoding is UTF-8. If you want to specify characters in encodings other than UTF-8, you must follow the XML encoding rules. To do this, use the XML encoding keyword as shown in the following example:

```xml
<?xml version="1.0" encoding="EUC-JP"?>
```

Event attributes with string values can be set to a different encoding, but state correlation only occurs if the event to be correlated also follows the same encoding.

Note: Line numbering is for explanation purposes only.

Basics of a rule

A rule file consists of the following:

- Specification of predicate libraries in the included rules
- Variables that are available to predicates and actions within a defined scope (coverage) of rules
- Rules containing predicates and actions
- Declaration of the required CDATA tag for predicates

The following is an example of a rule. See Table 3 on page 306 for an explanation of the rule elements.

```xml
1  <?xml version="1.0"?>
2  <!DOCTYPE rules SYSTEM "tecsce.dtd">
3  <!-- Comments -->
4  <rules predicateLib="ZCE">
5  <predicateLib
6   name="ZCE"
7   class="com.tivoli.zce.predicates.zce.parser.ZCEPredicateBuilder">
8  <parameter>
9   <field>defaultType</field>
10  <value>Strings</value>
11  </parameter>
```

© Copyright IBM Corp. 2003
Table 3. Explanation of a basic rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–2</td>
<td>Define the version of XML as well as the DTD used by the rule definitions.</td>
</tr>
<tr>
<td>4</td>
<td>Declares the rules: this is the root element of this XML document. A rule definition uses predicates to filter events. A predicate library (typically a set of boolean predicates and arithmetic operators) needs to be declared. In this example, all rules use the predicate library called ZCE by using the predicateLib element.</td>
</tr>
<tr>
<td>5–12</td>
<td>Declare the specifics of the ZCE library.</td>
</tr>
<tr>
<td>6</td>
<td>Declares the name of tag to be used everywhere in the XML file.</td>
</tr>
<tr>
<td>7</td>
<td>Declares which class needs to be instantiated. This class needs to implement this interface: com.tivoli.zce.predicate.IPredicateBuilder</td>
</tr>
<tr>
<td>9–10</td>
<td>Provide optional parameters for the initialization of the predicate: these parameters are predicate library specific. Within the ZCE library, the defaultType parameter is set to the value String: each new label seen in a predicate is typed as String by default.</td>
</tr>
<tr>
<td>13–15</td>
<td>Declare variables available to all predicates and actions within the defined scope test.</td>
</tr>
<tr>
<td>16</td>
<td>Specifies the rule identifier.</td>
</tr>
<tr>
<td>17</td>
<td>Specifies the event classes applicable to this rule.</td>
</tr>
<tr>
<td>18</td>
<td>Specifies the rule type.</td>
</tr>
<tr>
<td>19–23</td>
<td>Specify the predicate syntax for event filtering.</td>
</tr>
<tr>
<td>25</td>
<td>Specifies the action that occurs when the rule triggers.</td>
</tr>
</tbody>
</table>

Match rule example
The following is a partial example of a match rule. See Table 4 on page 307 for an explanation of the rule.

```xml
<rule id="rules.one" >
  <match>
    <predicate>
      true
    </predicate>
  </match>
</rule>
```
Table 4. Explanation of a match rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declares a simple rule, called rules.one. The naming convention for rule names follows the dotted OID (object identifier) notation, as in SNMP, where a dot denotes a change in the hierarchy level. This naming convention allows simplified management of rules. The naming convention makes it possible to apply a management operation, such as delete, to a group of rules. In this case, the naming convention appears as follows: rules.* Also, you cannot have a rule ID as a subset of another ID. For example, you cannot have one rule that is named rule.one and another is named rule.one.a.</td>
</tr>
<tr>
<td>2</td>
<td>Declares the type of the rule: here it is a match rule.</td>
</tr>
<tr>
<td>4</td>
<td>Declares the one predicate that a match rule needs. It actually equals to true: this rule triggers for all events.</td>
</tr>
<tr>
<td>7</td>
<td>Provides an optional description of what happens when a rule matches. In this example, the TECSummary action is called without any parameters. This simple action delivers the matched event to Tivoli Event Integration Facility for further forwarding.</td>
</tr>
</tbody>
</table>

Collector rule example

In this example of the collector rule, the header definitions are omitted for simplicity. See Table 5 on page 308 for an explanation of the rule.

```
<rule id="rules.2">
  <eventType>PIX_TCP_in_conn_denied</eventType>
  <collector timeInterval="10000">
    <cloneable attributeSet="pix_sev pix_code pix_ifname" />
    <predicate>true</predicate>
  </collector>
  <action function="TECSummary">
    <parameters><![CDATA[
      SET:msg=SUMMARY_Multiple_TCP/IP_Inbound_connections_denied
    ]]>}
  </action>
</rule>
```

This rule applies only to specified classes of events. This is enforced by including one or more eventType elements, each specifying an event class for the rule. In the example, the rule processes only events of class PIX_TCP_in_conn_denied. The eventType element is a mandatory parameter of internal event representation. It is possible to add a set of event classes as follows:
Table 5. Explanation of a collector rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Specifies the rule as a collector type rule. This type of rule collects events during a specified amount of time and forwards all received events. In the example, the rule triggers after 10 000 milliseconds, as specified by the timeInterval attribute.</td>
</tr>
<tr>
<td>4-5</td>
<td>Define this rule to be cloneable. The rule clones for each different value of pix_sev, pix_code, pix_ifname. There are many cases where event patterns are similar and occur in parallel. An example is a generic rule written to detect a series of logon attacks. These attacks are detected by examining events to determine logon attempts originating from the same source address to different hosts within a predefined amount of time. A single, non-cloneable rule is able to detect a single attack at a time unless the rule is duplicated as many times as there are hosts. Whereas with cloning, a single copy of the rule can detect several simultaneous attacks.</td>
</tr>
<tr>
<td>7</td>
<td>Defines a predicate. In this case, the predicate is redundant as it always evaluates to true. The attributeSet property of the cloneable element enables matching and filtering of events.</td>
</tr>
<tr>
<td>9-15</td>
<td>Declare an action TECSummary and declare arguments for the action. This action sends one summary event from a collection of events. This action accepts a SET parameter for the msg field only. You can define custom msg values for your summary events.</td>
</tr>
<tr>
<td>11-13</td>
<td>SET:msg=value, if present, adds or replaces the value of the msg attribute in the outgoing event with the specified value.</td>
</tr>
</tbody>
</table>

Duplicate rule example

The following code is an example of a duplicate rule. See Table 6 for an explanation of the rule

```xml
<rule id="test.duplicate">
  <eventType>TEC_Error</eventType>
  <duplicate timeInterval="10000">
    <predicate><![CDATA[
      &msg == "internal error on my adapter" &&
      &hostname == "hostname1" &&
      &errno = 10
    ]]>"
  </predicate>
</rule>
```

Table 6. Explanation of a match rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declares a simple rule called test.duplicate.</td>
</tr>
<tr>
<td>2</td>
<td>Specifies an event type handled by the rule (TEC_Error). In this case, only one event type is handled, but a rule can contain any number of eventType elements. If this element is omitted, the rule processes all events.</td>
</tr>
</tbody>
</table>
Table 6. Explanation of a match rule (continued)

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Specifies a duplicate rule, with a time interval of 10 000 milliseconds (10 seconds). Duplicates of the trigger event received during that interval are discarded.</td>
</tr>
<tr>
<td>4</td>
<td>Specifies the predicates used to detect the trigger event.</td>
</tr>
</tbody>
</table>

Threshold rule example

The following code is an example of a threshold rule with a sliding time window. See Table 7 for an explanation of the rule.

```xml
<rule id="test.threshold">
  <eventType>Node_Down</eventType>
  <threshold thresholdCount="5" timeInterval="60000" timeIntervalMode="slideWindow" triggerMode="allEvents">
    <predicate>
      <![CDATA[
        (&msg == "node down") &&
        (isMemberOf(&hostname, [ 192.168./16 ]))
      ]]>  
    </predicate>
  </threshold>
</rule>
```

Table 7. Explanation of a match rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declares a simple rule called test.threshold.</td>
</tr>
<tr>
<td>2</td>
<td>Specifies an event type handled by the rule (Node_Down). In this case, only one event type is handled, but a rule can contain any number of eventType elements. If this element is omitted, the rule processes all events.</td>
</tr>
<tr>
<td>3–4</td>
<td>Specifies a threshold rule, with the following parameters:</td>
</tr>
<tr>
<td></td>
<td>• A threshold count of 5. This number is the number of matching events that must be received in order to exceed the threshold.</td>
</tr>
<tr>
<td></td>
<td>• A time interval of 60 000 milliseconds (one minute). The threshold count must be satisfied within this interval in order for the threshold to be exceeded.</td>
</tr>
<tr>
<td></td>
<td>• A time interval mode of slideWindow. With a sliding time window, the threshold is exceeded if five events are received within any one-minute period.</td>
</tr>
<tr>
<td></td>
<td>• A trigger mode of allEvents. This mode means that if the threshold is exceeded, all of the matching events are forwarded.</td>
</tr>
<tr>
<td>5</td>
<td>Specifies the predicates used to detect matching events.</td>
</tr>
</tbody>
</table>

Passthrough rule example

The following code is an example of a passthrough rule with randomOrder=true. See Table 8 on page 310 for an explanation of the rule.

```xml
<rule id="detect_sequence.passThrough">
  <eventType>serverStatus</eventType>
  <passthrough timeInterval="600000" randomOrder="true">
    <cloneable ignoreMissingAttributes="false">
      <attributeSet="servername" />
    </cloneable>
    <predicate>
      <![CDATA[
        &serverType == "webserver" &&
        &serverOperational == "Off"
      ]]>  
    </predicate>
    <predicate>
      <![CDATA[
        &serverType == "database" &&
        &serverOperational == "On"
      ]]>  
    </predicate>
  </passthrough>
</rule>
```
13     &serverOperational == "Off" ]]>
14 </predicate>
15 </passthrough>
16 <action function="TECSummary" >
17 <parameters>
18 <![CDATA[
20 ]]>>
21 </parameters>
22 </action>
23 </rule>

Table 8. Explanation of a match rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declares a simple rule called detect_sequence.passThrough.</td>
</tr>
<tr>
<td>2</td>
<td>Specifies an event type handled by the rule (serverStatus). In this case, only one event type is handled, but a rule can contain any number of eventType elements. If this element is omitted, the rule processes all events.</td>
</tr>
</tbody>
</table>
| 3–4   | Specifies a passthrough rule with the following parameters:  
|       | • A time interval of 60,000 milliseconds (one minute). The predicates in the rule must be satisfied within this time period after the arrival of the trigger event.  
|       | • A random-order mode of true. This indicates that the predicates on lines 7 through 14 can be satisfied in any order, but both must be satisfied in order for the events to be forwarded. |
| 5–6   | Specifies that the rule is cloned on the serverName attribute. This means that a copy of the rule is created for each different value of the serverName attribute, not including a null value. |
| 7–10  | Specifies the first predicate that must be satisfied in order for the event to be forwarded (serverType equal to "webserver" and serverOperational equal to "Off"). |
| 11–14 | Specifies the second predicate that must be satisfied in order for the event to be forwarded (serverType equal to "database" and serverOperational equal to "Off"). |
| 16–22 | Specifies the action to which the events are forwarded if the predicates in the rule are satisfied. In this case, the TECSummary action is used to set the msg and repeat_count attributes of the last-received serverType event before forwarding it. The first event is discarded by this action. |

Reset on match rule example

The following code is an example of a passthrough rule with randomOrder=false. See Table 9 on page 311 for an explanation of the rule.

1 <rule id="detect_sequence.resetOnMatch" >
2 <eventType>serverStatus</eventType>
3 <resetOnMatch timeInterval="600000" randomOrder="false">
4 <cloneable ignoreMissingAttributes="false" attributeSet="serverName" />  
5 <predicate>
6 <![CDATA[
7     &serverType == "webserver" && 
8     &serverOperational == "Off" ]]>>
9 </predicate>
10 <predicate>
11 <![CDATA[
12     &serverType == "webserver" &&
13     &serverOperational == "On" ]]>>
14 </predicate>
15 </resetOnMatch>
16 </rule>
Table 9. Explanation of a match rule

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declares a simple rule called detect_sequence.resetOnMatch.</td>
</tr>
<tr>
<td>2</td>
<td>Specifies an event type handled by the rule (serverStatus). In this case, only one event type is handled, but a rule can contain any number of eventType elements. If this element is omitted, the rule processes all events.</td>
</tr>
</tbody>
</table>
| 3     | Specifies a passthrough rule with the following parameters:  
  - A time interval of 60 000 milliseconds (one minute). The predicates in the rule must be satisfied within this time period after the arrival of the trigger event.  
  - A random-order mode of false. This indicates that the predicates on lines 5 through 12 must be satisfied in the order in which they are specified in the rule in order for the events to be discarded. |
| 4     | Specifies that the rule is cloned on the serverName attribute. This means that a copy of the rule is created for each different value of the serverName attribute, not including a null value. |
| 5–8   | Specifies the first predicate that must be satisfied in order for the event to be forwarded (serverType equal to "webserver" and serverOperational equal to "Off"). |
| 9–12  | Specifies the second predicate that must be satisfied in order for the event to be forwarded (serverType equal to "webserver" and serverOperational equal to "On"). |

Declaration of variables

Similar to global definitions in operating systems, you define variables for predicates and actions. A variable is a reference to a constant or a calculated entity. Variables need a predicate or action library for parsing, construction, and use.

Each variable is typed using five generic types: String, Float, Int, Boolean, and Set. The predicate plug-in or the action library needs to parse the value of variables and translate it from the generic types to a predicate or action-specific, internal types.

The following example shows the declaration and usage of variables:

```xml
<?xml version="1.0"?>
<!DOCTYPE rules SYSTEM "tecsce.dtd">
<!-- Comments -->
<rules predicateLib="ZCE">
  <predicateLib name="ZCE"
    class="com.tivoli.zce.predicates.zce.parser.ZCEPredicateBuilder">
    <parameter>
      <field>defaultType</field>
      <value>String</value>
    </parameter>
  </predicateLib>
  <variables scope="test">
    <variable name="IPS" type="Set" value="[1.2.3.4 1.2.3.5]" />
    <variable name="INTS" type="Set" value="[1-5]" />
    <variable name="TRUE" type="Boolean" value="true" />
  </variables>
</rules>
```
The declaration of variables is made within a scope of rules. This example defines variables to be accessible for all rules under the scope test. It is possible to overload a value of an already declared variable by re-declaring the same variable with a more precise scope. A scope cannot be the name of a rule itself, but a collection of rules.

**Table 10. Declaration of variables explained**

<table>
<thead>
<tr>
<th>Lines</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Defines the scope of the variables.</td>
</tr>
<tr>
<td>2</td>
<td>Declares the variable IPS as a Set type with the value [1.2.3.4 1.2.3.5]. IPS is a set of IP addresses.</td>
</tr>
<tr>
<td>3–4</td>
<td>Declare a variable PRED1 as a boolean of value (&amp;a1=&quot;1&quot; &amp;&amp; true). The XML standard does not accept special characters inside an attribute value. Instead, use a variable definition such as PRED1 to overcome this restriction.</td>
</tr>
<tr>
<td>5</td>
<td>The predicate of the test.match rule uses the variable PRED1, which is resolved to the variable defined within the scope of test.</td>
</tr>
</tbody>
</table>

**Predicate library**

This section provides an overview of the predicate library. For the syntax of the predicates and functions, see page 313.

**Value types**

The predicate library has the following value types:

**Table 11. Value types for the predicate library**

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>2, 444, -44</td>
<td>Long</td>
</tr>
<tr>
<td>Float</td>
<td>3.4, 5.122</td>
<td>Float</td>
</tr>
<tr>
<td>String</td>
<td>&quot;somestring&quot;, &quot;John Smith&quot;</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
Table 11. Value types for the predicate library (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>true, false</td>
<td>boolean</td>
</tr>
<tr>
<td>StringSet</td>
<td>[ &quot;apples&quot;, &quot;oranges&quot; ], [ 1 2 3 ]</td>
<td>com.tivoli.zce.engine.StringSet. This class maintains a sorted list of set members as strings for efficiency.</td>
</tr>
<tr>
<td>IntSet</td>
<td>[ 1, 3-6, 99 ]</td>
<td>com.tivoli.zce.predicates.zce.IntSet</td>
</tr>
<tr>
<td>IPAddressSet</td>
<td>[ 10.1.2.3, 9.4./16, 128.89.9.1 ]</td>
<td>com.tivoli.zce.predicates.zce.IPAddressSet</td>
</tr>
<tr>
<td>IPAddress</td>
<td>10.1.2.3, 9.4./16</td>
<td>com.tivoli.zce.predicates.zce.IPAddress. Subnet representation requires a final period (.) For example, 9.4./16 distinguishes it from a float constant.</td>
</tr>
</tbody>
</table>

Range values are valid in IntSet and IPAddressSet types. With the IntSet value type, you define range with the hyphen (-). For example, 4-8 means all numbers from 4 to 8. With the IPAddressSet value type, you define range with submasks. For example, 9.4.3.2 is a straightforward IP address; 9.4./16 denotes all IP addresses 9.4.0.0 - 9.4.255.255.

Attributes

An incoming event, as viewed by state correlation, consists of a set of attribute=value pairs. The name of an attribute is prefixed by an ampersand (&) to distinguish it from function names. Hence, the attribute hostname is represented as &hostname.

In addition, an attribute name can be postfixed with a construct that allows reference to a previously matched rule in a state machine rule. Examples are the following:

&hostname
  Attribute in the current event

&hostname@0
  Attribute in the current event

&hostname@1
  Attribute in the first event, that is the trigger event

&hostname@-3
  Attribute in the third event from the current

Predicates and functions for state correlation

In this appendix, the predicates and functions for state correlation are grouped according to their argument types:

- Integer
- Float
- String and string set
- IP Address
- Boolean

The grammar for state correlation rules supports both infix and prefix syntax. Additionally, you can mix the infix and prefix syntax. For the infix notation, the operator is used between the arguments:
The prefix notation applies to the same operators when they are processed as functions, for example:

\[(\text{int.eq } \&\text{arg1 } \&\text{arg2})\]

In this example, the prefix \text{int.eq} operator is equivalent to the == infix form.

**Integer predicates**

The following integer predicates are described in the next sections:

- `==`
- `!=`
- `>`
- `<`
- `>=`
- `<=`

**==**

Mathematical equal to. Returns true if the arguments are equal.

**Synopsis:** Infix

`==`

Prefix

\text{int.eq}

**Arguments:** integer, integer

**Examples:** Infix

\((\&\text{integerSlot }== 4)\)

Prefix

\((\text{int.eq } \&\text{integerSlot } 4)\)

**!=**

Mathematical not equal to. Returns true if the arguments are not equal.

**Synopsis:** Infix

`!=`

Prefix

\text{int.ne}

**Arguments:** integer, integer

**Examples:** Infix

\((\&\text{integerSlot }!= 4)\)

Prefix

\((\text{int.ne } \&\text{integerSlot } 4)\)

**>**

Mathematical greater than. Returns true if the first argument is greater than the second.
Synopsis: Infix
>
Prefix
int.gt

Arguments: integer, integer

Examples: Infix
(&integerSlot > 4)

Prefix
(int.gt &integerSlot 4)

<
Mathematical less than. Returns true if the first argument is less than the second.

Synopsis: Infix
<

Prefix
int.lt

Arguments: integer, integer

Examples: Infix
(&integerSlot < 4)

Prefix
(int.lt &integerSlot 4)

>=
Mathematical greater than or equal to. Returns true if the first argument is greater
than or equal to the second.

Synopsis: Infix
>=

Prefix
int.ge

Arguments: integer, integer

Examples: Infix
(&integerSlot >= 4)

Prefix
(int.ge &integerSlot 4)

<=
Mathematical less than or equal to. Returns true if the first argument is less than or
equal to the second.

Synopsis: Infix
<=
Prefix
int.le

**Arguments:** integer, integer

**Examples:** Infix
(&integerSlot <= 4)

Prefix
(int.le &integerSlot 4)

**Integer functions**

The following integer functions are described in the next sections:

* +
* –
* *
* /
* absolute
* max
* min
* mod
* negate
* numOfEvents
* range
* timeNow
* timeSpan

+  
Mathematical addition. Returns the integer result of adding arguments.

**Synopsis:** Infix
+

Prefix
int.add

**Arguments:** integer, integer

**Examples:** Infix
(&integerSlot + 4)

Prefix
(int.add &integerSlot 4)

-  
Mathematical subtraction. Returns the integer result of subtracting the second argument from the first.

**Synopsis:** Infix
-

Prefix
int.sub

**Arguments:** integer, integer

**Examples:** Infix
`(&integerSlot - 4)`

Prefix
`(int.sub &integerSlot 4)`

* Mathematical multiplication. Returns the integer result of multiplying arguments.

**Synopsis:** Infix
`
`

Prefix
`int.mul`

**Arguments:** integer, integer

**Examples:** Infix
`(&integerSlot * 4)`

Prefix
`(int.mul &integerSlot 4)`

/ Mathematical division. Returns the integer result of dividing the first argument by the second.

**Synopsis:** Infix
`
`

Prefix
`int.div`

**Arguments:** integer, integer

**Examples:** Infix
`(&integerSlot / 4)`

Prefix
`(int.div &integerSlot 4)`

**absolute**
Returns the absolute value of the integer argument.

**Synopsis:** Infix
`abs(..)`

Prefix
`int.abs`

**Arguments:** integer
**Examples:** Infix
abs(&integerSlot)

Prefix
(int.abs &integerSlot)

**max**  
Returns the maximum from the set of integer arguments. Takes two or more arguments.

**Synopsis:** Infix
max(..)

Prefix
int.max

**Arguments:** integer*

**Examples:** Infix
max(&integerSlot, 4)

Prefix
(int.max &integerSlot 4)

**min**  
Returns the minimum from the set of integer arguments. Takes two or more arguments.

**Synopsis:** Infix
min(..)

Prefix
int.min

**Arguments:** integer*

**Examples:** Infix
min(&integerSlot, 4)

Prefix
(int.min &integerSlot 4)

**mod**  
Returns the integer remainder of the first argument divided by the second.

**Synopsis:** Infix
%

Prefix
int.mod

**Arguments:** integer, integer

**Examples:** Infix
(&integerSlot % 4)
Prefix
(int.mod &integerSlot 4)

**negate**
Returns the negation of the integer argument.

**Synopsis:** Infix
neg(..)

Prefix
int.neg

**Arguments:** integer

**Examples:** Infix
neg(&integerSlot)

Prefix
(int.neg &integerSlot)

**numOfEvents**
Number of matched events, used in history-based rules.

**Synopsis:** Infix
numOfEvents()

Prefix
int.numOfEvents

**Arguments:** There are no arguments.

**Examples:** Infix
numOfEvents()

Prefix
(int.numOfEvents)

**range**
Returns the difference between the maximum and minimum valued arguments.
Takes two or more arguments.

**Synopsis:** Infix
range(..)

Prefix
int.range

**Arguments:** integer*

**Examples:** Infix
range(&integerSlot, 4, 100)

Prefix
(int.range &integerSlot 4 100)
**timeNow**
Returns the current time in number of milliseconds since 00:00:00 GMT on January 1, 1970.

**Synopsis:**
Infix
```
timeNow()
```
Prefix
```
int.timeNow
```

**Arguments:** There are no arguments.

**Examples:**
Infix
```
timeNow()
```
Prefix
```
(int.timeNow)
```

**timeSpan**
Returns the milliseconds corresponding to the time specified in hours, minutes, and seconds.

**Synopsis:**
Infix
```
timeSpan(..)
```
Prefix
```
int.timeSpan
```

**Arguments:** integer, integer, integer

**Examples:**
Infix
```
timeSpan(hours, mins, seconds)
```
Prefix
```
(int.timeSpan hours mins seconds)
```

**Float predicates**
The following float predicates are described in the next sections:
- `==`
- `!=`
- `>`
- `>=`
- `<`
- `<=`

`==`
Mathematical equal to. Returns true if the arguments are equal.

**Synopsis:**
Infix
```
==
```
Prefix
```
float.eq
```
Arguments: float, float

Examples: Infix
(&floatSlot == 4.0)

Prefix
(float.eq &floatSlot 4.0)

!=
Mathematical not equal to. Returns true if the arguments are not equal.

Synopsis: Infix
!=

Prefix
float.ne

Arguments: float, float

Examples: Infix
(&floatSlot != 4.0)

Prefix
(float.ne &floatSlot 4.0)

>
Mathematical greater than. Returns true if the first argument is greater than the second.

Synopsis: Infix
>

Prefix
float.gt

Arguments: float, float

Examples: Infix
(&floatSlot > 4.0)

Prefix
(float.gt &floatSlot 4.0)

>=
Mathematical greater than or equal to. Returns true if the first argument is greater than or equal to the second.

Synopsis: Infix
>=

Prefix
float.ge

Arguments: float, float

Examples: Infix
(&floatSlot >= 4.0)

Prefix
(float.ge &floatSlot 4.0)

<
Mathematical less than. Returns true if the first argument is less than the second.

Synopsis: Infix
<

Prefix
float.lt

Arguments: float, float

Examples: Infix
(&floatSlot < 4.0)

Prefix
(float.lt &floatSlot 4.0)

<=
Mathematical less than or equal to. Returns true if the first argument is less than or equal to the second.

Synopsis: Infix
<=

Prefix
float.le

Arguments: float, float

Examples: Infix
(&floatSlot <= 4.0)

Prefix
(float.le &floatSlot 4.0)

**Float functions**
The following float functions are described in the next sections:
* +
* –
* *
* /
* absolute
* max
* min
* negate
* range
+  
Mathematical addition. Returns the float result of adding arguments.

**Synopsis:**  Infix
+  

Prefix  
float.add  

**Arguments:**  float, float  

**Examples:**  Infix  
(&floatSlot + 4.0)  

Prefix  
(float.add &floatSlot 4.0)  

-  
Mathematical subtraction. Returns the float result of subtracting the second argument from the first.

**Synopsis:**  Infix  
-  

Prefix  
float.sub  

**Arguments:**  float, float  

**Examples:**  Infix  
(&floatSlot - 4.0)  

Prefix  
(float.sub &floatSlot 4.0)  

*  
Mathematical multiplication. Returns the float result of multiplying arguments.

**Synopsis:**  Infix  
*  

Prefix  
float.mul  

**Arguments:**  float, float  

**Examples:**  Infix  
(&floatSlot * 4.0)  

Prefix  
(float.mul &floatSlot 4.0)  

/  
Mathematical division. Returns the float result of dividing the first argument by the second.
Synopsis: Infix
* 

Prefix
float.div

Arguments: float, float

Examples: Infix
(&floatSlot / 4.0)

Prefix
(float.div &floatSlot 4.0)

absolute
Returns the absolute value of the float argument.

Synopsis: Infix
abs(../)

Prefix
float.abs

Arguments: float

Examples: Infix
abs(&floatSlot)

Prefix
(float.abs &floatSlot)

max
Returns the maximum from the set of float arguments. Takes two or more arguments.

Synopsis: Infix
max(../)

Prefix
float.max

Arguments: float*

Examples: Infix
max(&floatSlot, 4.0)

Prefix
(float.max &floatSlot 4.0)

min
Returns the minimum from the set of float arguments. Takes two or more arguments.

Synopsis: Infix
min(../)
Prefix
float.min

**Arguments:** float*

**Examples:** Infix
min(&floatSlot, 4.0)

Prefix
(float.min &floatSlot 4.0)

**negate**
Returns the negation of the float argument.

**Synopsis:** Infix
neg(.

Prefix
float.neg

**Arguments:** float

**Examples:** Infix
neg(&floatSlot)

Prefix
(float.neg &floatSlot)

**range**
Returns the difference between the maximum and minimum valued arguments.
Takes two or more arguments.

**Synopsis:** Infix
range(.

Prefix
float.range

**Arguments:** float*

**Examples:** Infix
range(&floatSlot, 4.0, 100.5)

Prefix
(float.range &floatSlot 4.0 100.5)

**String predicates**
The following string predicates are described in the next sections:

- ==
- !=
- >
- >=
- <
• <=
• endsWith
• ignorecase equal
• ignorecase not equal
• startsWith

==
Mathematical equal to. Returns true if the arguments are equal.

**Synopsis:** Infix
==

Prefix
string.eq

**Arguments:** string, string

**Examples:** Infix
(&stringSlot == "somestr")

Prefix
(string.eq &stringSlot "somestr")

!=
Mathematical not equal to. Returns true if the arguments are not equal.

**Synopsis:** Infix
!=

Prefix
string.ne

**Arguments:** string, string

**Examples:** Infix
(&stringSlot != "somestr")

Prefix
(string.ne &stringSlot "somestr")

>
Mathematical greater than. Returns true if the first argument is greater than the second.

**Synopsis:** Infix
>

Prefix
string.gt

**Arguments:** string, string

**Examples:** Infix
(&stringSlot > "somestr")
Prefix
(string.gt &stringSlot "somestr")

>=
Mathematical greater than or equal to. Returns true if the first argument is greater than or equal to the second.

Synopsis: Infix
>=

Prefix
string.ge

Arguments: string, string

Examples: Infix
(&stringSlot >= "somestr")

Prefix
(string.ge &stringSlot "somestr")

<
Mathematical less than. Returns true if the first argument is less than the second.

Synopsis: Infix
<

Prefix
string.lt

Arguments: string, string

Examples: Infix
(&stringSlot < "somestr")

Prefix
(string.lt &stringSlot "somestr")

<=
Mathematical less than or equal to. Returns true if the first argument is less than or equal to the second.

Synopsis: Infix
<=

Prefix
string.le

Arguments: string, string

Examples: Infix
(&stringSlot <= "somestr")

Prefix
(string.le &stringSlot "somestr")
**contains**
Returns true if the second argument is a substring of the first argument.

**Synopsis:** Infix
contains(..)

Prefix
string.contains

**Arguments:** string, string

**Examples:**

Infix
contains(&stringSlot, "somestr")

Prefix
(string.contains&stringSlot "somestr")

**endsWith**
Returns true if the first argument ends with the second argument.

**Synopsis:** Infix
endsWith(..)

Prefix
string.endsWith

**Arguments:** string, string

**Examples:**

Infix
endsWith(&stringSlot, "somestr")

Prefix
(string.endsWith &stringSlot "somestr")

**ignorecase equal**
Returns true if the arguments are equal. The string case is ignored.

**Synopsis:** Infix
iceq(..)

Prefix
string.iceq

**Arguments:** string, string

**Examples:**

Infix
iceq(&stringSlot, "somestr")

Prefix
(string.iceq &stringSlot "somestr")

**ignorecase not equal**
Returns true if the arguments are not equal. The string case is ignored.

**Synopsis:** Infix
icne(..)

Prefix
string.icne

Arguments: string, string

Examples: Infix
icne(&stringSlot, "somestr")

Prefix
(string.icne &stringSlot "somestr")

**startsWith**

Returns true if the first argument starts with the second argument.

Synopsis: Infix
startsWith(..)

Prefix
string.startsWith

Arguments: string, string

Examples: Infix
startsWith(&stringSlot, "somestr")

Prefix
(string.startsWith &stringSlot "somestr")

**String functions**

The following string functions are described in the next sections:

- concat
- dayOfWeek
- dayOfWeekString
- existRuleId
- getEventType
- hourOfDay
- length
- month
- monthString
- substring
- toUpperCase
- toLowercase
- trim

**concat**

Returns a string formed by concatenating the arguments. Takes two or more arguments.

Synopsis: Infix
concat(..)
Prefix
string.concat

**Arguments:** string*

**Examples:** Infix
concat(&stringSlot, "-1")

Prefix
(string.concat &stringSlot "-1")

doOfWeek
Returns the day of the week as an integer (1 through 7 corresponding to Sunday through Saturday), given as a date string. Date formats must be one of the following:

```
yyyy/MM/dd,HH:mm:ss
   Without a time zone, assuming UTC
```

```
yyyy/MM/dd,HH:mm:ss z
   With a time zone, for example, 2001/10/23,14:30:33 PDT
```

```
yyyy/MM/dd,HH:mm:ss'Z'
   Without a time zone, but with Z to indicate UTC
```

**Synopsis:** Infix
doOfWeek(..)

Prefix
string.dayOfWeek

**Arguments:** string

**Examples:** Infix
doOfWeek("2001/10/23,14:30:33")

Prefix
(string.dayOfWeek "2001/10/23,14:30:33")

doOfWeekString
Returns the day of the week as a string (Sunday through Saturday), given as a date string. Date formats must be one of the following:

```
yyyy/MM/dd,HH:mm:ss
   Without a time zone, assuming UTC
```

```
yyyy/MM/dd,HH:mm:ss z
   With a time zone, for example, 2001/10/23,14:30:33 PDT
```

```
yyyy/MM/dd,HH:mm:ss'Z'
   Without a time zone, but with Z to indicate UTC
```

**Synopsis:** Infix
doOfWeekString(..)

Prefix
string.dayOfWeekString

**Arguments:** string
Examples: Infix  
\texttt{dayOfWeekString("2001/10/23,14:30:33")}

Prefix  
\texttt{(string.dayOfWeekString "2001/10/23,14:30:33")}

\textbf{existRuleId}  
Returns true if the given rule identifier exists.

\textbf{Synopsis:} Infix  
existRuleId(\ldots)

Prefix  
\texttt{string.existRuleId}

\textbf{Arguments:} string

Examples: Infix  
existRuleId("tree.left")

Prefix  
\texttt{(string.existRuleId "tree.left")}

\textbf{getEventType}  
Returns the type of the event currently processed as a string.

\textbf{Synopsis:} Infix  
gETEventType(\ldots)

Prefix  
\texttt{string.getEventType}

\textbf{Arguments:} There are no arguments.

Examples: Infix  
gETEventType()  

Prefix  
\texttt{(string.getEventType)}

\textbf{hourOfDay}  
Returns the hour of the day as an integer using the 24-hour clock, given as a date string. Date formats must be one of the following:

\texttt{yyyy/MM/dd,HH:mm:ss}  
Without a time zone, assuming UTC

\texttt{yyyy/MM/dd,HH:mm:ss z}  
With a time zone, for example, 2001/10/23,14:30:33 PDT

\texttt{yyyy/MM/dd,HH:mm:ss'Z'}  
Without a time zone, but with Z to indicate UTC

\textbf{Synopsis:} Infix  
hourOfDay(\ldots)

Prefix
string.hourOfDay

**Arguments:** string

**Examples:** Infix

    hourOfDay("2001/10/23,14:30:33")

  Prefix

    (string.hourOfDay "2001/10/23,14:30:33")

**length**

Returns the length (integer) of the string argument.

**Synopsis:** Infix

    length(••)

  Prefix

    string.length

**Arguments:** string

**Examples:** Infix

    length(&stringSlot)

  Prefix

    (string.length &stringSlot)

**month**

Returns the month of the year as an integer (zero through 11 corresponding to January through December), given as a date string. Date formats must be one of the following:

- yyyy/MM/dd,HH:mm:ss
  - Without a time zone, assuming UTC

- yyyy/MM/dd,HH:mm:ss z
  - With a time zone, for example, 2001/10/23,14:30:33 PDT

- yyyy/MM/dd,HH:mm:ss'Z'
  - Without a time zone, but with Z to indicate UTC

**Synopsis:** Infix

    month(••)

  Prefix

    string.month

**Arguments:** string

**Examples:** Infix

    month("2001/10/23,14:30:33")

  Prefix

    (string.month "2001/10/23,14:30:33")
**monthString**
Returns the month of the year as a string (January through December), given as a date string. Date formats must be one of the following:

- `yyyy/MM/dd,HH:mm:ss`
  Without a time zone, assuming UTC
- `yyyy/MM/dd,HH:mm:ss z`
  With a time zone, for example, 2001/10/23,14:30:33 PDT
- `yyyy/MM/dd,HH:mm:ss'Z'`
  Without a time zone, but with Z to indicate UTC

**Synopsis:**
Infix
monthString(…)  
Prefix
string.monthString

**Arguments:** string

**Examples:**
Infix
monthString("2001/10/23,14:30:33")
Prefix
(string.monthString "2001/10/23,14:30:33")

**substring**
Returns a substring of the string argument, marked by start and end characters. Takes three arguments.

**Synopsis:**
Infix
substring(…)
Prefix
string.substring

**Arguments:** string, int (start), int (end)

**Examples:**
Infix
substring(&stringSlot, 3, 5)
Prefix
(string.substring &stringSlot 3 5)

**toUpperCase**
Returns the argument string with all characters in upper case.

**Synopsis:**
Infix
toUpperCase(…)
Prefix
string.toUpperCase

**Arguments:** string

**Examples:**
Infix
toUpperCase(&stringSlot)

Prefix
(string.toUpperCase &stringSlot)

**toLowerCase**

Returns the argument string with all characters in lower case.

**Synopsis:** Infix
toLowerCase(..)

Prefix
string.toLowerCase

**Arguments:** string

**Examples:** Infix
toLowerCase(&stringSlot)

Prefix
(string.toLowerCase &stringSlot)

**trim**

Returns the argument string with all leading and trailing spaces removed.

**Synopsis:** Infix
trim(..)

Prefix
string.trim

**Arguments:** string

**Examples:** Infix
trim(&stringSlot)

Prefix
(string.trim &stringSlot)

**IP address predicates**

The following IP address predicates are described in the next sections:

- ==
- !=

**==**

Mathematical equal to. Returns true if the arguments are equal.

**Synopsis:** Infix
==

Prefix
ipaddress.eq

**Arguments:** ipaddress, ipaddress
Examples: Infix
(&ipAddrSlot == 9.4.3.2)

Prefix
(ipaddress.eq &ipAddrSlot 9.4.3.2)

!=
Mathematical not equal to. Returns true if the arguments are not equal.

Synopsis: Infix
!=

Prefix
ipaddress.ne

Arguments: ipaddress, ipaddress

Examples: Infix
(&ipAddrSlot != 9.4.3.2)

Prefix
(ipaddress.ne &ipAddrSlot 9.4.3.2)

String set predicates
The following string set predicates are described in the next sections:

• ==
• !=
• areDisjoint
• isMemberOf
• isSubsetOf

==
Mathematical equal to. Returns true if the arguments are equal.

Synopsis: Infix
==

Prefix
stringset.eq

Arguments: set, set

Examples: Infix
(&setSlot == [1, 5])

Prefix
(stringset.eq &setSlot [1, 5])

!=
Mathematical not equal to. Returns true if the arguments are not equal.

Synopsis: Infix
!=

Chapter 9. State correlation reference 335
Prefix
stringset.ne

Arguments: set, set

Examples: Infix
(&setSlot != [1, 5])

Prefix
(stringset.ne &setSlot [1, 5])

areDisjoint
Returns true if the two arguments are disjoint, that is they have no members in common.

Synopsis: Infix
areDisjoint(..)

Prefix
stringset.areDisjoint

Arguments: set, set

Examples: Infix
areDisjoint(&setSlot, ["off"])

Prefix
(stringset.areDisjoint &setSlot ["off"])
Arguments: set, set

Examples: Infix
isSubsetOf(&setSlot, [1, 2, 3])

Prefix
(stringset.isSubsetOf &setSlot [1, 2, 3])

**String set functions**
The following string set functions are described in the next sections:
- intersect
- join
- previousValues

**intersect**
Returns a set resulting from the intersection of the argument sets. Takes two or more arguments.

**Synopsis:** Infix
intersect(..)

Prefix
stringset.intersect

Arguments: set*

Examples: Infix
intersect(&setSlot1, &setSlot2)

Prefix
(stringset.intersect &setSlot1 &setSlot2)

**join**
Returns a set created by joining the argument sets. Takes two or more arguments.

**Synopsis:** Infix
join(..)

Prefix
stringset.join

Arguments: set*

Examples: Infix
join(&setSlot, ["ibm", "tivoli"])

Prefix
(stringset.join &setSlot ["ibm", "tivoli"])

**previousValues**
Returns a set of values corresponding to the argument attribute type from the set of events currently matched in the state machine.

**Synopsis:** Infix
previousValues(..)

Prefix
stringset.previousValues

**Arguments:** string (attribute_type)

**Examples:** Infix
previousValues("hostname")

Prefix
(stringset.previousValues "hostname")

## Boolean predicates

The following boolean predicates are described in the next sections:

- **and**
- **not**
- **or**

**and**

Returns the result of the boolean operator and of the arguments. Takes two or more arguments.

**Synopsis:** Infix
```Java
&&
```

Prefix
```Java
boolean.and
```

**Arguments:** boolean*

**Examples:** Infix
```Java
(&booleanSlot && (intSlot > 4))
```

Prefix
```Java
(boolean.and &booleanSlot (int.gt &intSlot 4))
```

**not**

Returns the result of the boolean negation of the arguments.

**Synopsis:** Infix
```Java
!
```

Prefix
```Java
boolean.not
```

**Arguments:** boolean

**Examples:** Infix
```Java
(!&booleanSlot)
```

Prefix
```Java
(boolean.not &booleanSlot)
```
**or**
Returns the result of the boolean operator or of the arguments. Takes two or more arguments.

Synopsis: Infix
||

Prefix
boolean.or

Arguments: boolean

Examples: Infix
(&booleanSlot || (&intSlot > 4))

Prefix
(boolean.or &booleanSlot (int.gt &intSlot 4))

**Integer set predicate**
The following is an integer set predicate.

**isMemberOf**
Returns true if the first argument of the type integer is a member of the second, an integer set.

Synopsis: Infix
isMemberOf(..)

Prefix
intset.isMemberOf

Arguments: integer, intset

Examples: Infix
isMemberOf(5, [1, 3-6, 9])

Prefix
(intset.isMemberOf 5 [1, 3-6, 9])

**IP address set predicate**
The following is an IP address set predicate.

**isMemberOf**
Returns true if the first argument of the type ipaddress is a member of the second, an ipaddresset set.

Synopsis: Infix
isMemberOf(..)

Prefix
ipaddress.isMemberOf

Arguments: ipaddress, ipaddressset

Examples: Infix
State correlation messages

The following are state correlation messages. Tivoli Event Integration Facility does not use some of the following messages; they are used by other applications. The messages are listed in alphanumeric order by message identifier.

**ECOZC1001E** The state correlation XML configuration file was not found at URL url.

*Explanation:* The location where the state correlation XML configuration file should reside has an incorrect URL format, or the file was not found.

*Administrator Response:* Check the URL format for the file specification, and the location for the file.

**ECOZC1002E** There was an error with the state correlation engine during initialization.

*Explanation:* The state correlation engine was not initialized correctly.

*Administrator Response:* Contact Tivoli Customer Support.

**ECOZC1003E** There was an error with the state correlation engine during initialization or while attempting to read configuration data.

*Explanation:* The state correlation engine was not initialized correctly.

*Administrator Response:* Contact Tivoli Customer Support.

**ECOZC1004E** Error when attempting to delete or change the state correlation rule rule.

*Explanation:* An attempt was made to delete or change a state correlation rule that does not exist.

*Administrator Response:* Contact Tivoli Customer Support.

**ECOZC1005E** Predicate predicate does not exist for a state correlation rule.

*Explanation:* A state correlation rule specifies an undefined predicate.

*Administrator Response:* Ensure the state correlation XML configuration file specifies only valid predicates in rules. See the Tivoli Event Integration Facility User’s Guide for information about predicates.

**ECOZC1006E** The action class class and function function are incorrect in a state correlation rule.

*Explanation:* An incorrect action was found in a state correlation rule.

*Administrator Response:* Ensure the state correlation XML configuration file specifies only valid actions in rules. See the Tivoli Event Integration Facility User’s Guide for information about rule actions.

**ECOZC1007E** The action class class and function function will not accept the parameter parameter in a state correlation rule.

*Explanation:* An action in a state correlation rule specifies an incorrect parameter.

*Administrator Response:* Ensure the state correlation XML configuration file specifies only valid actions in the rules and the actions specify valid parameters. See the Tivoli Event Integration Facility User’s Guide for information about rule actions.

**ECOZC1008E** Could not create persistent storage file file for state correlation.

*Explanation:* An incorrect file name was specified or there was a file system error when creating the file for persistent storage with state correlation.

*Administrator Response:* Ensure the file system where persistent storage for state correlation is being created exists, has write permissions, and has sufficient space. Also, ensure the file name is correct.

**ECOZC1009E** The location directory for the persistent storage file used with state correlation is incorrect.

*Explanation:* An incorrect location for creating the persistent storage file was specified.

*Administrator Response:* Ensure the file system where persistent storage for state correlation is being created exists, has write permissions, and has sufficient space.
ECOZC1010E Syntax error in the state correlation XML configuration file.

Explanation: The state correlation XML configuration file contains a syntax error.

Administrator Response: Ensure the state correlation XML configuration file exists and is syntactically correct.

ECOZC1011E Error reading the state correlation XML configuration file.

Explanation: The state correlation XML configuration file could not be read and therefore the state correlation engine could not run.

Administrator Response: Contact Tivoli Customer Support.

ECOZC1012E Error reading the state correlation XML configuration file.

Explanation: The state correlation engine was not initialized correctly.

Administrator Response: Contact Tivoli Customer Support.

ECOZC1013E Syntax error in the state correlation labels configuration file.

Explanation: The state correlation labels configuration file contains a syntax error.

Administrator Response: Ensure the state correlation labels configuration file exists and is syntactically correct.

ECOZC1014E Syntax error with a label declaration in the state correlation labels configuration file.

Explanation: A label declaration in the state correlation labels configuration file has a syntax error.

Administrator Response: Ensure the state correlation labels configuration file is syntactically correct.

ECOZC1015E Error reading the state correlation XML configuration file. The defaultType parameter value is not valid.

Explanation: An error occurred when reading the defaultType parameter in the state correlation XML configuration file.

Administrator Response: Ensure the defaultType parameter specified in the state correlation XML configuration file is a valid type. Valid types are Int, Float, Set, Boolean, and String.

The defaultType parameter is required when applications do not pass event metadata information, like a BAROC file, to the state correlation engine.

ECOZC1016E Syntax error in the state correlation XML configuration file. The predicate predicate is incorrect.

Explanation: The predicate in the state correlation XML configuration file contains a syntax error.

Administrator Response: Ensure the predicate is valid and syntactically correct. See the Tivoli Event Integration Facility for information about predicates.

ECOZC1017E Lexical error with term term found in predicate predicate in the state correlation XML Configuration file.

Explanation: A lexical error was found with the term in the predicate.

Administrator Response: Correct the term in the predicate. See the Tivoli Event Integration Facility User's Guide for information about predicates.

ECOZC1018E Error in action action in state correlation rule rule.

Explanation: The state correlation engine was not initialized correctly.

Administrator Response: Contact Tivoli Customer Support.

ECOZC1019E Error when trying to add rule rule. Rules folder folder already exists.

Explanation: An attempt was made to add a rule that refers to an existing rules folder. The rules folder must be deleted before the rule can be added successfully.

Administrator Response: Contact Tivoli Customer Support.

ECOZC1020E Error when trying to add rule rule1. Its path is already specified for rule rule2.

Explanation: The rule path refers to an existing rule. The existing rule must be deleted before the new rule can be added successfully.

Administrator Response: Contact Tivoli Customer Support.

ECOZC1021E Error when recovering persisted events after a failure. The events not recovered can be found in file: file.

Explanation: Persisted events were not recovered. This error can occur because of communications or file system problems.

Administrator Response: Ensure there are no communications or file systems problems in the
environment where the state correlation engine is running. If the problem persists, contact Tivoli Customer Support.

ECOZC1022E  A malformed line was found in the declaration of the state correlation labels configuration file.

Explanation:  The declaration in the state correlation labels configuration file has a malformed line.

Administrator Response:  The label line in the declaration must use the format: event_type name type.

ECOZC1023E  Unrecognized type type in the state correlation labels file.

Explanation:  The state correlation labels configuration file contains an unrecognized type in the declaration.

Administrator Response:  Correct the unrecognized type.

ECOZC1024E  Error when reading the state correlation labels configuration file file.

Explanation:  An incorrect state correlation labels configuration file was specified, or a file system error occurred.

Administrator Response:  Contact Tivoli Customer Support.

ECOZC1025E  The state correlation engine is in suspend mode. Event event, was not processed.

Explanation:  The state correlation engine was placed into suspend mode to halt processing of events. The most common causes of this error are communications or file system problems.

Administrator Response:  Ensure there are no communications or file system problems in the environment where the state correlation engine is running. If the problem persists, contact Tivoli Customer Support.

ECOZC1026E  The total size for persistent storage (totalSize) should be at least two times larger than the maximum size of the working persistent storage file (maxFileSize).

Explanation:  The state correlation engine was not initialized correctly.

Administrator Response:  Ensure the parameter for the total size of the persistent storage directory (persistence.provider.totalSize) is at least two times the maximum size of the working persistent storage file parameter (persistence.provider.maxFileSize).
Part 3. Appendixes
Appendix A. Using Prolog in rules

The Tivoli Enterprise Console rule language is precompiled into Prolog source code, which is then compiled into Prolog executable code. For this reason, you might want to include various features of Prolog in your rules.

This appendix provides an overview of Prolog and describes some of the more common Prolog features used in Tivoli Enterprise Console rule development, including reference information about some of the built-in predicates and operators in the version of Prolog supported by the rule language. See “Related publications” on page xi for more resources pertaining to Prolog.

Prolog predicate online information

Information in HTML format about each Prolog predicate described in this section, along with other IBM Tivoli Enterprise Console online reference information, is available on the event server host at:

$BINDIR/../generic_unix/TME/TEC/BOOKS/HTML/reference.html

It is also available on the product CD at:

/BOOKS/HTML/reference.html

Language basics

Prolog is a computer programming language that derives its power from the ability to define relationships between objects and to infer information from those relationships. These objects, referred to as terms in Prolog, are the building blocks of the language.

Terms

A term can be a constant, a variable, or a compound term, described as follows:

Constant

An atom or a number. An atom is a text constant beginning with a lowercase letter, similar to a string in other programming languages. The terms atom and string can be used interchangeably. Numbers refer to signed integers and signed real numbers.

Variable

An unknown term. A variable begins with an uppercase letter or underscore, and can be used in place of an unknown term in the same way as a mathematical variable. The scope of a variable is limited to a single action in a Tivoli Enterprise Console rule.

Compound term

A predicate or a list. A predicate defines a relationship between other terms. It consists of a functor, which names the term, and one or more arguments to the predicate. A predicate is similar to a function in other programming languages. The functor also serves as the name of a predicate, similar to a function name.

Predicates are written as follows:
functor(arg1, ...)

The functor is an atom and must follow the naming conventions of an atom. The name of the predicate should describe the relationship between its arguments. The number of arguments to a predicate is the predicate’s arity. For example, the following predicate is a term with functor in_state and arity of 2. The functor defines the relationship between Austin and Texas (specifically, that Austin is in the state of Texas):

\[\text{in\_state('Austin', 'Texas')}\]

A list is an ordered sequence of zero or more terms that can be constants, variables, or other lists. The following example shows how lists are written:

\[\text{['Austin', 'Dallas', 'Houston']}\]

An empty list, or null list, is written as follows:

\[\text{[]}\]

**Facts**

A fact in a Prolog program defines a piece of information about the known world—it defines a relationship between objects. For example, the knowledge that Austin is a city can be represented by the following fact:

\[\text{city('Austin')}\]

The knowledge that Austin is a city in the state of Texas can be represented by the following fact:

\[\text{in\_state('Austin', 'Texas')}\]

The knowledge that Texas is a state in the USA can be represented by the following fact:

\[\text{in\_country('Texas', 'USA')}\]

**Rules**

A rule in a Prolog program defines the relationships about a set of facts. (A Prolog rule is not the same as a Tivoli Enterprise Console rule.) A rule defines some fact that depends on some other set of facts (or rules). A rule consists of a head and a body, separated by the :- operator (comprised of a colon and a hyphen and referred to as the if operator). The head of a rule is a predicate. The body of the rule is a conjunction of other predicates, facts, or more rules, each of which must succeed for the rule to succeed. For example, if you wanted to create a rule that contained the knowledge that a city is in the USA only if the city was in a state of the USA, you could create the following rule:

\[\text{in\_usa(_city) :-}
\begin{align*}
\text{city(_city),} \\
\text{in\_state(_city, _state),} \\
\text{in\_country(_state, 'USA')}.
\end{align*}\]

For the in_usa rule to succeed, each predicate within the rule must succeed. First, the rule ensures that the city passed to the city predicate (through the _city argument of the in_usa predicate) is known in the knowledge base. If so, the city predicate succeeds. Next, if the city is known to be in a state of the USA, then the in_state predicate succeeds and the name of the state to which the city belongs is
in the _state variable. Lastly, if the state is in the USA, the in_country predicate succeeds. If all three predicates succeed, then the rule succeeds.

Note that each predicate in this rule, except for the last one, is followed by a comma. When used between predicates, the comma is the conjunction operator. This operator basically performs a logical AND operation between these predicates, where each predicate must succeed for the next one to be evaluated. All predicates in this clause must succeed for the rule to succeed. The period after the last predicate in the rule marks the end of the rule.

Knowledge base

Facts and rules comprise what is known as the knowledge base. The knowledge base is a database that contains everything known to be true to the running Prolog program. Facts and rules are added to the knowledge base with an assert predicate. If you wanted to add the two following facts to the knowledge base, you can assert them as follows:

\[
\text{assert(city('Austin'))}
\]

\[
\text{assert(in_state('Austin', 'Texas'))}
\]

A rule can be asserted in the same manner, but an extra set of parentheses is required if the rule contains the :- operator or commas. The rule in_usa(_city) is asserted into the knowledge base as follows:

\[
\text{assert((in_usa(_city) :- city(_city), in_state(_city, _state), in_country(_state, 'USA')))}
\]

A Prolog program can later query the knowledge base by searching through it for a fact or rule that matches the predicate. To query the knowledge base for a particular fact or rule, the predicate is put on a line by itself. To determine whether Austin is in the state of Texas, you can write:

\[
\text{city('Austin')}
\]

The city predicate succeeds if there is a fact or rule in the knowledge base with the name of city and an argument of Austin. If there is not, then the predicate fails and is considered to not be true. For a rule to satisfy a query, each fact or rule that comprises the rule must succeed. If just one fact or rule in a rule fails, the entire rule fails. If you want to know if Austin is in the USA, you can write the following predicate to query the knowledge base:

\[
\text{in_usa('Austin')}
\]

The predicate in_usa('Austin') runs the following queries in the order shown:

\[
\text{city(_city)}
\]

\[
\text{in_state(_city, _state)}
\]

\[
\text{in_country(_state, 'USA')}
\]

If all three of the predicates in the previous example succeed, then the in_usa('Austin') predicate succeeds.

Information can be returned in a predicate through variables. The following predicate queries the knowledge base for the state in which the city of Austin is
located. This predicate matches the fact \texttt{in\_state('Austin', 'Texas')} and the value Texas is assigned to the \_state variable, which can then be used in subsequent queries.

\texttt{in\_state('Austin', \_state)}

\textbf{Comments}

There are two forms of comment delimiters that can be used in Prolog code and in Tivoli Enterprise Console rules. Text embedded within the /* (comprised of a forward slash and an asterisk) and */ (comprised of an asterisk and a forward slash) delimiters is treated as a comment and ignored by the compiler. You can create comments that span multiple lines using these delimiters. The other comment delimiter is the % (percent). All text after this character until the end of the single line is a comment. You cannot nest the first form of comments, as it will cause a compilation error. You can use a % character within /* */ delimiters, but it is treated as literal text and not a comment delimiter.

\textbf{Data Types and Ranges}

The following list describes the range for each Prolog data type:

\textbf{Atom} A string that must begin with a lowercase letter and can contain letters, digits, and the underscore character. If an atom is contained within single quotation marks it can contain any character. The length of an atom is restricted to 32767 characters. Note that the BAROC attribute type of STRING can hold a maximum of 255 characters.

\textbf{Atomic} A general term that refers to an integer, real, pointer, or an atom data type.

\textbf{Integer} A 29-bit signed value with a range from $-2^{28}$ to $2^{28}-1$. The default base is 10, but numbers can be represented in any base from 1 to 36. The notation for representing a non-base 10 number is \textit{base'number}. For example, 2'1111011, 8'173, and 16'7B all represent the decimal number 123 in binary, octal, and hexadecimal, respectively. Integers can also be expressed in scientific notation; for example, 1.23e+2 represents 123.

\textbf{List} The length of a list is limited only by the available memory of the heap.

\textbf{Number} A general term that refers to either an integer or real data type.

\textbf{Pointer} The pointer type is used in Tivoli Enterprise Console rules to represent a handle to the event under analysis, certain dates (for example, the values for the date\_reception attribute and cause\_date\_reception attribute) and other INT32 values in a BAROC file. A pointer value is coded in 32 bits and is represented by a hexadecimal number beginning with 0x.

\textbf{Predicate} The maximum arity of a predicate is 64.

\textbf{Real} A signed double precision value with approximately 16 significant decimal digits. Real numbers can be written in decimal and scientific notation. The numbers 1.23 and 123e-2 both represent the real number 1.23. The range for real numbers is $\pm2.250738585072014e-308$ through $\pm1.79769313486231570e+308$. 
Data type mapping between BAROC and Prolog
Tivoli Enterprise Console event attributes are defined with BAROC data types. If you are adding Prolog code to your Tivoli Enterprise Console rules, you need to know how BAROC data types map to Prolog data types. The following table shows the mapping.

<table>
<thead>
<tr>
<th>BAROC Type</th>
<th>Prolog Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENUM</td>
<td>atom</td>
</tr>
<tr>
<td>INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>INT32</td>
<td>pointer</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>STRING</td>
<td>atom</td>
</tr>
<tr>
<td>LIST</td>
<td>list</td>
</tr>
</tbody>
</table>

Example of using Prolog in a Tivoli Enterprise Console rule
All of the Prolog predicates used in the preceding examples can be used in a Tivoli Enterprise Console rule. Prolog statements used in a Tivoli Enterprise Console rule must be contained within an action clause of a Tivoli Enterprise Console rule. The example on page 350 shows what the preceding examples of facts and rules would look like in a Tivoli Enterprise Console rule. Before viewing the following example, some additional terms need to be defined:

**Free**
Describes a variable that does not yet have a value.

**Instantiated**
Describes a variable that has been assigned a value. Once a variable has been instantiated, it can be used in other predicates within the same action clause. Instantiated is sometimes called *ground*.

**Unification**
The process of making two variables equal. If the unification of two variables is possible, the operation succeeds; otherwise, the operation fails. In the following example on the line `in_state('Houston', _state)`, the _state variable is unified with the Texas atom. Before this operation the _state variable did not have a value assigned to it. Unification can also be achieved with the = operator (unification operator).
Quick reference of Prolog predicates and operators

This section categorizes the built-in predicates and operators by the function they provide. Each subsection contains a table that lists and briefly describes the predicates and operators for a particular category.

**Instantiation and unification**

Unification tries to make two terms equal by the substitution of variables. A query matches a fact or a rule if it has the same name and if the arguments match. If the arguments are variables, then they are unified with a value to make the terms match. When a variable is unified with a value, the variable is instantiated.
Variables can also be instantiated with the = operator. If a variable has already been instantiated with a value, it can be tested with the == operator.

The following table lists the predicates and operators of this category.

<table>
<thead>
<tr>
<th>Predicate/Operator</th>
<th>Description</th>
</tr>
</thead>
</table>
| \
| Term inequality operator |
| =                  | Term unification operator |
| =?                 | Expression evaluation |
| ==                 | Term equality operator |
| not                | Term negation predicate |

**Test data type and variable instantiation**

Test predicates are used to obtain the data type of a term or to test whether a variable has been instantiated.

The following table lists the predicates and operators of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>Tests whether a variable is of type atom.</td>
</tr>
<tr>
<td>atomic</td>
<td>Tests whether a variable is of type atomic.</td>
</tr>
<tr>
<td>ground</td>
<td>Tests whether a variable is instantiated.</td>
</tr>
<tr>
<td>integer</td>
<td>Tests whether a variable is of type integer.</td>
</tr>
<tr>
<td>is_list</td>
<td>Tests whether a variable is a list.</td>
</tr>
<tr>
<td>number</td>
<td>Tests whether a variable is of type number.</td>
</tr>
<tr>
<td>pointer</td>
<td>Tests whether a variable is of type pointer.</td>
</tr>
<tr>
<td>real (test real type)</td>
<td>Tests whether a variable is of type real.</td>
</tr>
<tr>
<td>term_type</td>
<td>Gets the type of a variable.</td>
</tr>
</tbody>
</table>

**Convert data types**

Conversion predicates are used to convert the data type of a term.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>Converts between an ASCII character and ASCII code.</td>
</tr>
<tr>
<td>inttoatom</td>
<td>Converts between an integer and an atom representation of an integer.</td>
</tr>
<tr>
<td>pointertoatom</td>
<td>Converts between a pointer and an atom representation of a pointer.</td>
</tr>
<tr>
<td>pointertoint</td>
<td>Converts between a pointer and an integer representation of a pointer.</td>
</tr>
<tr>
<td>real (convert integer)</td>
<td>Converts integer type to real type.</td>
</tr>
<tr>
<td>realtimeatom</td>
<td>Converts between a real number and an atom representation of a real number.</td>
</tr>
</tbody>
</table>
Atom manipulation

Strings are not implemented in Prolog as they are in other programming languages. The closest data type in Prolog to a string is an atom. An atom is written within single quotation marks, as in ‘atom’. An atom can also be represented as a Prolog list type of one-character atoms (for example, [a,t,o,m]) or as a list of ASCII codes (for example, [97,116,111,109]). If you write an atom with double quotation marks (for example, "string"), what you actually define is a list of ASCII codes (for example, [115, 116, 114, 105, 110, 103]).

Standard order comparison

When terms of different types are compared (for example, atoms to numbers), they are ordered as follows:

- Variables @< atoms @< numbers @< pointers @< predicates. This means variables precede (are less than) atoms, atoms precede numbers, and so forth.

Otherwise, terms are ordered as follows:

- Variables are ordered according to their age.
- Atoms are ordered alphabetically.
- Numbers are ordered numerically.
- Pointers are ordered numerically.
- Predicates are ordered according to their arity.
  - If the arities are equal, they are ordered according to their name.
  - If the name and arity are equal, they are ordered recursively to their arguments, from left to right.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@&lt;</td>
<td>Tests whether a term alphabetically precedes another.</td>
</tr>
<tr>
<td>@=&lt;</td>
<td>Tests whether a term is alphabetically equal or precedes another.</td>
</tr>
<tr>
<td>@&gt;</td>
<td>Tests whether a term alphabetically follows another.</td>
</tr>
<tr>
<td>@&gt;=</td>
<td>Tests whether a term is alphabetically equal or follows another.</td>
</tr>
</tbody>
</table>

Functions

These predicates are used like function calls in other programming languages. They compute a value and return it to the variable on the left side of the expression. This is done by unification, except that the =? operator is used instead of = operator to perform the unification.

The following table lists the predicates of this category:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_to_hex</td>
<td>Hexadecimal string representation of an integer</td>
</tr>
<tr>
<td>strip</td>
<td>Remove characters from an atom</td>
</tr>
<tr>
<td>substring</td>
<td>Get a substring from a string</td>
</tr>
</tbody>
</table>

Miscellaneous

These predicates perform a variety of actions with atoms.
The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomlength</td>
<td>Determine the length of an atom</td>
</tr>
<tr>
<td>atomconcat</td>
<td>Concatenate atoms</td>
</tr>
<tr>
<td>atompart</td>
<td>Get a substring from an atom</td>
</tr>
<tr>
<td>lowertoupper</td>
<td>Convert between lowercase and uppercase letters in an atom</td>
</tr>
<tr>
<td>sprintf</td>
<td>Print formatted data to an atom</td>
</tr>
</tbody>
</table>

**List manipulation**

With list manipulation predicates, you can perform various operations on list data types.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Append elements to a list</td>
</tr>
<tr>
<td>atomtolist</td>
<td>Convert between the atomic type and a list of characters</td>
</tr>
<tr>
<td>delete</td>
<td>Delete elements from a list</td>
</tr>
<tr>
<td>disjoint</td>
<td>Compare elements in two lists for uncommon values</td>
</tr>
<tr>
<td>empty_list</td>
<td>Test if a list is empty</td>
</tr>
<tr>
<td>intersect</td>
<td>Compare elements in lists for common values</td>
</tr>
<tr>
<td>length</td>
<td>Get the length of a list</td>
</tr>
<tr>
<td>member</td>
<td>Check a list for a value</td>
</tr>
<tr>
<td>name</td>
<td>Convert between the atomic type and a list of character codes</td>
</tr>
<tr>
<td>nmember</td>
<td>Check a list for a single value at an index</td>
</tr>
<tr>
<td>nmembers</td>
<td>Check a list for multiple values at indexes</td>
</tr>
<tr>
<td>remove_dups</td>
<td>Remove duplicate elements from a list</td>
</tr>
<tr>
<td>rremove</td>
<td>Remove the first element from a list</td>
</tr>
<tr>
<td>sort</td>
<td>Sort the elements of a list alphabetically</td>
</tr>
<tr>
<td>subset</td>
<td>Test whether a list is a subset of another list</td>
</tr>
<tr>
<td>subtract</td>
<td>Remove elements that are common to two lists</td>
</tr>
<tr>
<td>union</td>
<td>Add uncommon elements between two lists</td>
</tr>
</tbody>
</table>

**Mathematical expressions**

With predicates and operators, you can perform mathematical operations.

**Expression evaluation**

These predicates are used when you need to evaluate a mathematical expression on the right side of the operator and return the result to the left side.

The following table lists the predicate of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>Mathematical unification</td>
</tr>
</tbody>
</table>
**Arithmetic comparison**

These predicates are used when you need to compare the results of mathematical expressions on the right and left side of the operator. Arguments on both sides of the operator must be instantiated.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Mathematical less than</td>
</tr>
<tr>
<td>==</td>
<td>Mathematical equal to</td>
</tr>
<tr>
<td>=/=</td>
<td>Mathematical not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Mathematical less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Mathematical greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Mathematical greater than or equal to</td>
</tr>
</tbody>
</table>

**Mathematical operators**

These operators are used when you need to perform mathematical operations.

The following table lists the operators for this category.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>– (sign reversal)</td>
<td>Unary negation operator</td>
</tr>
<tr>
<td>– (subtraction)</td>
<td>Subtraction operator</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
</tr>
<tr>
<td>/</td>
<td>Real division operator</td>
</tr>
<tr>
<td>//</td>
<td>Integer division operator</td>
</tr>
</tbody>
</table>

**Pointer arithmetic**

The pointer type in Tivoli Enterprise Console rules is used for two purposes:
- 32-bit integer storage
- dates stored in epoch time format (the number of seconds since the epoch)

BAROC file INT32 types are represented in Tivoli Enterprise Console rules as pointers.

Pointer arithmetic predicates and operators are used when you need to perform arithmetical operations with pointers.

The following table lists the predicates and operators of this category.

<table>
<thead>
<tr>
<th>Predicate/Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>– (pointer offset subtraction)</td>
<td>Pointer offset subtraction operator.</td>
</tr>
<tr>
<td>– (pointer subtraction)</td>
<td>Pointer subtraction operator.</td>
</tr>
<tr>
<td>+ (pointer offset addition)</td>
<td>Pointer offset addition operator.</td>
</tr>
<tr>
<td>pointeroffset</td>
<td>Get the difference between two pointer values.</td>
</tr>
</tbody>
</table>
Miscellaneous
These predicates perform a variety of functions with mathematical expressions.

The following table lists the predicate of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>round</td>
<td>Round a real number to the closest integer.</td>
</tr>
</tbody>
</table>

Knowledge base
The collection of predicates (facts and rules) in a Prolog program collectively make up what is known as the knowledge base. The knowledge base can be modified at run time by adding or removing predicates with the predicates listed in this section.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abolish</td>
<td>Remove all clauses from the knowledge base.</td>
</tr>
<tr>
<td>assert</td>
<td>Add a clause to the knowledge base.</td>
</tr>
<tr>
<td>compile</td>
<td>Compile a Prolog source file.</td>
</tr>
<tr>
<td>consult</td>
<td>Load a compiled Prolog file into the knowledge base.</td>
</tr>
<tr>
<td>flisting</td>
<td>Write predicates to an open file.</td>
</tr>
<tr>
<td>reconsult</td>
<td>Reload a compiled Prolog file into the knowledge base.</td>
</tr>
<tr>
<td>retract</td>
<td>Remove a specific clause from the knowledge base.</td>
</tr>
</tbody>
</table>

I/O
These predicates are used for reading and writing data from files. Be aware of the impacts to rule engine performance when using these predicates. Rule execution temporarily halts during any I/O operation until the operation is complete.

The following table lists the predicates of this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclose</td>
<td>Close an open file.</td>
</tr>
<tr>
<td>fopen</td>
<td>Open a file.</td>
</tr>
<tr>
<td>printf</td>
<td>Write formatted output to a file.</td>
</tr>
<tr>
<td>read</td>
<td>Read a Prolog term from an open Prolog source file.</td>
</tr>
<tr>
<td>readln</td>
<td>Read a line from an open file.</td>
</tr>
<tr>
<td>write</td>
<td>Write to an open file.</td>
</tr>
</tbody>
</table>

Alphabetic listing of built-in predicates and operators
The following section lists the predicates and operators in alphabetical order.
Addition operator.

**Synopsis**

_\texttt{number1} + _\texttt{number2}

**Description**

Operands can be integers or real numbers. If one or more of the operands is a real number, the result returned is a real number.

**Arguments**

_\texttt{number1}

A number. Must be integer or real.

_\texttt{number2}

A number. Must be integer or real.

**Examples**

The following example shows various uses of the predicate:

\begin{verbatim}
\texttt{integer1} is 100,
\texttt{integer2} is 30,
\texttt{real1} is 2.1,
\%
\texttt{Assign values.}

\texttt{sum1} is \texttt{integer1} + \texttt{integer2},
\%
\texttt{sum1} is unified with 130.

\texttt{sum2} is \texttt{real1} + \texttt{integer2}
\%
\texttt{sum2} is unified with 3.210000000000000e+01.
\end{verbatim}

**See Also**

None.
– (pointer offset subtraction)

Pointer offset subtraction operator.

**Synopsis**

\[ \text{pointer} - \text{integer} \]

**Description**
The first operand must be a pointer and the second must be an integer. The result is a pointer, so the \( =? \) operator must be used for returning results.

**Arguments**

\( \text{pointer} \)

A pointer.

\( \text{integer} \)

An integer.

**Examples**
The following example shows a Tivoli Enterprise Console rule that:

1. Computes the number of seconds since the reception of a first duplicate event using the pointeroffset predicate and unifies that value with the \_offset1 variable.
2. Computes the same result as in step\[1\] using pointer subtraction, and unifies the value with the \_offset2 variable.
3. Computes the date of the of the most recently received event by using pointer offset addition to add the \_offset2 variable to the reception date of the first duplicate.
4. Computes the reception date of the duplicate event by using pointer offset subtraction to subtract the offset between the reception dates of the original and duplicate events, from the reception date of the original event.

```prolog
rule: pointer_offset: (
  event: _event of_class _class
  where [  
    date_reception: _date_reception  
    % _date_reception unified with 0x37695cd4.
  ],
  action: ( 
    first_duplicate(_event, event: _dup_event 
    where [ 
      status: outside ['CLOSED'],  
      date_reception: _dup_date_reception  
      % _dup_date_reception unified with 0x376958d4.
    ]
  ),
  pointeroffset(_dup_date_reception, _offset1, _date_reception,  
  % 1. _offset1 is unified with 1024 (0x37695cd4  
  % - 0x376958d4).
  _offset2 is _date_reception - _dup_date_reception,  
  % 2. _offset2 unified with 1024 (0x37695cd4 -  
  % 0x376958d4).
  _originalDate =? _dup_date_reception + _offset2,  
  %3. _originalDate unifies with 0x37695cd4  
  % (0x376958d4 + 1024).
  _originalDupDate =? _date_reception - _offset1
)```
4. _originalDupDate unifies with 0x376958d4
   \( (0x37695cd4 - 1024) \).
)
}).

See Also
– (pointer subtraction)

Pointer subtraction operator.

Synopsis

_pointer1 – _pointer2

Description

Both operands must be pointers. The result is an integer, so the is operator must be used for returning the results.

Arguments

_pointer1

A pointer.

_pointer2

A pointer.

Examples

The following example shows a Tivoli Enterprise Console rule fragment that:

1. Computes the number of seconds since the reception of a first duplicate event using the pointeroffset predicate and unifies that value with the _offset1 variable.

2. Computes the same result as in step 1 using pointer subtraction, and unifies the value with the _offset2 variable.

rule: pointer_offset: {
  event: _event of_class _class
  where [
    date_reception: _date_reception
    % _date_reception unified with 0x37695cd4.
  ],
  action: {
    first_duplicate(_event, event: _dup_event
    where [
      status: outside ['CLOSED'],
      date_reception: _dup_date_reception
      % _dup_date_reception unified with 0x376958d4.
    ]
  ),

  pointeroffset(_dup_date_reception,
    _offset1,
    _date_reception),
  % 1. _offset1 is unified with 1024 (0x37695cd4 % – 0x376958d4).

  _offset2 is _date_reception -
  _dup_date_reception,
  % 2. _offset2 unified with 1024 (0x37695cd4 % - 0x376958d4).
}

See Also

is

Appendix A. Using Prolog in rules  359
Unary negation operator.

**Synopsis**

\[-_\text{number}\]

**Description**

Reverses the sign of \_number.

**Arguments**

\_number

The number whose sign to reverse. Must be integer or real.

**Examples**

The following example shows various uses of the operator:

\_real1 is 2.1,
\_integer1 is 100,
\%; Assign values.

\_neg1 is \_integer1,
\%; \_neg1 is unified with -100.

\_neg2 is \_real1
\%; \_neg2 is unified with -2.100000000000000e+00.

\_neg3 is \_neg1,
\%; \_neg3 is unified with 100.

\_neg4 is \_neg2
\%; \_neg4 is unified with 2.100000000000000e+00.

**See Also**

None.
– (subtraction)

Subtraction operator.

**Synopsis**

_number1 – _number2

**Description**

Operands can be integers or real numbers. If one or more of the operands is a real number, the result returned is a real number.

**Arguments**

_number1

The number to subtract from. Must be integer or real.

_number2

The number to subtract. Must be integer or real.

**Examples**

The following example shows various uses of the operator:

_integer1 is 100,
_integer2 is 30,
_integer3 is 10,
_real1 is 2.1,

% Assign values.

diff1 is _integer - _integer2,
% diff1 is unified with 70.

diff2 is _integer3 - _real1
% diff2 is unified with 7.900000000000000e+00.

**See Also**

None.
* 

Multiplication operator.

**Synopsis**

_\text{number1} \ast \text{number2}

**Description**

Operands can be integers or real numbers. If one or more of the operands is a real number, the result returned is a real number.

**Arguments**

_\text{number1}

A number. Must be integer or real.

_\text{number2}

A number. Must be integer or real.

**Examples**

The following example shows various uses of the operator:

_\text{integer1} \text{ is } 100,
_\text{integer2} \text{ is } 30,
_\text{real1} \text{ is } 2.1,\
% Assign values.

_\text{product1} \text{ is } \text{integer1} \ast \text{integer2},
%_\text{product1} \text{ is unified with } 3000.

_\text{product2} \text{ is } \text{integer1} \ast \text{real1}
%_\text{product2} \text{ is unified with } 2.100000000000000e+02.

**See Also**

None.
Real division operator.

**Synopsis**

_\text{number1} / \text{number2}

**Description**

Operands can be integers or real numbers. If one or more of the operands is a real number, the result returned is a real number.

**Arguments**

_\text{number1}

The dividend. Must be an integer or real number.

_\text{number2}

The divisor. Must be an integer or real number.

**Examples**

The following example shows various uses of the operator:

_\text{integer1} is 100,
_\text{integer2} is 30,
_\text{real1} is 2.1,
_\text{real2} is 10.5,
\% Assign values.

_\text{quotient1} is \text{real2} / \text{real1},
\% _\text{quotient1} is unified with 5.000000000000000e+00.

_\text{quotient2} is \text{integer1} / \text{integer2}
\% _\text{quotient2} is unified with 3.

**See Also**

None.
Integer division operator.

**Synopsis**

```
_integer1 // _integer2
```

**Description**

Operands must be integers only.

**Arguments**

- `_integer1`
  
The dividend. Must be an integer.
- `_integer2`
  
The divisor. Must be an integer.

**Examples**

The following example shows various uses of the operator:

- `_integer1` is 100,
- `_integer2` is 30,
- `_integer3` is 10,
  
% Assign values.

  ```
  _quotient1 is _integer1 // _integer3,
  % _quotient1 is unified with 10.
  
  _quotient2 is _integer1 // _integer2
  % _quotient2 is unified with 3.
  ```

**See Also**

None.
@<
Tests whether a term alphabetically precedes another.

**Synopsis**
_term1 @< _term2

**Description**
Succeeds if _term1 is alphabetically less than (precedes) _term2.

**Arguments**
_term1  Any Prolog term.
_term2  Any Prolog term.

**Examples**
The following example shows various alphabetical tests:
'a' @< 'b',
% Succeeds.

'A' @< 'a',
% Succeeds.

12 @< 23.2,
% Succeeds.

'12' @< 12
% Succeeds.

**See Also**
None.
Tests whether a term is alphabetically equal or precedes another.

Synopsis

_term1 @=< _term2

Description

Succeeds if _term1 is alphabetically less than or equal to _term2.

Arguments

_term1 Any Prolog term.
_term2 Any Prolog term.

Examples

The following example shows various alphabetical tests:

12.0 @=< 12, % Succeeds.

'ADMINISTRATOR' @=< 'Administrator' % Succeeds.

pointertoint(_pointer1, 100), % _pointer1 unified with 0x64.

pointertoint(_pointer2, 200), % _pointer2 unified with 0xc8.

100 @=< 200, % Succeeds.

_pointer1 @=< _pointer2, % Succeeds.

100 @=< _pointer2, % Succeeds.

_pointer1 @=< 200 % Fails.

See Also

None.
Tests whether a term alphabetically follows another.

**Synopsis**

_\_term1 @> _\_term2

**Description**

Succeeds if _\_term1 is alphabetically greater than _\_term2.

**Arguments**

_\_term1  Any Prolog term.

_\_term2  Any Prolog term.

**Examples**

The following example shows various alphabetical tests:

'Administrator' @> 'ADMINISTRATOR',
% Succeeds.

'aab' @> 'aaa',
% Succeeds.

23.1 @> 10.2
% Succeeds.

**See Also**

None.
Tests whether a term is alphabetically equal or follows another.

Synopsis

_term1 @>= _term2

Description
Succeeds if _term1 is alphabetically greater than or equal to _term2.

Arguments

_term1  Any Prolog term.
_term2  Any Prolog term.

Examples
The following examples show various alphabetical tests:

'atom2' @>= 'atom1',
% Succeeds.

'Atom2' @>= 'atom1'
% Fails.

See Also
None.
\!==

Term inequality operator.

**Synopsis**

\_term1 \!== \_term2

**Description**

Equivalent to the negation of the == operator.

**Arguments**

\_term1  Any Prolog term.

\_term2  Any Prolog term.

**Examples**

The following example shows the unification of _hostname and _dup_hostname, and then testing _hostname for inequality.

\_hostname = 'tec.tivoli.com',
\_dup_hostname = 'tec.tivoli.com',
\% Assign values.

\_hostname \!== 'arrakis.tivoli.com'
\% Succeeds.

\_hostname \!== _dup_hostname
\% Fails.

**See Also**

==
+ (pointer offset addition)

Pointer offset addition operator.

Synopsis

_pointer + _integer

Description

One operand must be an integer and the other a pointer. The result is a pointer, so the =? operator must be used for returning results.

Arguments

_pointer

A pointer.

_integer

An integer.

Examples

The following example shows a Tivoli Enterprise Console rule fragment that:

1. Computes the number of seconds since the reception of a first duplicate event using the pointeroffset predicate and unifies that value with the _offset1 variable.

2. Computes the same result as in step[1] using pointer subtraction, and unifies the value with the _offset2 variable.

3. Computes the date of the of the most recently received event by using pointer offset addition to add the _offset2 variable to the reception date of the first duplicate.

```
rule: pointer_offset: {
  event: _event of_class _class
  where [   
    date_reception: _date_reception
    % _date_reception unified with 0x37695cd4.
  ],
  action: {
    first_duplicate(_event, event: _dup_event
      where [   
        status: outside ['CLOSED'],
        date_reception: _dup_date_reception
        % _dup_date_reception unified with 0x376958d4.
      ]
    ),
    pointeroffset(_dup_date_reception,
      _offset1,
      _date_reception,
      % 1. _offset1 is unified with 1024 (0x37695cd4 - 0x376958d4).
      _offset2 is _date_reception -
      _dup_date_reception,
      % 2. _offset2 unified with 1024 (0x37695cd4 - 0x376958d4).
      _originalDate =? _dup_date_reception +
      _offset2,
      % 3. _originalDate unifies with 0x37695cd4
      % (0x376958d4 + 1024).
  }
}
```

See Also

=?
Mathematical less than.

**Synopsis**

_arithExp1 < _arithExp2

**Description**

Succeeds if the evaluation of _arithExp1 is less than the evaluation of _arithExp2.

**Arguments**

_arithExp1

Any arithmetic expression. Must be instantiated.

_arithExp2

Any arithmetic expression. Must be instantiated.

**Examples**

The following example shows various uses of the operator:

```prolog
% Assign values.
_integer3 < _integer2,
% Succeeds.
\nint(_real1) < _real1,
% Succeeds.
_integer1 < _integer3
% Fails.
```

**See Also**

None.
Term unification operator.

**Synopsis**

\_

**Description**
The Prolog assignment operator. Succeeds if \_term1 and \_term2 can be unified with each other. Note that the = operator is not the same as the == operator, which tests for equality.

**Arguments**

\_term1  Any Prolog term.

\_term2  Any Prolog term.

**Examples**
The following example unifies the atom tec.tivoli.com with the variable _hostname:

\_hostname = 'tec.tivoli.com'

**See Also**

==, =?, is
Mathematical equal to.

**Synopsis**

_arithExp1 =:= _arithExp2

**Description**

Succeeds if the evaluation of _arithExp1 is equal to the evaluation of _arithExp2.

**Arguments**

_arithExp1

Any arithmetic expression. Must be instantiated.

_arityExp2

Any arithmetic expression. Must be instantiated.

**Examples**

_integer1 is 100,

% Assign value.

_integer1 =:= integer2 + 70,

% Succeeds, _integer2 is unified with 30.

_integer1 =:= real(integer1),

% Succeeds.

_integer1 =:= - _integer1

% Fails.

**See Also**

None.
Expression evaluation.

**Synopsis**

\[ _\text{result} =? _\text{function} \]

**Description**

Term \texttt{function} is evaluated and the result is unified with \texttt{result}. Use this operator when \texttt{function} is a computable function, such as those listed in “Functions” on page 352.

**Arguments**

- \texttt{function}
  A computable function.
- \texttt{result}
  Any Prolog term.

**Examples**

The following example shows that the substring function is evaluated and its result is placed in the variable \texttt{msg}:

\[ _\text{msg} =? \text{substring}(\_\text{msg}, \_\text{start}, \_\text{length}) \]

**See Also**
### Synopsis

`_arithExp1 \=\= _arithExp2`

### Description

Succeeds if the evaluation of `_arithExp1` is not equal to the evaluation of `_arithExp2`.

### Arguments

- `_arithExpr1`
  - Any arithmetic expression. Must be instantiated.
- `_arithExpr2`
  - Any arithmetic expression. Must be instantiated.

### Examples

The following example shows various uses of the operator. Note that the last line uses the sign reversal operator.

```prolog
_integer1 is 100,
\%^ Assign value.

_integer1 =\= _integer1 + 1,
\%^ Succeeds.

- _integer1 =\= -100
\%^ Fails.
```

### See Also

None.
Mathematical less than or equal to.

**Synopsis**

_\text{arithExp1} \leq \text{arithExp2}

**Description**

Succeeds if the evaluation of _\text{arithExp1} is less than or equal to the evaluation of _\text{arithExp2}.

**Arguments**

- _\text{arithExpr1}
  - Any arithmetic expression. Must be instantiated.
- _\text{arithExpr2}
  - Any arithmetic expression. Must be instantiated.

**Examples**

The following example shows various uses of the operator:

- _\text{integer1} is 100,
- _\text{real1} is 2.1,

% Assign values.

- _\text{integer1} \leq \text{integer1 + 1},
  % Succeeds.
- _\text{real1} \leq \text{integer1}
  % Succeeds.

**See Also**

None.
Term equality operator.

**Synopsis**

```
_term1 == _term2
```

**Description**
The Prolog equality operator. Succeeds if `_term1` is identical to `_term2`. No unification is performed. Note that the `==` operator is not the same as the `=` operator, which is for assignment.

**Arguments**

- `_term1` Any Prolog term.
- `_term2` Any Prolog term.

**Examples**
The following example shows the unification of `_hostname` and `_dup_hostname`, and then testing `_hostname` for equality.

```
hostname = 'tec.tivoli.com',
dup_hostname = 'tec.tivoli.com',
% Assign values.
hostname == dup_hostname,
% Succeeds.
hostname == 'arrakis.tivoli.com
% Fails.
```

**See Also**

`==`, `\`
Mathematical greater than.

**Synopsis**

`arithExp1 > arithExp2`

**Description**

Succeeds if the evaluation of `arithExp1` is greater than the evaluation of `arithExp2`.

**Arguments**

- `arithExpr1`
  Any arithmetic expression. Must be instantiated.

- `arithExpr2`
  Any arithmetic expression. Must be instantiated.

**Examples**

The following example shows various uses of the operator:

- `integer1` is 100,
  `integer2` is 30,
  
  ```
  integer1 > integer2,
  ```
  
  % Succeeds.

- `integer2 * 2 > integer1`,
  
  ```
  integer2 * 2 > integer1,
  ```
  
  % Succeeds.

- `integer1 // 10 > integer2`
  
  ```
  integer1 // 10 > integer2
  ```
  % Fails.

**See Also**

None.
Mathematical greater than or equal to.

**Synopsis**

`_arithExp1 >= _arithExp2`

**Description**

Succeeds if the evaluation of `_arithExp1` is greater than or equal to the evaluation of `_arithExp2`.

**Arguments**

- `_arithExpr1`
  - Any arithmetic expression. Must be instantiated.

- `_arithExpr2`
  - Any arithmetic expression. Must be instantiated.

**Examples**

The following example shows various uses of the operator:

```prolog
% Assign values.
_integer1 is 100,
_integer2 is 30,
_integer3 is 10,
% Succeeds.
_integer1 >= _integer3 + 90,
% Succeeds.
_integer2 >= _integer1 - _integer1,
% Fails.
_integer3 >= _integer2
```

**See Also**

None.
abolish

Remove all clauses from the knowledge base.

Synopsis
abolish(_name, _arity)

Description
Removes all clauses of the predicate _name with arity _arity from the knowledge base.

Arguments
_arity The number of arguments to the predicate _name. Must be an integer.
_name The name (functor) portion of a Prolog fact or rule. Must be an atom.

Examples
The following example shows a use of the predicate:
assert(unix_hosts(arrakis, support)),
% Assert the fact into the knowledge base.

unix_hosts(arrakis, support),
% Succeeds.

abolish(unix_hosts, 2),
% Removes all facts named unix_hosts
% with an arity of 2 from the knowledge base.

unix_hosts(arrakis, support)
% Fails, because this fact no longer exists
% in the knowledge base.

See Also
assert, retract
Append elements to a list.

**Synopsis**

`append(_list1, _list2, _list3)`

**Description**

The list `_list2` is appended to the end of list `_list1`, with the results unified in list `_list3`. If `_list3` is already instantiated, then either `_list1` or `_list2` can be free. Whichever list is free then becomes instantiated to the elements in `_list3` that are not in the instantiated list. If all three arguments are already instantiated, then the predicate succeeds only if `_list3` is already the concatenation of `_list1` and `_list2`.

**Arguments**

- `_list1` A list, either instantiated or free.
- `_list2` A list, either instantiated or free.
- `_list3` A list, either instantiated or free.

**Examples**

The following example shows various uses of the predicate:

```
_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
_ntHosts = ['charon', 'scary', 'mach5', 'callisto'],
% Assign values.
append(_unixHosts, _ntHosts, _allHosts),
% _allHosts is unified with the list
% [arrakis,rooster,dune,perro,charon,scary,mach5,callisto].
append(_unixHosts, _moreHosts, _allHosts)
% _moreHosts is unified with [charon,scary,mach5,callisto].
```

**See Also**

None.
ascii

Converts between an ASCII character and ASCII code.

Synopsis

ascii(_char, _asciicode)

Description

If _char is instantiated to a valid ASCII character, the ASCII code is unified with _asciicode. If _asciicode is instantiated to a valid ASCII code, the ASCII character is unified with _char. If both _char and _asciicode are instantiated, the predicate succeeds only if they represent the same ASCII character. At least one of the arguments must be instantiated to a valid value.

Arguments

_asciicode

An integer in the range of 0–255, either instantiated or free.

_char

A one character atom, either instantiated or free.

Examples

The following example shows various ASCII character and code conversions:

ascii(a, _code1),
% Succeeds and _code1 is unified with 97.

ascii('
', _code2),
% Succeeds and _code2 is unified with 10.

ascii(_char1, 65),
% Succeeds and _char1 is unified with 'A'.

ascii(_char2, 13)
% Succeeds and _char2 is unified with '\r'.

See Also

None.
assert

Add a clause to the knowledge base at runtime.

**Synopsis**

assert(_clause)

**Description**

The clause _clause is asserted into the knowledge base.

It is recommended that this method for adding terms to the knowledge base be used if there are only a few terms to add, or if it is not known until runtime which terms are to be added. Otherwise, it is more efficient to compile the Prolog source file and include it in the rule base. See “Making your predicates and facts available to the Tivoli Enterprise Console rule engine” on page 442 for additional information.

**Arguments**

Clause _clause Any Prolog fact or rule.

**Examples**

1. The following example shows various uses of the predicate:

   ```prolog
   assert( unix_hosts(arrakis, support) ),
   assert( unix_hosts(io, manufacturing) ),
   assert( unix_hosts(tycho, development) ),
   assert( unix_hosts(acme, training) ),
   % Assert the facts named unix_hosts into
   % the knowledge base.
   %
   % Now look up the following facts.
   unix_hosts(arrakis, support),
   % Succeeds.
   unix_hosts(tycho, _tycho_area),
   % Succeeds and unifies _tycho_area with 'development'.
   unix_hosts(cerberus, _cerberus_area)
   % Fails because a fact named unix_hosts for
   % 'cerberus' does not exist in the knowledge base.
   ``

2. Any Prolog predicate can be asserted at event server start-up time and subsequently run from a Tivoli Enterprise Console rule action. IBM recommends that you assert the predicate when a TEC_Start event is received. The following example shows:

   a. A Tivoli Enterprise Console rule that asserts a predicate named my_string_match at event server start-up time.

   ```prolog
   rule: boot_string_match:
   (event:_start of_class within ['TEC_Start']
   where [ ],
   reception_action: action0:
     assert((my_string_match(_string, _substring, _left, _right) :-
     % my_string_match takes _string and _substring
     % as input and binds _left and _right such that
     % _left + _substring + _right = _string
   ```

   b. The Prolog interpreter input and output to test the asserted predicate. See “Using the Prolog interpreter” on page 443 for additional information.

   c. A Tivoli Enterprise Console rule fragment that uses the asserted predicate in an action.
atomlength(_substring, _sublen),
atompart(_string, _substring, _startsub, _len),
_lenl is _startsub - 1,
atompart(_string, _left, 1, _lenl),
_startr is _lenl + _sublen + 1,
atompart(_string, _right, _startr, _lenr),
!,
true))
).

The following example shows the results of testing when the `my_string_match` query is run from the Prolog interpreter. The second query shows how the `my_string_match` predicate can be applied repeatedly (twice in this case) to break up a complex string to get a substring. The variable values are printed after the end query character, which is a period.

?- my_string_match('foobarmoo', bar, L, R).
    L = foo
    R = moo
    Yes

?- my_string_match('Interface xyz is Up', ' ', L, R),
   my_string_match(R, ' ', _interface, R1).
    L = Interface
    R = xyz is Up
    _interface = xyz
    R1 = is Up
    Yes

If you have an original string like `Interface name is Up` and want the interface name embedded in a message, the following example Tivoli Enterprise Console rule fragment isolates the name so you can create a message later in the action shown or in a subsequent action within the rule:

```
rule: plain_rule1:
  (event: _ev of_class within ['Some_Class']
   where [ msg: _msg ] ,
   reception_action: action0: ( 
     my_string_match(_msg, ' ', L, R),
     my_string_match(R, ' ', _interface, R1),
     % Isolate interface name. 
   )
  ).
```

See Also
**compile**
atom

Tests whether a variable is of type atom.

Synopsis
atom(_term)

Description
Succeeds if _term is an atom.

Arguments
_term Any variable, either instantiated or free.

Examples
The following example shows various tests for a type atom:

_atom = 'An atom',
\% Assign value.
atom(_atom),
\% Succeeds.

_atom = 10
\% Assign value.
atom(_atom)
\% Fails because the variable is an integer.

See Also
None.
atomic

Tests whether a variable is of type atomic.

Synopsis

atomic(_term)

Description

Succeeds if _term is either a real number, integer, atom, or pointer.

Arguments

_term  Any variable, either instantiated or free.

Examples

The following example shows various tests for a type atomic:

_integer = 200,
_real = 2.345,
_atom = 'An atom',
_list = [element1, element2, element3],
% Assign values.

atomic(_integer),
% Succeeds.
atomic(_real),
% Succeeds.
atomic(_atom),
% Succeeds.
atomic(_list)
% Fails.

See Also

None.
atomlength

Determine the length of an atom.

Synopsis
atomlength(_atom, _length)

Description
The length of the value instantiated in _atom is unified with _length. If both _atom and _length are instantiated, then this predicate simply succeeds if _atom is of length _length.

Arguments

_atom  The atom whose length to obtain. Must be an atom.

_length  The length for _atom. Must be an integer.

Examples
The following example show various uses of the predicate:

atomlength('This is an atom', _length),
\% _length is unified with 15.

atomlength('atom', 4)
\% Succeeds.

See Also
None.
atompart

Get a substring from an atom.

**Synopsis**

atompart(_atom, _atomPart, _startPos, _length)

**Description**

The atom _atomPart is the substring obtained from _atom, which started at position _startPos in _atom with a length of _length. The first character of the atom is at position 1.

If _atomPart is free, it is instantiated to the part of _atom specified by _startPos and _length. If the _startPos and _length arguments are not specified, the default values of 1 and the remaining length of _atom, respectively, are used.

If _atomPart is instantiated and _startPos is free, _startPos is instantiated to the starting position of the first occurrence of _atomPart in _atom.

**Arguments**

_**atom**_ The original string. Must be an atom.

_**atomPart**_ A substring in _atom. Must be an atom.

_**length**_ The length of the substring _atomPart. Must be an integer.

_**startPos**_ The starting position of the substring _atomPart in _atom. Must be an integer.

**Examples**

The following example shows how to:

1. Determine the index where the first space is located and then use that value to compute the length of the first word (the host name).
2. Get a substring from the _msg variable starting at position 1 (that is, the first character of _msg) and of the computed length.

```
atompart(_msg, ' ', _startOfSpace, _lenOfSpace),
% If an event were received where the msg attribute contained the string 'pikes-peak is down',
% then _startOfSpace is unified with 11,
% and _lenOfSpace is unified with 1.

_lenOfHost is _startOfSpace - 1,
% _lenOfHost is unified with 10.

atompart(_msg, _host, 1, _lenOfHost)
% _host is unified with 'pikes-peak'.
```

**See Also**

substring
atomconcat

Concatenate atoms.

Synopsis

atomconcat(ListOfAtomics, _concatAtom)

—OR—

atomconcat(_atom1, _atom2, _concatAtom)

Description

Variable _concatAtom is the atom constructed by concatenating all items from the list _ListOfAtomics in the order of the list, or by concatenating the instantiated values of _atom1 and _atom2. The values in _ListOfAtomics, _atom1, and _atom2 can be of type atom, integer, real, or pointer.

In the second form of the predicate with three arguments, only one argument can be free at the most; the free argument is unified with the result.

Values that are not atoms are automatically converted to atoms before concatenation.

Arguments

_atom1 An atom, integer, real, or pointer that _atom2 will concatenate to.
_atom2 An atom, integer, real, or pointer that will concatenate to _atom1.
_concatAtom The results of concatenating the arguments. This value is an atom.
_listOfAtomics A list of atoms, integers, real numbers, or pointers to concatenate in the order of the list.

Examples

1. The following example shows using atomconcat with two arguments:

   _hostname = 'shinai',
   _repeat_count is 5,
   % Assign values.

   atomconcat([['Host ', _hostname, ' unavailable ',
               _repeat_count, ' times.'], _new_msg])
   % The variable _new_msg is unified with the
   % atom 'Host shinai unavailable 5 times.'

2. The following example shows various uses of atomconcat with three arguments:

   _hostname = 'shinai',
   % Assign value.

   atomconcat(_hostname, ' is down.', _down_msg),
   % _down_msg is unified with 'shinai is down.'.

   _msg = 'shinai is back up.',
   % Assign value.

   atomconcat(_up_hostname, ' is back up.', _msg)
   % _up_hostname is unified with 'shinai'.

See Also

None.
atomtolist

Convert between the atomic type and a list of characters.

**Synopsis**

`atomtolist(atomic, _listOfChar)`

**Description**

If `atomic` is instantiated, a list of atoms is unified with `_listOfChar`.

If `_listOfChar` is instantiated, an atom is unified with `atomic`.

At least one of the arguments must not be instantiated.

When a list is transformed into an atom, its type is determined by the elements in the list. The resulting type can be:

- A pointer, if the list starts with `0x`
- An integer, if all the elements of `_listOfChar` are numerical characters in the range 0–9
- A real, if the list is a combination of numeric characters and a decimal point
- An atom in all other cases

**Arguments**

`atomic`

An atomic value, either instantiated or free.

`_listOfChar`

A list of atoms, either instantiated or free.

**Examples**

The following example shows various conversions using the predicate:

```
atomtolist(hexNumber, ['0', 'x', '1', '3', '2']),
% Unifies _hexNumber with 0x132.

atomtolist(_integer, ['2', '3', '4', '5']),
% Instantiates _integer to 2345.

atomtolist(_realNumber, ['1', '0', '.', '3', '2']),
% Instantiates _realNumber to 1.032000000000000e+01.

atomtolist(prolog, _list),
% Instantiates _list to [p,r,o,l,o,g].

atomtolist(_atom, _list)
% Instantiates _atom to 'prolog'.
```

**See Also**

None.
**compile**

Compile a Prolog source file.

**Synopsis**

```
compile(_fileName)
```

**Description**

The Prolog source code in the file specified with `_fileName` is compiled into a Prolog object file that can later be asserted into the knowledge base with the `consult` or `reconsult` predicate. The Prolog source file (filename.pro) is compiled into a Prolog object file named `filename.wic` in the same directory as the source file.

The contents of the Prolog source file must be Prolog predicates and facts. These predicates and facts are the same as those used with the `assert` predicate. Each predicate and fact must be terminated with a period.

Before using the compile predicate, you must set the value of the BIM_PROLOG_DIR environment variable to the value of $BINDIR/TME/TEC in the environment for the Tivoli Enterprise Console event server’s object dispatcher. The following steps describe how to do this:

1. From a shell command line, redirect the output from the following `odadmin` `environ` `get` command into a file. The following example shows how to do this:
   ```
   odadmin environ get > /tmp/oserv.env
   ```
2. Using a text editor, add the BIM_PROLOG_DIR environment variable and set its value to `$BINDIR/TME/TEC`, where `$BINDIR` is the actual value of the $BINDIR environment variable.
3. From a shell command line, reset the object dispatcher environment to the information contained in the newly created and modified temporary file (`/tmp/oserv.env` in this example). The following example shows how to do this:
   ```
   odadmin environ set < /tmp/oserv.env
   ```

See the *Tivoli Management Framework Reference Manual* for additional information about the `odadmin` command.

Alternatively, Prolog source files can be compiled from the command line of a shell or with the Prolog interpreter. These methods of adding facts and rules to the knowledge base are recommended because they are less of a performance impact to the rule engine than using the compile or assert predicates. See “Making your predicates and facts available to the Tivoli Enterprise Console rule engine” on page 442 for additional information.

**Arguments**

_fileName_

The file name of the Prolog source file. You can specify a path, also. Must be an atom.

**Examples**

The following example uses a Prolog source file named unix_hosts.pro in the `/var/prolog` directory. The compile predicate compiles it into an object file named `unix_hosts.wic` in the same directory. You can then use the consult predicate to load this Prolog object file into the rule engine.
unix_hosts(arrakis, support).
unix_hosts(odin, support).
% Contents of /var/prolog/unix_hosts.pro

compile('/var/prolog/unix_hosts')
% Creates /var/prolog/unix_hosts.wic.

See Also
consult, reconsult
**consult**

Load a compiled Prolog file into the knowledge base.

**Synopsis**

`consult(_fileName)`

**Description**

The compiled file `_fileName` of Prolog clauses is loaded and asserted in the knowledge base. The file of clauses being consulted must have been compiled using the `compile` predicate, the `TECcomp` command, or the Prolog interpreter. A compiled Prolog file is also called a Prolog object file.

**Note:** Do not consult a Prolog object file more than once. Otherwise, multiple occurrences of each fact and rule will exist in the knowledge base. If a file must be consulted more than once, use the `reconsult` predicate.

**Arguments**

`_fileName`

The file name of the Prolog object file. Do not specify the `.wic` file name extension. Must be an atom.

**Examples**

The following example shows the use of this predicate:

```prolog
unix_hosts(arrakis, support).
unix_hosts(odin, support).
% Contents of /var/prolog/tec_r.unix_hosts.pro
compile('/var/prolog/tec_r.unix_hosts')
% Creates /var/prolog/unix_hosts.wic.
consult('/var/prolog/unix_hosts'),
% Load and assert the clauses in the compiled
% unix_hosts.wic file.
unix_hosts(arrakis, support),
% Query of the knowledge base succeeds, this clause exists.
unix_hosts(cerberus, development)
% Query of the knowledge base fails, this clause
% does not exist.
```

**See Also**

assert, compile, reconsult
delete

Delete elements from a list.

**Synopsis**

delete(list, element, residue)

**Description**
List `residue` is instantiated to list `list`, in which all elements equal to `element` are deleted.

If `list` and `residue` are instantiated to lists, then `element` is unified with the element in `list` that is not in `residue`.

**Arguments**

- **element**
  An atomic value, either instantiated or free.

- **list**
  A list. Must be instantiated.

- **residue**
  A list, either instantiated or free.

**Examples**
The following example shows various uses of the predicate:

```prolog
_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
 ntHosts = ['charon', 'scary', 'mach5', 'callisto'],
 % Assign values.

append(_unixHosts, _ntHosts, _allHosts),
 % _allHosts is unified with the list
 % ['arrakis', 'rooster', 'dune', 'perro', 'charon',
 % 'scary', 'mach5', 'callisto'].

delete(_allHosts, 'arrakis', _residue),
 % _residue is unified with the list
 % ['rooster', 'dune', 'perro', 'charon', 'scary', 'mach5', 'callisto'].

delete(_allHosts, _host, _residue)
 % _host is unified with 'arrakis'.
```

**See Also**
None.
disjoint

Compare elements in two lists for uncommon values.

**Synopsis**

\[
\text{disjoint}(_\text{list1}, _\text{list2})
\]

**Description**

Succeeds if \(_\text{list1}\) and \(_\text{list2}\) do not have any elements in common.

**Arguments**

\(_\text{list1}\quad\text{Any list.}\)

\(_\text{list2}\quad\text{Any list.}\)

**Examples**

The following example shows a successful comparison of two lists that do not have any elements in common:

\[
\begin{align*}
_\text{unixHosts} &= ['\text{arrakis}', '\text{rooster}', '\text{dune}', '\text{perro}'], \\
_\text{ntHosts} &= ['\text{charon}', '\text{scary}', '\text{mach5}', '\text{callisto}'], \\
\%
\text{Assign values.}
\end{align*}
\]

\[
\text{disjoint}(_\text{unixHosts}, _\text{ntHosts}) \\
\%
\text{Succeeds.}
\]

**See Also**

[intersect]
empty_list

Test if a list is empty.

Synopsis
empty_list(_list)

Description
Succeeds if _list is an empty list.

To test if a list is not empty, the not predicate can be used to reverse the test.

Arguments

_list  A list, either instantiated or free.

Examples

1. The following example shows the list _emptyList is instantiated to an empty list, so the test succeeds. The test fails, however, when the list _list is tested because it isn’t empty.

   _list = [element1, element2, element3],  
   _emptyList = [ ],  
   % Assign values.

   empty_list(_emptyList),  
   % Succeeds.

   empty_list(_list)  
   % Fails

2. The following example shows the not predicate can be used in this test to reverse the result. If you want to perform this same test on the variable _list, but have the predicate succeed when the variable has elements in the list, then apply the not predicate as follows:

   not empty_list(_list)  
   % Succeeds. List contains no elements,  
   % but the result is reversed.

See Also

not
fclose

Close an open file.

**Synopsis**

fclose(_file)

**Description**

Closes the file specified by _file.

**Arguments**

_file The file pointer obtained from opening the file with the fopen predicate.

**Examples**

The following example Tivoli Enterprise Console rule fragment shows how to close a Prolog source file that had been previously opened for reading. Closing the file in a separate action ensures the fclose predicate runs regardless of the success or failure of any actions between the open_file and close_file actions.

```prolog
action: open_file: (  
fopen(_fp, '/var/prolog/nt_hosts', 'r'),
  
action: close_file: (  
  fclose(_fp)
  )
```

**See Also**

fopen
**flisting**

Write predicates to an open file.

**Synopsis**

```plaintext
flisting(_file, _predName)
```

**Description**

All clauses of predicate _predName in the knowledge base are written to the current output stream. The current output stream is a text file, the file can be compiled later with the compile predicate.

**Arguments**

- `_file` A variable that holds the file pointer obtained from the fopen predicate.
- `_predName` The name of the predicate to write.

**Examples**

The following example Tivoli Enterprise Console rule fragment shows how to write all clauses of the predicate unix_hosts to the file /tmp/maint_mode.pro.

```plaintext
action: write_predicates: (  
  fopen(_fp, 'maint_mode.pro',w),  
  flisting(_fp, 'unix_hosts')  
  fclose(_fp)  
),
```

**See Also**

`fclose` `fopen`
**fopen**

Open a file.

**Synopsis**

```prolog
fopen(_file, _physFileName, _mode)
```

**Description**

Opens the physical file whose name is instantiated in `_physFileName` for reading, writing, or appending, depending on the `_mode` argument. `_physFileName` is associated with the file pointer `_file`, which must be free. This pointer is used in all subsequent I/O operations for this file. If the file cannot be opened, the fopen predicate fails.

**Arguments**

- `_file` The file pointer. Must be free.
- `_mode` The mode for opening the file. The mode is specified as follows and must be an atom:
  - `a` Open the file for appending.
  - `r` Open the file for reading.
  - `w` Open the file for writing.
- `_physFileName` The name of a physical file. Must be an atom.

**Examples**

The following example Tivoli Enterprise Console rule fragment shows how to open a Prolog source file for reading:

```prolog
action: open_and_read: (  
  fopen(_fp, '/var/prolog/nt_hosts', 'r'),
),
```

**See Also**

`fclose`
ground

Tests whether a variable is instantiated.

Synopsis

ground(_term)

Description

Succeeds if _term is instantiated. To test if a variable is not instantiated, the not
predicate can be used to reverse this test.

Arguments

_term   Any variable, either instantiated or free.

Examples

1. In the following example, the variable _someVariable is instantiated and unified
   with the atom some_String. The ground predicate tests whether the variable
   has been instantiated. The variable _someOtherVariable has not been
   instantiated, so that test fails.

   _someVariable = 'some_String',
   % Succeeds.
   ground(_someVariable),
   % Succeeds. Variable is instantiated to some_String.
   ground(_someOtherVariable)
   % Fails. Variable is not instantiated.

2. The not predicate can be applied to this test to reverse the result. If you want
   to perform this same test on the variable _someOtherVariable but have the
   predicate succeed when the variable has not been instantiated, then apply the
   not predicate as follows:

   not ground(_someOtherVariable)
   % Succeeds. Variable is not instantiated, but the
   % result is reversed.

See Also

not
**int_to_hex**

Hexadecimal string representation of an integer.

**Synopsis**

\[
\text{int_to_hex}(_\text{integer})
\]

—OR—

\[
\text{int_to_hex}(_\text{integer}, _\text{width})
\]

**Description**

Converts the integer value instantiated in \texttt{_integer} to an atom representation of the hexadecimal value. The \texttt{_width} argument specifies the field width of the result, in which case the value of \texttt{_atom} is left-padded with zeros if necessary.

**Arguments**

\texttt{_integer}

An integer, either instantiated or free.

\texttt{_width} An integer. Specifies the field width for \texttt{_atom}.

**Examples**

The following example shows various conversions from integer to hexadecimal:

\%
Converting an integer to hexadecimal.
\%
\begin{verbatim}
_hexstring1 =\? \text{int_to_hex}(100)
\%
_hexstring1 is unified with '64'.
\%
Converting an integer to hexadecimal with a specific field width.
\%
\begin{verbatim}
_hexstring2 =\? \text{int_to_hex}(100, 8)
\%
_hexstring2 is unified with '00000064'.
\end{verbatim}
\%

**See Also**

None.
integer
Tests whether a variable is of type integer.

Synopsis
integer(_term)

Description
Succeeds if _term is an integer.

Arguments
_term Any variable, either instantiated or free.

Examples
The following example shows various tests for a type integer:
_int = 200,
% Assign value.

integer(int),
% Succeeds.

_int = 1.234,

integer(_integer)
% Fails because the variable is a real number.

See Also
None.
intersect

Compare elements in lists for common values.

Synopsis

intersect(_list1, _list2)

—OR—

intersect(_list1, _list2, _intersection)

Description

The first form succeeds if the lists _list1 and _list2 have an element in common.

In the second form, _intersection is a list instantiated to the intersection of lists _list1 and _list2.

Arguments

_intersection

The intersection of elements common to _list1 and _list2. A list.

_list1 A list.

_list2 A list.

Examples

1. The following example shows the first form (two arguments) of the predicate:

   unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
   % Assign value.
   intersect(unixHosts, ['perro', 'dune'])
   % Succeeds.

2. The following example shows the second form (three arguments) of the predicate:

   unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
   % Assign value.
   intersect(unixHosts, ['perro', 'dune', 'odin'], _intersection)
   % _intersection is unified with ['dune', 'perro'].

See Also

None.
inttoatom

Converts between an integer and an atom representation of an integer.

Synopsis

inttoatom(_integer, _atom)

Description

If _integer is instantiated to an integer, the atom representation of the integer is unified with _atom.

If _atom is instantiated to an atom representing an integer, it is converted to an integer type and is unified with _integer.

At least one of the arguments must be instantiated to a valid value.

Arguments

.atom An atom, either instantiated or free.

.integer An integer, either instantiated or free.

Examples

The following example shows various integer and atom conversions:

inttoatom(100, _atom1),
% Succeeds and _atom1 is unified with '100'.

inttoatom(_int1, '123')
% Succeeds and _int1 is unified with 123.

See Also

None.
is

Mathematical unification.

Synopsis

_result is _arithExpr

Description

The arithmetic expression _arithExpr is evaluated and the result (integer or real) is unified with _result. _arithExpr must be an arithmetic expression of integer, real, or atom constants, and computable functions.

Arguments

_arithExpr

Any arithmetic expression.

_result

Any integer or real number result.

Examples

The following example shows various uses of the operator:

_integer is 10
½ _integer is unified with 10.

_result is _integer + 90
½ _result is unified with 100.

See Also

=?

Appendix A. Using Prolog in rules  405
is_list

Tests whether a variable is a list.

Synopsis

is_list(_list)

Description

Succeeds if _list is a type list. _list can contain elements or it can be an empty list.

Arguments

_list A Prolog list, either containing elements or empty.

Examples

The following example shows various tests for a type list:

_list = [element1, element2, element3],
_emptylist = []
‰ Assign values.

is_list(_list),
‰ Succeeds

is_list(_emptylist)
‰ Succeeds

See Also

None.
**length**

Get the length of a list.

**Synopsis**

`length(_list, _length)`

**Description**

The number of elements in `_list` is instantiated in `_length`.

**Arguments**

- `_length`
  
  The length of the list `_list`. An integer.

- `_list`
  
  The list from which to obtain the length. Must be instantiated.

**Examples**

The following example shows how to get the length of the list `_unixHosts`:

```prolog
_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
% Assign value.
length(_unixHosts, _listLength)
% Unifies _listLength with 4.
```

**See Also**

None.
lowertoupper

Convert between lowercase and uppercase letters in an atom.

**Synopsis**

lowertoupper(_lowercase, _uppercase)

**Description**

If _lowercase is free, it is instantiated to the lowercase value of _uppercase.

If _uppercase is free, it is instantiated to the uppercase value of _lowercase.

One of the arguments must be instantiated and the other free.

**Arguments**

_lowercase

The lowercase value. Must be an atom, either instantiated or free.

_uppercase

The uppercase value. Must be an atom, either instantiated or free.

**Examples**

The following example shows various uses of the predicate:

```prolog
_user = 'Administrator',
%Assign value.
lowertoupper(_user, _uppercase),
% _uppercase is unified with 'ADMINISTRATOR'.
lowertoupper(_lowercase, _uppercase)
% _lowercase is unified with 'administrator'.
```

**See Also**

None.
member

Check a list for a value.

Synopsis

member(_element, _list)

Description

Standard membership predicate. Succeeds if element _element is in list _list.

To check if an element is not in the list, the not predicate can be used to reverse the check.

If _element is free, all of the successive elements in the list are returned by backtracking.

Arguments

_element
The element for which to check membership. Any term.

_list
The list from which to check for membership of the element.

Examples

1. The following example checks whether the atom arrakis is in the list _unixHosts:

   _unixHosts = ['arrakis', 'rooster','dune', 'perro'],
   % Assign value.

   member('arrakis', _unixHosts)
   % Succeeds.

   member('odin', _unixHosts)
   % Fails.

2. The following example checks if some element is not in the list, using the not predicate to reverse the test. The following example checks whether the atom odin is not in the list _unixHosts:

   _unixHosts = ['arrakis', 'rooster','dune', 'perro'],
   % Assign value.

   not member('odin', _unixHosts)
   % Succeeds. List does not contain the % element, but the result is reversed.

See Also

not
name

Convert between the atomic type and a list of character codes.

Synopsis
name(_atomic, _listOfAsciiCodes)

Description
If _atomic is instantiated, a list of ASCII codes is unified with _listOfAsciiCodes.

If _listOfAsciiCodes is instantiated, an atom is unified with _atomic. At least one of
the arguments must not be instantiated. When a list is transformed into an atomic
value, its type is determined by the elements in the list. The resulting type can be:

- A pointer, if the list starts with the ASCII codes for 0x
- An integer, if all the elements of _listOfAsciiCodes are the ASCII codes for
  numbers in the range 0–9
- A real, if the list is a combination of ASCII codes for numbers and a decimal
  point
- An atom in all other cases

Arguments

_atomic
An atomic value, either instantiated or free.

_listOfAsciiCodes
A list of integer values.

Examples
The following example shows various conversions using the predicate:
name(0x132, _hexList),
% Instantiates _hexList to [48,120,49,51,50].

name(_hexNumber, _hexList),
% Instantiates _hexNumber to 0x132.

name(2345, _integerList),
% Instantiates _integerList to [50,51,52,53].

name(_integer, _integerList),
% Instantiates _integer to 2345.

name(10.32, _realList),
% Instantiates _realList to 10.32.

name(_realNumber, _realList),
% Instantiates _realNumber to 10.32.

name(prolog, _list),
% Instantiates _list to 'prolog'.

See Also
None.
nmember

Check a list for a single value at an index.

Synopsis

nmember(_element, _list, _index)

Description

Succeeds when _element is at the index specified by _index in the list _list. The index of the list begins at 1.

If _element and _list are instantiated, the index of _element is unified with _index. If _list and _index are instantiated, _element is unified with the element of _list at index _index.

Arguments

_element
  An atomic value, either instantiated or free.

_index  An integer, either instantiated or free.

_list  A list, must be instantiated.

Examples

The following example shows uses of the predicate:

_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
% Assign value.
nmember('rooster', _unixHosts, _index),
% _index is unified with 2.
nmember(_element, _unixHosts, 3)
% _element is unified with 'dune'.

See Also

nmembers
nmembers

Check a list for multiple values at indexes.

**Synopsis**

```prolog
nmembers(_indexes, _list, _listOfElements)
```

**Description**

This predicate is similar to nmember, except it uses lists to specify multiple index and element argument values. Both `_indexes` and `_listOfElements` must not be free; that is, one must be instantiated.

**Arguments**

- `_indexes`  
  A list, either instantiated or free.
- `_list`  
  A list, must be instantiated.
- `_listOfElements`  
  A list, either instantiated or free.

**Examples**

The following example shows various uses of the predicate:

```prolog
_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
% Assign value.

nmembers(_indexes, _unixHosts, ['arrakis', 'dune']),  
% _indexes is unified with the list [1,3].

nmembers([2, 4], _unixHosts, _listOfElements)  
% _listOfElements is unified with the list [rooster, perro].
```

**See Also**

`nmember`
**not**

Term negation predicate.

**Synopsis**

`not _term`

**Description**

This predicate succeeds if `_term` fails, otherwise it fails if `_term` succeeds.

**Arguments**

 `_term`  Any Prolog term.

**Examples**

The following example shows the integer predicate succeeds if the argument is an integer, and fails otherwise. The `not` predicate, shown in the last example, can be applied to test for the opposite condition.

```
integer(100),
% Succeeds.
```

```
integer(1.234)
% Fails because this is a real number, not an integer.
```

```
not integer(1.234),
% Succeeds, because `integer(1.234)` would fail.
```

**See Also**

None.
**number**

Tests whether a variable is of type number.

**Synopsis**

```plaintext
number(_term)
```

**Description**

Succeeds if `_term` is either a real number or an integer.

**Arguments**

`_term` Any variable, either instantiated or free.

**Examples**

The following example shows various tests for a type of number:

```plaintext
_integer = 200,
_real = 2.345,
% Assign values.

call number(_integer),
% Succeeds.

call number(_real)
% Succeeds.
```

**See Also**

None.
pointer

Tests whether a variable is of type pointer.

Synopsis

pointer(_term)

Description

Succeeds if _term is a pointer.

Arguments

_term Any variable, either instantiated or free.

Examples

The following example Tivoli Enterprise Console rule shows that the value of the date_reception event attribute is assigned to a variable of type pointer when an event is received:

```prolog
rule: pointer:
   event: _event of_class _class
   where [date_reception: _date_reception],
   reception_action:
      pointer(_date_reception) % Succeeds
).
```

See Also

None.
pointeroffset
Get the difference between two pointer values.

Synopsis
pointeroffset(_pointer1, _offset, _pointer2)

Description
Pointer _pointer1, adjusted with offset _offset, is pointer _pointer2. One of the arguments can be free at most. The offset can be positive or negative. _offset must be an integer. When used to get the difference between two time values, the resulting difference is the number of seconds between the two time values.

Arguments
_offset  The difference between _pointer1 and _pointer2. Must be an integer.
_pointer1  A pointer.
_pointer2  A pointer.

Examples
The following example Tivoli Enterprise Console rule fragment shows how to compute the number of seconds since the reception of the first duplicate event of an event class and unify that value with the _offset1 variable:
rule: pointer_offset: {
  event: _event of_class _class
  where [
    date_reception: _date_reception
    % _date_reception unified with 0x37695cd4.
  ],
  action: {
    first_duplicate(_event, event: _dup_event
    where [
      status: outside ['CLOSED'],
      date_reception: _dup_date_reception
      % _dup_date_reception unified with 0x376958d4.
    ],
    pointeroffset(_dup_date_reception,
      _offset1,
      _date_reception),
    % _offset1 is unified with 1024 (0x37695cd4 -
    % 0x376958d4).
  }
}

See Also
None.
pointertoatom

Converts between a pointer and an atom representation of a pointer.

Synopsis
pointertoatom(_pointer, _atom)

Description
If _pointer is instantiated to a pointer value, the atom representation is unified with _atom. If _atom is instantiated to an atom representation of a pointer value, it is converted to a pointer and is unified with _pointer. At least one of the arguments must be instantiated to a valid value.

Arguments
_atom An atom, either instantiated or free.

_pointer A pointer, either instantiated or free.

Examples
The following example Tivoli Enterprise Console rule shows various pointer and atom conversions:

rule: pointertoatom: {
  event: _event of_class _class
  where [
    date_reception: _date_reception
  ],
  reception_action: (
    pointertoatom(_date_reception, _atom),
    % _Atom is unified with '0x3745d5ed'.
    pointertoatom(_pointer, '0x1234')
    % _Pointer is unified with the hexadecimal number 0x1234.
  )
}.

See Also
None.
**pointertoint**

Converts between a pointer and an integer representation of a pointer.

**Synopsis**

`pointertoint(_pointer, _integer)`

**Description**

If `_pointer` is instantiated to a pointer value, the integer representation is unified with `_integer`.

If `_integer` is instantiated to an integer, it is converted to a pointer type and is unified with `_pointer`. If the value of `_pointer` is outside the range of integers, it is truncated.

At least one of the arguments must be instantiated to a valid value.

**Arguments**

- **_integer**
  
  An integer, either instantiated or free.

- **_pointer**
  
  A pointer, either instantiated or free.

**Examples**

The following example Tivoli Enterprise Console rule shows various pointer and integer conversions:

```plaintext
rule: pointertoint: (  
  event: _event of_class _class  
  where [  
    date_reception: _date_reception  
  ],  
  reception_action: (  
    pointertoint(_date_reception, _int_reception),  
    % If the date_reception slot were to  
    % contain the value 0x3745d5ed,  
    % this line would unify the variable  
    % _int_reception with -146418195. The  
    % actual integer value of 0x3745d5ed  
    % is 927323629 in decimal. However,  
    % this is larger than the maximum  
    % value of 268435455 (2^32-1) that an  
    % integer type can hold. So this  
    % number is the result of overflow.  
    pointertoint(_pointer, 2000)  
    % (_pointer is unified with 0x7d0.  
    )  
  ).
```

**See Also**

None.
printf
Write formatted output to a file.

Synopsis
printf(_file, _format, _value)

Description
Writes to a file specified by the _file argument, with the format specified in _format. The format specification is similar to that in the fprintf() function in the C programming language.

Arguments
_file  The file pointer obtained from opening the file with the fopen predicate.
_format  The format specification for the output. Must be an atom instantiated with a format specification from the following list:
%c   Character.
%d   Integer printed in decimal notation.
%e   Real printed in exponential notation.
%f   Real printed in decimal notation.
%g   Real printed in its shortest form (decimal or exponential notation).
%o   Integer printed in octal notation, without sign and leading zero.
%s   String.
%u   Integer printed in unsigned decimal notation.
%x   Integer printed in hexadecimal notation, without sign and leading 0x.

You can supply more detailed conversion specifications between the % sign and the conversion character, as follows:
–   Left adjustment.
0   Zero padding to the left.
n   In cases of an integer or a string, n is the minimum length of the field.
n.m   In cases of a real, n is the minimum length of the field and m indicates the number of digits after the decimal point.
_value  An atomic value or a list of atomic values to be formatted for output.

Examples
The following example Tivoli Enterprise Console rule fragment shows how to write formatted output to the /tmp/eventdata.txt file:
action: write_data: (  
fopen(_fp, '/tmp/eventdata.txt', a),  
  printf(_fp, 'Event of class %s was closed by %s', [_class, _administrator])  
),  
action: close_file: (  
fclose(_fp)  
)
See Also
None.
**read**

Read a Prolog term from an open Prolog source file.

**Synopsis**

\[ \text{read}(_\text{file}, _\text{term}) \]

**Description**

The term _term is unified with the next term read from the file _file. Characters are read from the input file until an end-point is found (a period character followed by an end-of-line character) or until an error is encountered. This predicate is for reading Prolog terms only. See "readln" on page 422 for reading arbitrary text data.

**Arguments**

_\text{file}  
The file pointer obtained from opening the file with the fopen predicate.

_\text{term}  
The Prolog term read from the file.

**Examples**

The following example Tivoli Enterprise Console rule fragment shows how to open a Prolog source file that contains Prolog terms and read a term into a variable. Assume this example reads from the file named /var/prolog/tec_r.nt_hosts, which contains the following clauses:

```
nt_hosts('hmckinne').
nt_hosts('scary').
```

The actions for the rule to read the first term from the file is shown following:

```
\begin{verbatim}
action: open_and_read: ( 
    fopen(_fp, '/var/prolog/tec_r.nt_hosts', 'r'),
    read(_fp, _term)
% _term is unified with nt_hosts(hmckinney).
),
action: close_file: ( 
    fclose(_fp)
)
\end{verbatim}
```

**See Also**

readln

Appendix A. Using Prolog in rules 421
**readln**

Read a line from an open file.

**Synopsis**

`readln(_file, _line)`

**Description**

The next text line read from the file `_file` is unified with `_line`. A text line ends at a newline character or at end-of-file. The newline character is discarded when read.

**Arguments**

- `_file` The file pointer obtained from opening the file with the fopen predicate.
- `_line` The line read from the file. The line is read as an atom.

**Examples**

The following example Tivoli Enterprise Console rule fragment shows how to read one line from the `/etc/hosts` file, which contains the following lines:

```
127.0.0.1   localhost
146.84.113.31 arrakis
```

The actions for the rule to read the first line from the file is shown following:

```
loghostaction: open_hosts: {
    fopen(_fp, '/etc/hosts', 'r'),
    readln(_fp, _hostsEntry) % _hostsEntry is unified with % '127.0.0.1\tlocalhost\t'.
},
action: close_hosts: { fclose(_fp)
}
```

**See Also**

`read`
real (convert integer)

Converting integer type to real type.

**Synopsis**

real(_integer)

**Description**

Converts _integer to a real number.

**Arguments**

_integer

the integer to convert.

**Examples**

The following example shows predicate usage:

```
real is real(1)
% _real is unified with 1.00000000000000e+00.
```

**See Also**

None.
real (test real type)
Tests whether a variable is of type real.

Synopsis
real(_term)

Description
Succeeds if _term is a real number.

Arguments
_term Any variable, either instantiated or free.

Examples
The following example shows various tests for a type of real:
_real = 2.345,
% Assign value.

real(_real),
% Succeeds.

_real = 123,

real(_real)
% Fails because the variable is an integer.

See Also
None.
realtoatom

Converts between a real number and an atom representation of a real number.

Synopsis
realtoatom(_real, _atom)

Description
If _real is instantiated to a real number, the atom representation is unified with _atom.

If _atom is instantiated to an atom representing a real number, it is converted to a real number and is unified with _real.

At least one of the arguments must be instantiated to a valid value.

Arguments
_atom  An atom, either instantiated or free.
_real  A real number, either instantiated or free.

Examples
The following example shows various real number and atom conversions:
  realtoatom(2.345, _atom1),  % Succeeds. _atom1 is unified with '2.345000000000000e+00'.
  realtoatom(_real1, '1.234')  % Succeeds. _real1 is unified with 1.234000000000000e+00.

See Also
None.
reconsult

Reload a compiled Prolog file into the knowledge base.

Synopsis
reconsult(_fileName)

Description
The Prolog clauses in the new version of _fileName are loaded and asserted in the
knowledge base. First, all clauses that were defined in the previously loaded
version of _fileName are deleted. The file of clauses being consulted must have
been compiled using the compile predicate, the TECpcomp command, or the
Prolog interpreter. A compiled Prolog file is also called a Prolog object file.

Arguments
_fileName
The file name of the Prolog object file. Do not specify the .wic file name
extension. Must be an atom.

Examples
The following example shows the use of this predicate:

unix_hosts(arrakis, support).
unix_hosts(odin, support).
unix_hosts(orange, support).
% Contents of /var/prolog/tec_r.unix_hosts.pro
compile('/var/prolog/tec_r.unix_hosts')
% Creates /var/prolog/unix_hosts.wic.
reconsult('/var/prolog/unix_hosts'),
% Reload and assert the clauses in the compiled
% new version of the unix_hosts.wic file.
unix_hosts(orange, support),
% Query of the knowledge base succeeds, this
% clause exists.
unix_hosts(cerberus, development)
% Query of the knowledge base fails, this clause
% does not exist.

See Also
compile consult
**remove_dups**

Remove duplicate elements from a list.

**Synopsis**

`remove_dups(_list, _pruned)`

**Description**

Removes duplicate elements from list `_list`. The results are unified with `_pruned`.

**Arguments**

- `_list` The original list. Must be instantiated.
- `_pruned` The new list, without duplicates, either instantiated or free.

**Examples**

The following example creates a list with duplicates and then removes the duplicates:

```prolog
_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
% Assign value.
append(_unixHosts, ['odin', 'perro', 'arrakis'], _dupHosts),
% _unixHosts is unified with the list
% ['arrakis', 'rooster', 'dune', 'perro', 'odin', 'perro', 'arrakis'].
remove_dups(_dupHosts, _noDups)
% _noDups is unified with the list
% ['arrakis', 'dune', 'odin', 'perro', 'rooster'].
```

**See Also**

None.
retract

Remove a specific clause from the knowledge base.

Synopsis
retract(_clause)

Description
All clauses that match _clause are removed from the knowledge base.

Arguments
_clause Any Prolog fact or rule.

Examples
The following example shows various uses of the predicate:
assert(unix_hosts(arrakis, support)),
% Assert the fact unix_hosts(arrakis, support) into the knowledge base.

assert(unix_hosts(odin, support)),
% Assert the fact unix_hosts(odin, support) into the knowledge base.

unix_hosts(arrakis, support),
% Succeeds because the fact unix_hosts(arrakis, support) has been asserted into the knowledge base.
retract( unix_hosts(arrakis, support) ),
% Remove the fact unix_hosts(arrakis, support) from the knowledge base.

unix_hosts(odin, support),
% Succeeds because the fact unix_hosts(odin, support) has been asserted into the knowledge base.

unix_hosts(arrakis, _area)
% Fails because the only fact about 'arrakis' has been retracted.

See Also
abolish
**round**

Round a real number to the closest integer.

**Synopsis**

`round(_real)`

**Description**

Defined internally to Prolog as `round(_number) = sign(_real) * trunc(abs(_real) + 0.5)`.

**Arguments**

- `_real` The real number to round.

**Examples**

The following example shows various uses of the predicate:

- `_float1 is 12.49,`  
- `_float2 is -12.49,`  
- `_float3 is 12.51,`  
- `_float4 is -12.51,`  

// Assign values.

- `_int1 is round(_float1),`  
  \[\text{\_int1 is unified with 12.}\]

- `_int2 is round(_float2),`  
  \[\text{\_int2 is unified with -12.}\]

- `_int3 is round(_float3),`  
  \[\text{\_int3 is unified with 13.}\]

- `_int4 is round(_float4),`  
  \[\text{\_int4 is unified with -13.}\]

**See Also**

None.
rremove

Remove the first element from a list.

Synopsis

\texttt{rremove(_element, _list, _tail)}

Description

If \texttt{_element} is free, instantiates \texttt{_element} to the first element of the list \texttt{_list} and unifies the rest of the list with \texttt{_tail}.

If \texttt{_element} is instantiated, \texttt{_element} is removed from the list and the modified list is unified with \texttt{_tail}.

Arguments

\texttt{_element}

If free, the first element of the list. If instantiated, the element to remove from the list. Any term.

\texttt{_list}

The original list to remove an element from.

\texttt{_tail}

The resulting list with the element removed from the original list.

Examples

The following example shows how to remove one element from the list \texttt{_unixHosts} and assign the modified list to the new list \texttt{_tail}:

\texttt{_unixHosts = ['arrakis', 'rooster','dune', 'perro'],}
\texttt{\% Assign value.}

\texttt{rremove(_firstElement, _unixHosts, _tail)}
\texttt{\% Unifies \texttt{_tail} with \texttt{[rooster,dune,perro]}.}

\texttt{rremove('rooster', _unixHosts, _tail2)}
\texttt{\% Unifies \texttt{_tail} with \texttt{[arrakis,dune,perro]}.}

See Also

None.
sort

Sort the elements of a list alphabetically.

Synopsis

\[
\text{sort}(_\text{list}, _\text{sorted})
\]

Description

The list \_list is sorted in ascending order, with duplicates removed. The resulting list is unified with \_sorted.

Arguments

\_list  The list to sort.

\_sorted  A sorted list from \_list.

Examples

The following example shows how to sort the list \_ntHosts:

\[
\begin{align*}
_\text{ntHosts} &= ['\text{charon}', '\text{scary}', '\text{mach5}', '\text{callisto}'], \\
% \text{Assign value.} \\
\text{sort}( _\text{ntHosts}, _\text{sorted} ) \\
% _\text{sorted} \text{ is unified with ['callisto', 'charon', 'mach5', 'scary']}.
\end{align*}
\]

See Also

None.
**sprintf**

Print formatted data to an atom.

**Synopsis**

```lisp
sprintf(_atom, _format, _value)
```

**Description**

_\_atom_ is instantiated with the atom in _\_value_ with the format specified in _\_format_. The format specification is similar to that for the sprintf() function in the C programming language.

**Arguments**

- **_atom**  
The formatted atom that is printed. Must be an atom and must be free.
- **_format**  
The format specification for _\_atom_. Must be an atom and must be instantiated using the format specifications shown in the following list:
  - **%c**  
    Character.
  - **%d**  
    Integer printed in decimal notation.
  - **%e**  
    Real printed in exponential notation.
  - **%f**  
    Real printed in decimal notation.
  - **%g**  
    Real printed in its shortest form (decimal or exponential notation).
  - **%o**  
    Integer printed in octal notation, without sign and leading zero.
  - **%s**  
    String.
  - **%u**  
    Integer printed in unsigned decimal notation.
  - **%x**  
    Integer printed in hexadecimal notation, without sign and leading 0x.

You can specify more detailed conversion specifications between the % sign and the conversion character, as follows:
- `-`  
  Left adjustment.
- `0`  
  Zero padding to the left.
- `n`  
  In cases of an integer or a string, n is the minimum length of the field.
- `n.m`  
  In cases of a real, n is the minimum length of the field and m indicates the number of digits after the decimal point.

- **_value**  
The atomic value or a list of atomic values to be formatted for printing.

**Examples**

The following example shows various uses of the predicate:

```lisp
integer is 123,
real is 12.3,
string = 'Hello, World',
\% Assign values.

sprintf(string1, '%s', string),  
\% string1 is unified with 'Hello, World'.

sprintf(string2, '%20s', string),  
\% string2 is unified with ' Hello, World'.
```

432 IBM Tivoli Enterprise Console: Rule Developer’s Guide
sprintf(_string3, '%-20s', _string),
% _string3 is unified with 'Hello, World '.

sprintf(_atom1, 'Integer in decimal notation:
%d', _integer),
% _atom1 is unified with 'Integer in decimal
% notation: 123'.

sprintf(_atom2, 'Integer in decimal notation
with field width: %10d', _integer),
% _atom2 is unified with 'Integer in decimal
% notation with field width: 123'

sprintf(_atom3, 'Integer in decimal notation
with leading zeros: %010d', _integer),
% _atom3 is unified with 'Integer in decimal
% notation with leading zeros: 0000000123'.

sprintf(_atom4, 'Integer in octal notation: %o', _integer),
% _atom4 is unified with 'Integer in octal
% notation: 173'.

sprintf(_atom5, 'Integer in hexadecimal notation: %x', _integer),
% _atom5 is unified with 'Integer in hexadecimal
% notation: 7b'.

sprintf(_atom6, 'Real in decimal notation: %f', _real),
% _atom6 is unified with 'Real in decimal
% notation: 12.300000'.

sprintf(_atom7, 'Real in decimal notation with
field width: %3.2f', _real),
% _atom7 is unified with 'Real in decimal
% notation with field width: 12.30'.

sprintf(_atom8, 'Real in real notation: %f', _real),
% _atom8 is unified with 'Real in real notation:
% 12.300000'.

sprintf(_atom9, 'Real in exponential notation: %e', _real),
% _atom9 is unified with 'Real in exponential
% notation: 1.230000e+01'.

sprintf(_atom10, 'Real in its shortest form: %g', _real)
% _atom10 is unified with 'Real in its shortest form: 12.3'.

See Also
None.
strip

Remove characters from an atom.

**Synopsis**

```
strip(_atom, _position)

—OR—

strip(_atom, _position, _chars)
```

**Description**

The first form (without the _chars argument) strips blank characters from atom _atom.

The second form (with the _chars argument) strips the character _char from _atom.

The value of _position determines where the strip is to be performed.

**Arguments**

- **_atom**  The atom from which to strip characters. Must be an atom.
- **_chars**  The non-blank character to strip from _atom. Must be an atom.
- **_position**  The location within _atom to perform the strip, as described in following table. Must be an integer representing a three-bit binary number. A binary 1 specifies to strip at one of the positions in the atom. A binary 0 means don’t strip.

<table>
<thead>
<tr>
<th>Beginning of _atom (bit 2)</th>
<th>Middle of _atom (bit 1)</th>
<th>End of _atom (bit 0)</th>
<th>Description</th>
<th>Integer Value for _position Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No stripping.</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Strip characters at the end of the atom.</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Strip characters at the middle of the atom.</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Strip characters at the middle and end of the atom.</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Strip characters at the beginning of the atom.</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Strip characters at the beginning and end of the atom.</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Strip characters at the beginning and middle of the atom.</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Strip characters from the entire atom.</td>
<td>7</td>
</tr>
</tbody>
</table>
Examples

1. The following example shows various uses of the strip predicate with two arguments (stripping blanks):

   ```prolog
   _string = string(' atom with spaces '),
   % instantiate _string with an atom.
   _newstring1 = strip(_string, 4),
   % _newstring1 is unified with 'atom with spaces '. Characters stripped at beginning.
   _newstring2 = strip(_string, 2),
   % _newstring2 is unified with ' atomwithspaces '. Characters stripped at middle.
   _newstring3 = strip(_string, 1),
   % _newstring3 is unified with ' atom with spaces'. Characters stripped at end.
   _newstring4 = strip(_string, 7)
   % _newstring4 is unified with 'atomwithspaces'. Characters stripped from entire atom.
   ```

2. The following example shows the strip predicate with three arguments (stripping a particular non-blank character). In this example a Tivoli Enterprise Console event has been received with double quotation marks in the msg attribute. The value of this attribute is stored in the _msg variable. This example removes the double quotation marks from the atom and stores the new atom in the _newmsg variable.

   ```prolog
   _msg = 'The following file has changed: "C:\CONFIG.SYS"',
   % Assign message text to variable.
   _newmsg = strip(_msg, 7, '"
   % _newmsg is unified with 'The following file has changed: C:\CONFIG.SYS'. Double quotation marks stripped from entire atom.
   ```

See Also

None.
subset
Test whether a list is a subset of another list.

Synopsis
subset(_list1, list2)

Description
Succeeds if every element in _list1 is a member of _list2.

Arguments
_list1 The subset list.
_list2 The original list from which to test whether _list1 is a subset.

Examples
The following example tests whether the list containing two hosts is a subset of the list of all hosts:

```pl
_ntHosts = ['charon', 'scary', 'mach5', 'callisto'],
% Assign value.

subset(['mach5', 'callisto'], _ntHosts)
% Succeeds.
```

See Also
None.
substring
Get a substring from a string.

Synopsis
substring(_atom, _start, _length)

Description
The substring from _atom beginning at index _start and continuing for _length characters is returned. The index begins at zero for the first character. This is similar to the atompart predicate, except that atompart allows you to also find the first occurrence of a character in a string.

Arguments
_atom The original string. Must be an atom.
_length The length of the substring to get from _atom. Must be an integer.
_start The starting position of the substring within _atom. Must be an integer.

Examples
The following example shows that the six-character substring that starts at position 12 is instantiated in _substr. The first character in the string starts at position zero.

(substr =? substring(' atom with spaces ',
12, 6)
% _substr is unified with 'spaces'.

See Also
atompart
**subtract**

Remove elements that are common to two lists.

**Synopsis**

\[
\text{subtract}(_\text{list1}, _\text{list2}, _\text{difference})
\]

**Description**

If \_difference is instantiated, the predicate succeeds if \_difference contains the elements of list \_list1 that are not in list \_list2.

If \_difference is free, \_difference is unified with the list of elements in \_list1 that are not in \_list2.

**Arguments**

\_difference

The difference between \_list1 and \_list2. Any list.

\_list1

Any list.

\_list2

Any list.

**Examples**

The following example shows various uses of the predicate:

\[
\text{subtract(_nthosts, ['charon', 'scary'], ['mach5', 'callisto'])},
\]

% Succeeds.

\[
\text{subtract(_nthosts, ['mach5', 'callisto'], _difference)}
\]

% Instantiates _difference with ['charon','scary'].

**See Also**

None.
term_type

Gets the type of a variable.

**Synopsis**

```prolog
term_type(_term, _type)
```

**Description**

The type of `_term` is unified with `_type`. `_type` is instantiated with one of the atoms from the right column of the table.

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free variable</td>
<td>var</td>
</tr>
<tr>
<td>Integer</td>
<td>integer</td>
</tr>
<tr>
<td>Real</td>
<td>real</td>
</tr>
<tr>
<td>Pointer</td>
<td>pointer</td>
</tr>
<tr>
<td>Atom</td>
<td>atom</td>
</tr>
<tr>
<td>Compound term</td>
<td>functor</td>
</tr>
</tbody>
</table>

**Arguments**

- **_term** Any variable, either instantiated or free.
- **_type** A variable of type atom, either instantiated or free, from the preceding table of types.

**Examples**

The following example shows various tests for the types of variables:

```prolog
atom = 'Another atom',
_integer = 100,
_real = 1.234,
_list = [element1, element2, element3],
% Assign values.

term_type(_integer, _type1),
% Succeeds and _type1 is unified with 'integer'.

term_type(_real, _type2),
% Succeeds and _type2 is unified with 'real'.

term_type(_atom, _type3),
% Succeeds and _type3 is unified with 'atom'.

term_type(_var, _type4),
% Succeeds and _type4 is unified with 'var'.

term_type(_list, _type5)
% Succeeds and _type5 is unified with 'functor'.
```

**See Also**

None.
union

Add uncommon elements between two lists.

Synopsis

union(_list1, _list2, _union)

Description

List _union contains the elements of list _list1 that are not members of list _list2, followed by the elements in _list2.

Arguments

_list1  Any list.
_list2  Any list.
_union  The union of _list1 and _list2. A list.

Examples

The following example shows the union of elements between two lists:

_unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],
% Assign value.

union(_unixHosts, ['perro', 'dune', 'odin'], _union)
% instantiates _union with ['arrakis', 'rooster', 'perro', 'dune', 'odin'].

See Also

None.
write

Write to an open file.

Synopsis
write(_file, _term)

Description
The value of _term is written to the file specified by the file pointer _file. Newline characters are not written.

Arguments
_file The file pointer obtained from opening the file with the fopen predicate.
_term The Prolog term to write.

Examples
This following example Tivoli Enterprise Console rule shows how to write data to the /tmp/event.txt file. It is not generally a good idea to write data to a file every time an event is received because it will degrade the performance of the event server.

rule: write: (event: _event of_class _class,
action: write_assertions: (fopen(_fp, '/tmp/event.txt', 'w'),
write(_fp, 'Event of class '),
write(_fp, _class),
write(_fp, ', '),
write(_fp, _source),
write(_fp, ' received.
')
),
action: close: (fclose(_fp)
)
).

If an event of class Su_Success with source LOGFILE was sent, then the /tmp/event.txt file would contain the following text after the rule ran:
Event of class Su_Success, LOGFILE received.

See Also
None.
Making your predicates and facts available to the Tivoli Enterprise Console rule engine

You can make predicates and facts that you create in Prolog available to the Tivoli Enterprise Console rule engine. The following list shows the various ways to do this:

- Specifying the Prolog files containing the predicates and facts to load in the user_predicates file. These predicates and facts are loaded when the rule base is activated. See “Loading predicates and facts from a file” to use this method.
- Using the assert predicate when a TEC_Start event is received by the rule engine. See “Using the assert predicate with a TEC_Start event” to use this method.
- Using the compile, consult/reconsult predicates in a Tivoli Enterprise™ console rule. See “compile” on page 391, “consult” on page 393, and “reconsult” on page 426 to use this method.

Loading predicates and facts from a file

This procedure lets you specify Prolog source files to load into the knowledge base when a rule base is activated:

1. Create the predicates and facts in Prolog source files.
2. Store the Prolog source files in the TEC_TEMPLATES subdirectory of the rule base to which you want to add the knowledge.
3. In the same TEC_TEMPLATES subdirectory, create a file named user_predicates.
4. For each Prolog source file to load, create an entry in the user_predicates file as shown in the following example:
   ```prolog
   re_user_predicates('filename').
   ```
5. Load and activate the rule base.

If an active rule base uses this method to load predicates and facts into the knowledge base and the rule base is reloaded and reactivated after making modifications to the user_predicates file or the Prolog source files, the previously loaded predicates and facts are deleted and the user_predicates file is read again to reload predicates and facts. This enables you to modify a set of predicates and facts in a running event server without having to stop and restart it.

Note: In order to load predicates and facts from files, you must first ensure that the object dispatcher environment includes the BIM_PROLOG_DIR environment variable. See “compile” on page 391 for more information.

Using the assert predicate with a TEC_Start event

Create a Tivoli Enterprise Console rule with a reception action containing assert predicates that add predicates and facts to the knowledge base. The rule is run when a TEC_Start event is received by the rule engine. The following example shows how a predicate named my_string_match is asserted upon the reception of a TEC_Start event. Use of this particular rule is shown as an example for “assert” on page 383.

```prolog
rule: boot_string_match:
   {event: _start of_class within ['TEC_Start']
    where [],
   reception_action: action0: {
    assert((my_string_match(_string, _substring, _left,
    }.
```
Using the Prolog interpreter

A Prolog interpreter is included with the Tivoli Enterprise Console product. You can use it to develop and test predicates and facts independent of the rule engine.

A typical session for using the interpreter consists of creating your predicates and facts in a text editor and then consulting them from the interpreter.

Any syntax errors in your file are detected during the execution of the consult predicate and shown on your display. Errors in your file cause the consult predicate to fail. When your file is error-free, the consult predicate succeeds.

After the consult predicate succeeds and your file is loaded into the knowledge base, you can query the predicates and facts using standard Prolog interpreter interaction. Also, the consult predicate compiles your file and produces a Prolog object file. After you are satisfied with the testing of your file, the object file can be consulted from Tivoli Enterprise Console rules.

To use the interpreter:
1. Unzip the files from the $BINDIR/TME/TEC/interpreter.tar file.
2. Setup the environment with one of the following commands from a bash or supported UNIX shell. This command sets the environment variable to the root directory of the host on which the rule engine resides.
   - UNIX:
     export BIM_PROLOG_DIR=$BINDIR/TME/TEC
   - Windows:
     export BIM_PROLOG_DIR=%BINDIR%\TME\TEC
3. Start the interpreter with the BIMprolog command from the directory where you unzipped it, or elsewhere if you have a path set to the $BIM_PROLOG_DIR/bin directory.
   The interpreter displays the ?- prompt.
4. To exit the interpreter, enter halt. (note the period).

An example of using the Prolog interpreter

For this example, assume you have a /prolog/source/connected.pro source file that contains the following clauses:

connected(nodea,nodeb).
connected(nodeb,nodec).

is_connected(_first,_last):-
    (connected(_first,_last) -> !,true)
\connected\(_first, _next)\),
\is\_connected\(_next, _last\)
\).  

Now assume you have set the environment variable for the interpreter in a shell and have started the interpreter, and the interpreter prompt is displayed.

The following example runs the consult predicate with the example file:

?- consult('connected.pro').
compiling -c+ connected.pro
loaded connected.pro
Yes

The following example shows various queries of the knowledge base. Notice that the last query fails because that knowledge does not exist.

?- \is\_connected\(nodea, nodeb)\).
Yes
?- \is\_connected\(nodea, nodec)\).
Yes
?- \is\_connected\(nodea, noded\).
No
Appendix B. BNF grammar for BAROC files

Non-terminals

parse ::= ( root | enumType | eventClass )* eof
root ::= name ( <COLON> <TEC_CLASS> )? <SEMICOLON> <END>
enumType ::= <ENUMERATION> name ( <INTEGER> ( name | string ) )* <END>
eventClass ::= <TEC_CLASS> <COLON> name ( superClasses )? ( <DEFINES> <LBRACE> ( name <COLON> ( ( complexity )? type )? ( ( slotFacets | slotFacets )+ )? <SEMICOLON> )? <RBRACE> )? <SEMICOLON> <END> ( <SEMICOLON> )?
slotFacets ::= <COMMA> slotFacet
slotFacet ::= default_value | parse_value | dup_detect | reverse
reverse ::= <REVERSE> <EQUALS_SIGN> ( name | string )
parse_value ::= <PARSE> <EQUALS_SIGN> ( name | string )
dup_detect ::= <DUP_DETECT> <EQUALS_SIGN> ( name | string )
superClasses ::= <ISA> ( name | ( <LSQBRACH> name ( <COMMA> name )* <RSQBRACH> ) )
complexity ::= ( <SINGLE> | <LIST_OF> )
type ::= ( <STRING_TOK> | <INTEGER_TOK> | <INT32> | <REAL> | <NAME> )
default_value ::= <DEFAULT_TOK> <EQUALS_SIGN> ( <FLOAT> | string | list | name )
list ::= <LSQBRACH> ( ( name | string | <FLOAT> ) ( <COMMA> ( name | string | <FLOAT> ) )+)? <RSQBRACH>
tokToStr ::= // Returns the string representation of the token.
string ::= <STRING>
name ::= ( <INTEGER> | <NAME> )
eof ::= ( "\u001a" | <EOF> )

Terminals

< COLON: "":" >
< COMMA: "," >
< DEFAULT_TOK: "default" >
< DEFINES: "DEFINES" >
< DOT: ",." >
< DUP_DETECT: "dup_detect" >
< END: "END" >
Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
2Z4A/101  
11400 Burnet Road  
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.
If you are viewing this information in softcopy form, the photographs and color illustrations might not appear.

**Trademarks**

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- IBM
- IBM Logo
- Tivoli
- Tivoli Logo
- DB2
- IBMLink
- NetView
- Tivoli Enterprise Console
- TME

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.
## Index

### Special characters
- (Prolog anonymous variable) 61
  - class specification 61
  - class class specification 61
  - class outside class specification 62
  - class variable 61
  - class within class specification 61
  - event variable 60
  - (mathematical subtraction) 316, 323
  - (pointer offset subtraction) Prolog predicate 357
  - (pointer subtraction) Prolog predicate 359
  - (sign reversal) Prolog predicate 360
  - (subtraction) Prolog predicate 361
  != (mathematical not equal to) 314, 321, 326
  / (mathematical division) 317, 323
  // Prolog predicate 364
  / Prolog predicate 364
  */ */ (comment delimiters) 52, 348
  .conf file 192
  .rbtargets directory 14
  .rls file 49
  .rpk file 49
  * (mathematical multiplication) 317, 323
  * Prolog predicate 362
  \ 371
  \= Prolog predicate 369
  @\ 365
  @> Prolog predicate 367
  @>= Prolog predicate 368
  @=\ 366
  % (comment delimiter) 52, 348
  > (mathematical greater than) 314, 321, 326
  >= (mathematical greater than or equal to) 314, 321, 326
  >= Prolog predicate 379
  && 338
  # (comment delimiter) 52
  + (addition) Prolog predicate 356
  + (mathematical addition) 316, 323
  + (pointer offset addition) Prolog predicate 370
  = Prolog predicate 372
  == Prolog predicate 373
  >= Prolog predicate 374
  !=\ 376
  != = (mathematical equal to) 314, 320, 326, 334
  == Prolog predicate 377

### A
- abolish Prolog predicate 380
- absolute float 324
- absolute function 317
- ACK status 8
- acl attribute 7
- action body, rule 67
- action library 311
- actions, rules
  - See rules, actions 67
- active rule base 31

### B
- back slashes, converting 197
- BAROC files
  - */ */ (comment delimiters) 52
  - attribute definitions, retrieving 92
  - comments 52
  - data types, mapping to Prolog 349
  - enumeration definitions, described 43
  - enumeration definitions, retrieving 94
  - importing modified files 44
BAROC files (continued)
location in rule base 19
naming conventions 49
order 18, 41
root.baroc 18, 43
See also event classes 35
syntax 36
tec.baroc 18, 43
UTF-8 support 44
BAROC manipulation rule language predicates 72
base event 41
bo_add_at_slotval_begin rule language predicate 89
bo_add_at_slotval_end rule language predicate 90
bo_get_class_of rule language predicate 91
bo_get_class_slots rule language predicate 92
bo_get_enum_options rule language predicate 94
bo_get_slotval rule language predicate 95
bo_is_defined_for_class rule language predicate 96
bo_is_direct_super_of rule language predicate 97
bo_is_super_of rule language predicate 98
bo_remove_from_slotval rule language predicate 99
bo_reset_default_slotval rule language predicate 100
bo_set_slotval rule language predicate 101
books
see publications x, xi
Boolean variable 311
Boolean and 338
Boolean not 338
Boolean or 339

C

cache, event
See event cache 9
cancel_all_timers rule language predicate 102
cancel_timer rule language predicate 103
causal relationships between events
creating a compound rule, rule builder procedure 258
recording the relationships 217
special cases 218
using relationship information 219
cause events
described 219
linking effect events 178
processing guidelines 221
searching for most causal 152
unlinking effect events 214
cause_date_reception attribute 7
cause_event_handle attribute 7
CDATA tag 305
change request
described 10
drop 139
external 46
place 182
processing by rule engine 46
senders 64
change rule filters
attribute change filter 65
described 64
sender filter 64
senders of change requests 64
change rules
change request, placing 182
See also change request 10
described 10
change rules (continued)
processing by rule engine 46
structure 54
change_event_administrator rule language predicate 105
change_event_severity rule language predicate 106
change_event_status rule language predicate 107
check_all_thresholds rule language predicate 108
check_and_increment_count rule language predicate 110
check_event_criteria rule language predicate 111
check_threshold rule language predicate 113
classes
See event classes 35
clear_closed_events rule language predicate 115
clearing events
checking for existence 177
defining a clearing event 126
described 220
processing guidelines 222
searching for 84
target events, finding 79, 83
clears rule language predicate 116
cloning rules 296, 308
CLOSED status 8
collector rules 292
commands
postemsg 6, 231
upgrade_gui.sh 243
wpostemsg 6, 231
wr
rule base targets, creating 14
rule packs, creating 16
usage recommendation 13
wr_upgrade 13
wtdumprl 5
comments
BAROC files 52
Prolog 348
rule language 52
commit_action rule language predicate 118
commit_rule rule language predicate 119
commit_set_rule language predicate 120
compile Prolog predicate 391
compiling a rule base, rule builder procedure 30, 263
compiling older rule bases 13
complex_type 38
compound rules 258
See also correlation rules 11
compound rules, creating, rule builder procedure 258
concat string 329
conditions, simple rule 57
conf file 192
constants, rule language 52
consult Prolog predicate 393
contains string 328
conventions
typeface xiii
convert data type Prolog predicates 351
convert_ascii_time rule language predicate 121
convert_gm_time rule language predicate 122
convert_local_time rule language predicate 123
copying a rule base, rule builder procedure 26
copying rule bases 15
correlating events 217
correlating events, rule builder 260
correlation examples 217
correlation rule language predicates 70
correlation rules
described 11
examples 217
See also compound rules 258
structure 58
correlation, state
See state correlation 283
counters in rule actions 109, 172
create_cache_search_criteria rule language predicate 124
create_clearing_event rule language predicate 126
create_event_criteria rule language predicate 129
create_event_sequence rule language predicate 132
create_threshold rule language predicate 137
creating a compound rule, rule builder procedure 258
creating a new rule base, rule builder procedure 23
creating a new rule set, rule builder procedure 247
creating a simple rule, rule builder procedure 249
credibility attribute 8
criteria
checking against event 111
creating for event 129
creating for event cache search 124
evaluation of event 223
current rule base 31
customer support
see software support  xii

data types, mapping BAROC to Prolog 349
date_reception attribute 8, 9
dayOfWeek string 330
dayOfWeekString string 330
DBCS support
See UTF-8 support 33
debug
detailed debugging for rules 204
rules
logging error messages 180
profile 239
trace 234
tracing user-defined Prolog predicates 213
debug rule language predicates 77
decrement_slot rule language predicate 138
decrementing attribute values 138
default facet 39
default rule base 15, 19
delete Prolog predicate 394
deleting a rule base, rule builder procedure 27
developing event classes 35
developing rule bases 3
directives
fire_on_non_leaf 40, 69
profile 69
See also rules, directives 69
trace 69
directory names, notation  xiv
directory paths, converting back slashes 197
directory paths, rule base 49
directory permissions for UNIX rule bases 32
discard events 140
disjoint Prolog predicate 395
distributed rule bases 12
distributing files with rule bases 15
domain-significant events 4
double-byte character support (DBCS)
See UTF-8 support 33
drop_change_request rule language predicate 139
drop_received_event rule language predicate 140
dropping change requests 139
dropping received events 140
DTD file 305
dup_detect facet 39
duplicate events
counting duplicates 78
described 72
dup_detect facet 39
searching the event cache for all duplicates 80
searching the event cache for the first duplicate 153
duplicates rules 289
duration attribute 8

editing a rule set, rule builder procedure 248
editing an existing rule base, rule builder procedure 24, 247
editing rule sets with a text editor 243
effect events
described 220
linking cause events 178
processing guidelines 221
searching for 155
unlinking cause events 214
empty_list Prolog predicate 396
encoding, UTF-8 305
endsWith string 328
trace 69
enterprise-significant event 3
enumeration definitions, described 43
enumeration definitions, retrieving 94
environment variables, notation  xiv
environment variables, TEC_UMASK 32
erase_globals rule language predicate 141
event actions for simple rules 57
event activity reporting
capturing information 215
defining reporting criteria 174
resetting counts 198
writing the report 185
event activity rule language predicates 73
event adapters 5
event analysis xi, 3, 35
event attributes
acl 7
adapter, supplied by 5
adding a value to a list 89, 90
administrator
changing the value 105
described 7
setting the value 205
cause_date_reception 7
cause_event_handle 7
complex_type 38
conditions 223
credibility 8
date_reception 8, 9
decrementing values 138
default value, setting 100
definition in BAROC file 37
definitions, checking 96
duration 8
event server, supplied by 7
event_handle 8, 9
facets 39
incrementing values 171
event attributes (continued)
  mapping to Prolog data types  349
  modified, mark as  189
  msg  206
  num_actions  8
  retrieving values  162
  server_handle  8, 9
  server_path  8
  server_path, parsing  193
  severity  106, 208
  simple_type  38
  status
    changing the value  107
    described  8
    setting the value  209
  test for values  63
  value, deleting from a list  99
  value, modifying  101

event cache
  all duplicates search  80
  all instances search  82
  cause event search  152
  described  9
  duplicate events, searching for all duplicates  80
  duplicate events, searching for the first duplicate  153
  effect event search  155
  first duplicate search  153
  first instance search  156
  forced cleaning  45
  related event search  158, 224
  reverse lookup  221
  search criteria, creating  124
  searching based on event criteria  203
  writing to file  183

event cache search rule language predicates  71
EVENT class  41
event class definition files
  See BAROC files  35

event classes
  attribute definitions  37, 92
  attribute definitions, checking  96
  attribute values, retrieving  95
  attributes, retrieving information about  92
  base event  41
  class names, retrieving  91
  definition  36
  design recommendation  43
  designing event classes  35
  distributed rule base  12
  enumerations, compared to  43
  EVENT class  41
  fire_on_non_leaf directive  69
  hierarchy  37
  hierarchy, writing to file  184
  importing modified BAROC files  44
  inheritance  40, 42
  leaf  40
  leaf-node  40
  naming conventions  49
  non-leaf classes, rule evaluation  69
  order  41
  root event  41
  See also BAROC files  35
  subclass, described  40
  superclass checking  97, 98
  superclass, described  40
  synchronizing among distributed rule bases  18

event classes (continued)
  triggering rules with non-leaf classes  69
  event conditions for simple rules  57
  event correlation rule language predicates  70, 222
  event criteria rule language predicates  73
  event criteria, checking  111
  event criteria, creating  129
  event filters
    attribute conditions  63
    attribute filter  11, 63
    class filter  11, 60
    class specification
      _  61
      _class  61
      _class outside  62
      _class within  61
      described  61
    described  11, 60
    multiple solutions, tracing  238
    of_class operator  61
    operators  63
    structure  60
    syntax  60
    variable instantiation  64
    variables  60
    where clause  63

event ID  9

event sequence
  attribute conditions  223
  attribute conditions, defining  85
  attribute exceptions, defining  87
  attribute sequence  88
  attribute values, changing  88
  cause event search  152
  clearing events, defining  116, 126
  creating an event sequence  132, 217
  definition  217
  described  217
  effect event search  155
  matching different attributes in other events  87
  related event search  158

event server
  .conf file  192
  configuration file  163, 192
  forwarding events  160, 192
  path parsing  193
  sending events  160, 192
  server_path attribute parsing  193
  start  266
  stop  266

event_handle attribute  8, 9

events
  activity report
    capturing information  215
    defining criteria  174
    resetting counts  198
    writing the report  185
  administrator, changing  105, 205
  administrator, setting  205
  attribute conditions  63, 223
  cache searching  124, 203
  cache, described  9
  causal relationships
    creating a compound rule, rule builder procedure  258
    recording relationships  217
    using relationship information  219
events (continued)

cause events
- described 219
- linking effect events 178
- processing guidelines 221
- searching for 152
- unlinking effect events 214
change request
- described 10
- external 46
- place 182
- processing by rule engine 46
clearing events
- checking for existence 177
- defining a clearing event 116, 126
- described 220
- finding target events for clearing 79, 83
- processing guidelines 222
- searching for 84
closed events, clearing 115
correlation examples 217
counting 109, 172
criteria
- check 111
- create 129
- evaluation of event 223
described 4
discarding events 140
domain-significant 4
dropping events 140
duplicate events
- described 72
- dup_detect facet 39
- searching the event cache for all duplicates 80
- searching the event cache for the first duplicate 153
duplicates 289
effect events
- described 220
- linking cause events 178
- processing guidelines 221
- searching for 155
- unlinking cause events 214
enterprise-significant 3
filtering 6
first instance search 156
forwarding to another server 160, 192
generating an internal event 161
ID 9
instances, searching the cache 82
internal events, processing 46
internal events, sending 161
message, changing 206
message, setting 206
path, event server 193
reanalysis 10, 46
reception log, viewing events 5
redo request 10, 46
related event search 158, 224
relationships, defining 223
reporting activity
- See events, activity report 185
- sending from command line 6, 231
- sending to another server 160, 192
- sending with SendEvents program 232
- sending with tec_agent_demo program 231
severity, changing 106, 208
severity, setting 208
first_related_event rule language predicate 158
firstEvent mode 289
flisting Prolog predicate 398
float functions 323
float predicates 320
Float variable 311
float.abs 324
float.add 323
float.div 323
float.eq 320
float.ge 321
float.gt 321
float.le 322
float.lt 322
float.max 324
float.min 324
float.mul 323
float.ne 321
float.neg 325
float.range 325
float.sub 323
flow control in rules...
flow control rule language predicates 72
fopen Prolog predicate 399
forced cleaning of the event cache 45
forward slashes, converting from back slashes 197
forward_event rule language predicate 160
forwardEvents mode 290
forwarding events 160, 192
free, described 349

G

general purpose rule language predicates 76
generate_event rule language predicate 161
generating internal events 161
global variable rule language predicates 75
global variables...
ground Prolog predicate 400
ground, described 349

H

hourOfDay string 331

I

I/O Prolog predicates 355
icons xiv
Event Server xiv
Profile xv
Rulebase xiv
id attribute 288
ignorecase equal string 328
ignorecase not equal string 328
importing a rule set into a rule base, rule builder procedure 28
importing BAROC files into a rule base, rule builder procedure 29
importing modified BAROC files 44
importing objects into a rule base 13
importing predicates 18
importing rule packs 15
importing rule sets 15
increment_slot rule language predicate 171
incrementing attribute values 171
infix syntax 313
inheritance, event class 40, 42
init_count rule language predicate 172
init_event_activity rule language predicate 174
instantiated, described 349
instantiation and unification Prolog predicates 350
Int variable 311
int_to_hex Prolog predicate 401
intabs 317
int.add 316
int.div 317
int.eq 314
int.ge 315
int.gt 314
int.le 315
int.lt 315
int.max 318
int.min 318
int.mod 318
int.mul 317
int.neg 319
int numOfEvents 319
int.range 319
int.sub 316
int.timeNow 320
int.timeSpan 320
integer functions 316
integer predicates 314
integer Prolog predicate 402
integer set predicates 339
internal events, generating 161
internal events, processing 46
internationalization UTF-8 encoding 305
internationalization support 33, 44, 243
interpreter, Prolog 443
intersect 337
<table>
<thead>
<tr>
<th>Index</th>
<th>457</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersect Prolog predicate</td>
<td>403</td>
</tr>
<tr>
<td>intset.isMemberOf</td>
<td>339</td>
</tr>
<tr>
<td>inttoatom Prolog predicate</td>
<td>404</td>
</tr>
<tr>
<td>IP address predicates</td>
<td>334</td>
</tr>
<tr>
<td>IP address set predicates</td>
<td>339</td>
</tr>
<tr>
<td>IP address, checking for unreachable</td>
<td>176</td>
</tr>
<tr>
<td>ip_node_unreachable rule language predicate</td>
<td>176</td>
</tr>
<tr>
<td>ipaddress.eq</td>
<td>334</td>
</tr>
<tr>
<td>ipaddress.isMemberOf</td>
<td>339</td>
</tr>
<tr>
<td>ipaddress.ne</td>
<td>335</td>
</tr>
<tr>
<td>is Prolog predicate</td>
<td>405</td>
</tr>
<tr>
<td>is_closing_event rule language predicate</td>
<td>177</td>
</tr>
<tr>
<td>is_list Prolog predicate</td>
<td>406</td>
</tr>
<tr>
<td>isMemberOf</td>
<td>336, 339</td>
</tr>
<tr>
<td>isSubsetOf</td>
<td>336</td>
</tr>
<tr>
<td>J</td>
<td>337</td>
</tr>
<tr>
<td>join</td>
<td>337</td>
</tr>
<tr>
<td>K</td>
<td>337</td>
</tr>
<tr>
<td>knowledge base Prolog predicates</td>
<td>355</td>
</tr>
<tr>
<td>knowledge base, described</td>
<td>347</td>
</tr>
<tr>
<td>L</td>
<td>337</td>
</tr>
<tr>
<td>lastEvent mode</td>
<td>289</td>
</tr>
<tr>
<td>launching a program from rules</td>
<td>142, 144</td>
</tr>
<tr>
<td>launching a task from rules</td>
<td>147, 149</td>
</tr>
<tr>
<td>leaf class</td>
<td>40</td>
</tr>
<tr>
<td>leaf-node class</td>
<td>40</td>
</tr>
<tr>
<td>length Prolog predicate</td>
<td>407</td>
</tr>
<tr>
<td>length string</td>
<td>332</td>
</tr>
<tr>
<td>link_effect_to_cause rule language predicate</td>
<td>178</td>
</tr>
<tr>
<td>linking cause and effect events</td>
<td>178</td>
</tr>
<tr>
<td>list manipulation Prolog predicates</td>
<td>353</td>
</tr>
<tr>
<td>listing known rule bases, rule builder procedure</td>
<td>24</td>
</tr>
<tr>
<td>listing rule packs</td>
<td>16</td>
</tr>
<tr>
<td>listing rule sets</td>
<td>16</td>
</tr>
<tr>
<td>listing rule sets and BAROC files in a rule base, rule builder procedure</td>
<td>25</td>
</tr>
<tr>
<td>listing the loaded rule base, rule builder procedure</td>
<td>25</td>
</tr>
<tr>
<td>load_globals rule language predicate</td>
<td>179</td>
</tr>
<tr>
<td>loaded rule base</td>
<td>31</td>
</tr>
<tr>
<td>loading and activating a rule base, rule builder procedure</td>
<td>30</td>
</tr>
<tr>
<td>loading user-defined predicates</td>
<td>442</td>
</tr>
<tr>
<td>log_error rule language predicate</td>
<td>180</td>
</tr>
<tr>
<td>lowertoupper Prolog predicate</td>
<td>408</td>
</tr>
<tr>
<td>M</td>
<td>337</td>
</tr>
<tr>
<td>manuals</td>
<td>288</td>
</tr>
<tr>
<td>matching rules</td>
<td>288</td>
</tr>
<tr>
<td>mathematical expression Prolog predicates</td>
<td>353</td>
</tr>
<tr>
<td>mathematical less than or equal to symbol</td>
<td>315, 322, 327</td>
</tr>
<tr>
<td>mathematical less than symbol</td>
<td>315, 322, 327</td>
</tr>
<tr>
<td>max float</td>
<td>324</td>
</tr>
<tr>
<td>max function</td>
<td>318</td>
</tr>
<tr>
<td>MBCS support</td>
<td>33</td>
</tr>
<tr>
<td>See UTF-8 support</td>
<td>33</td>
</tr>
<tr>
<td>member Prolog predicate</td>
<td>409</td>
</tr>
<tr>
<td>message, changing for an event</td>
<td>206</td>
</tr>
<tr>
<td>message, setting for an event</td>
<td>206</td>
</tr>
<tr>
<td>min float</td>
<td>324</td>
</tr>
<tr>
<td>min function</td>
<td>318</td>
</tr>
<tr>
<td>mod function</td>
<td>318</td>
</tr>
<tr>
<td>modifying rule sets</td>
<td>17</td>
</tr>
<tr>
<td>month string</td>
<td>332</td>
</tr>
<tr>
<td>monthString string</td>
<td>333</td>
</tr>
<tr>
<td>msg attribute</td>
<td>206, 285</td>
</tr>
<tr>
<td>multiple-byte character support (MBCS)</td>
<td>33</td>
</tr>
<tr>
<td>N</td>
<td>337</td>
</tr>
<tr>
<td>name Prolog predicate</td>
<td>410</td>
</tr>
<tr>
<td>naming conventions, rule language</td>
<td>49</td>
</tr>
<tr>
<td>negate float</td>
<td>325</td>
</tr>
<tr>
<td>negate function</td>
<td>319</td>
</tr>
<tr>
<td>newsgroups</td>
<td>xiii</td>
</tr>
<tr>
<td>nmember Prolog predicate</td>
<td>411</td>
</tr>
<tr>
<td>nmembers Prolog predicate</td>
<td>412</td>
</tr>
<tr>
<td>non-editable rule sets</td>
<td>243</td>
</tr>
<tr>
<td>not Prolog predicate</td>
<td>413</td>
</tr>
<tr>
<td>notation</td>
<td>178</td>
</tr>
<tr>
<td>environment variables</td>
<td>xiv</td>
</tr>
<tr>
<td>path names</td>
<td>xiv</td>
</tr>
<tr>
<td>typeface</td>
<td>xiv</td>
</tr>
<tr>
<td>num_actions attribute</td>
<td>8</td>
</tr>
<tr>
<td>number Prolog predicate</td>
<td>414</td>
</tr>
<tr>
<td>numOfEvents function</td>
<td>319</td>
</tr>
<tr>
<td>O</td>
<td>337</td>
</tr>
<tr>
<td>object file, Prolog</td>
<td>391</td>
</tr>
<tr>
<td>of_class operator</td>
<td>61</td>
</tr>
<tr>
<td>online publications</td>
<td>xiv</td>
</tr>
<tr>
<td>accessing</td>
<td>xi</td>
</tr>
<tr>
<td>OPEN status</td>
<td>9</td>
</tr>
<tr>
<td>ordering publications</td>
<td>xii</td>
</tr>
<tr>
<td>P</td>
<td>337</td>
</tr>
<tr>
<td>parse facet</td>
<td>39</td>
</tr>
<tr>
<td>parser grammar for rules</td>
<td>53</td>
</tr>
<tr>
<td>path names, notation</td>
<td>xiv</td>
</tr>
<tr>
<td>paths, converting back slashes</td>
<td>197</td>
</tr>
<tr>
<td>performance tips, rule engine</td>
<td>47</td>
</tr>
<tr>
<td>place_change_request rule language predicate</td>
<td>182</td>
</tr>
<tr>
<td>plain rules, described</td>
<td>10</td>
</tr>
<tr>
<td>plain rules, structure</td>
<td>53</td>
</tr>
<tr>
<td>pointer Prolog predicate</td>
<td>415</td>
</tr>
<tr>
<td>pointeroffset Prolog predicate</td>
<td>416</td>
</tr>
<tr>
<td>pointertoatom Prolog predicate</td>
<td>417</td>
</tr>
<tr>
<td>pointertoint Prolog predicate</td>
<td>418</td>
</tr>
<tr>
<td>postemsg command</td>
<td>6, 231</td>
</tr>
<tr>
<td>predicate</td>
<td>324</td>
</tr>
<tr>
<td>predicate library</td>
<td>312</td>
</tr>
<tr>
<td>predicates</td>
<td>337</td>
</tr>
<tr>
<td>boolean</td>
<td>338</td>
</tr>
<tr>
<td>described</td>
<td>69, 345</td>
</tr>
<tr>
<td>float</td>
<td>320</td>
</tr>
<tr>
<td>import, automatic</td>
<td>18</td>
</tr>
<tr>
<td>integer</td>
<td>314</td>
</tr>
<tr>
<td>integer set</td>
<td>339</td>
</tr>
<tr>
<td>integers</td>
<td>314</td>
</tr>
<tr>
<td>IP address</td>
<td>334</td>
</tr>
<tr>
<td>IP address set</td>
<td>339</td>
</tr>
<tr>
<td>loading user-defined Prolog predicates</td>
<td>442</td>
</tr>
</tbody>
</table>
Prolog predicates (continued)
\( \leq \) 375
\( == \) 377
abolish 380
append 381
ascii 382
assert 383, 442
atom 385
atom manipulation predicates 352
atomconcat 389
atomic 386
atomlength 387
atompart 388
atomtolist 390
categories 350
compile 391
consult 393
delete 394
described 345
disjoint 395
empty_list 396
cfclose 397
flisting 398
fopen 399
ground 400
I/O predicates 355
instantiation and unification predicates 350
int_to_hex 401
integer 402
intersect 403
inttoatom 404
is 405
is_list 406
knowledge base predicates 355
length 407
list manipulation predicates 353
loading user-defined predicates 442
lowertoupper 408
mathematical expression predicates 353
member 409
name 410
nmember 411
nmembers 412
not 413
number 414
online information 345
pointer 415
pointeroffset 416
pointertoatom 417
pointertoint 418
printf 419
Prolog interpreter 443
quick reference 350
read 421
readln 422
real (convert integer) 423
real (test real type) 424
realtoatom 425
reconsult 426
remove_dups 427
retract 428
round 429
rremove 430
sort 431
sprint 432
strip 434

IBM Tivoli Enterprise Console: Rule Developer’s Guide
Prolog predicates (continued)
  subset 436
  substring 437
  subtract 438
  term_type 439
  test data type and variable predicates 351
  union 440
  write 441
publications
  accessing online xi
  ordering xii
R
  range float 325
  range function 319
  rbtargets directory 14
  re_after_match rule language predicate 186
  re_before_match rule language predicate 187
  re_create rule language predicate 188
  re_mark_as_modified rule language predicate 189
  re_match rule language predicate 190
  re_search_string rule language predicate 191
  re_send_event_conf rule language predicate 192
  re_split_event_id rule language predicate 193
  re_substitute rule language predicate 194
  re_substitute_global rule language predicate 195
read Prolog predicate 421
readln Prolog predicate 422
real (convert integer) Prolog predicate 423
real (convert float) Prolog predicate 424
real (convert real) Prolog predicate 425
realtoatom Prolog predicate 426
reanalysis of an event 10, 196
reception log, viewing events 5
reception_action 68
reconsult Prolog predicate 427
redo request
  described 10
  processing, rule engine 46
  redo_analysis rule language predicate 196
redo_action 68
redo_analysis rule language predicate 196
reference event, described 71
regular expression rule language predicates 74
regular expressions
  creating a regular expression 188
  match
    replace all strings found 195
    replace string found 194
    return no string 191
    return string after 186
    return string before 187
    return string found 190
related event search 158
remove_bslashes rule language predicate 197
remove_dups Prolog predicate 427
reports
  class tree 184
  event activity
    capturing information 215
    defining criteria 174
    resetting counts 198
    writing the report 185
  profile, rule execution 239
reset_event_activity rule language predicate 198
reset_global_grp rule language predicate 199
resolve_time rule language predicate 200
RESPONSE status 9
response, automated 9
restarting the event server 266
retract Prolog predicate 428
reverse lookup 221
ris file 49
root event 41
root.baroc file 18, 43
round Prolog predicate 429
rpk file 49
remove Prolog predicate 430
rule actions
  See rules, actions 68
rule base targets
  described 12, 14
  event classes 18
  See also event classes 35
  importing rule packs 15
  importing rule sets 15
  structure 16
rule bases
  active 31
  BAROC file location 19
  class hierarchy, writing to file 184
  commands 14
  compile 13
  compile, rule builder procedure 263
  compile, wrb command
    See the Tivoli Enterprise Console Reference Manual 263
  copy 15
  default 15, 19
  described 12
  development 3, 12
  directory paths 49
  directory permissions, UNIX 32
  distributed 12
  editing an existing rule base, rule builder procedure 247
  event classes 12
  See also event classes 35
  example 19
  importing objects 13
  importing rule packs 15
  importing rule sets 15
  loaded 31
  manipulation 13
  naming conventions 49
  rule base targets 12, 14
  See also rule base targets 14
  rule builder
    See rule builder 243
    rule packs 15
    See also rule packs 15
    rule sets 15
    See also rule sets 15
    structure 14, 16
    targets 12, 14
    See also rule base targets 14
  UNIX directory permissions 32
  upgrading 13
  wrb command 13
rule builder
  compiling a rule base 30, 263
  copying a rule base 26
  correlating events 260
  creating a compound rule 258
  creating a new rule base 23
rule builder

creating a new rule set 247
creating a simple rule 249
deleting a rule base 27
described 243
editing a rule set 248
editing an existing rule base 24, 247
event actions for simple rules 255
event conditions for simple rules 253
examples 267
importing a rule set into a rule base 28
importing BAROC files into a rule base 29
listing known rule bases 24
listing rule sets and BAROC files in rule base 25
listing the loaded rule base 25
loading and activating a rule base 30
overview 244
procedures 244
saving a rule set 262
start 246
text editor, using 243
upgrading rules from previous releases 23
UTF-8 support 243

Rule cache full
forced cleaning 45

rule engine
configuration, retrieving 163
described 45
distributed 12
event processing 45
knowledge base 347
performance tips 47
processing change rules 46
processing timer rules 47
transaction processing 47

rule language predicates
add_to_repeat_count 78
all_clear_targets 79
all_duplicates 80
all_instances 82
any_clear_target 83
any_clearing_event 84
attr_condition 85
attr_exception 87
attr_sequence 88
BAROC manipulation predicates 72
bo_add_at_slotval_begin 89
bo_add_at_slotval_end 90
bo_get_class_of 91
bo_get_class_slots 92
bo_get_enum_options 94
bo_get_slotval 95
bo_is_defined_for_class 96
bo_is_direct_super_of 97
bo_is_super_of 98
bo_remove_from_slotval 99
bo_reset_default_slotval 100
bo_set_slotval 101
cancel_all_timers 102
cancel_timer 103
categories 70
change_event_administrator 105
change_event_severity 106
change_event_status 107
check_all_thresholds 108
check_and_increment_count 109
check_event_criteria 111

rule language predicates (continued)
check_threshold 113
clear_closed_events 115
clears 116
commit_action 118
commit_rule 119
commit_set 120
convert_ascii_time 121
convert_gm_time 122
convert_local_time 123
create_cache_search_criteria 124
create_clearing_event 126
create_event_criteria 129
create_event_sequence 132
create_threshold 137
debug predicates 77
decrement_slot 138
described 10, 70
drop_change_request 139
drop_received_event 140
eraseGlobals 141
event activity predicates 73
event cache search predicates 71
event correlation predicates 71, 222
event criteria predicates 73
event_predicate 142
exec_program 142
exec_program_local 144
exec_task 147
exec_task_local 149
first_causal_event 152
first_duplicate 153
first_effect_event 155
first_instance 156
first_related_event 158
flow control 72
forward_event 160
general purpose predicates 76
generate_event 161
get_attributes 162
global_variable predicates 75
global_exists 170
increment_slot 171
init_count 172
init_event_activity 174
ip_node_unreachable 176
is_clearing_event 177
link_effect_to_cause 178
loadGlobals 179
log_error 180
place_change_request 182
print_cache 183
print_class_tree 184
print_event_activity 185
quick reference 70
re_after_match 186
re_before_match 187
re_create 188
re_mark_as_modified 189
re_match 190
re_search_string 191
rule language predicates (continued)
  re_send_event_conf 192
  re_split_event_id 193
  re_substitute 194
  re_substitute_global 195
  redo_analysis 196
  regular expression predicates 74
  remove_bslashes 197
  reset_event_activity 198
  reset_global_grp 199
  resolve_time 200
  saveGlobals 202
  search_cache 203
See also Prolog predicates or predicates 355
set_detailed_debugging 204
set_event_administrator 205
set_event_message 206
set_event_severity 208
set_event_status 209
set_global_var 210
set_log_error_source 211
set_timer 212
threshold predicates 74
time manipulation predicates 75
trace_it 213
unlink_from_cause 214
update_event_activity 215
rule packs
described 15
example 16
files 49
list 16
naming conventions 49
structure 16
rule sets
  creating a new rule set, rule builder procedure 247
default 15
described 15
  editing a rule set, rule builder procedure 248
files 49
flow control 119, 120
import 15
list 16
modify 17
naming conventions 49
rule builder, created with 243
saving a rule set, rule builder procedure 262
structure 16
rule types
  change rules 54
    See also change rules 54
  compound rules 258
  correlation rules 58
    See also correlation rules 58
  plain rules 53
    See also plain rules 53
  simple rules 57
    See also simple rules 57
  timer rules 55
    See also timer rules 55
rules
  /* */ (comment delimiters) 52
actions
  action 68
  action body 67
  action name 67
rules (continued)
  actions (continued)
    action type 67
    change rules 67
    correlation rules 59
    described 67
    examples 12
    flow control 118
    naming conventions 49
    plain rules 67
    processing order 68
    reception_action 68
    redo_action 68
    simple rule 57
    timer rules 67
types 68
analysis
    profile 69, 239
    trace 69, 234
change rule filters
    See change rule filters 64
change rules 46, 54
    See also change rules 46
correlation rules 58
    See also correlation rules 58
correlation, event sequence 132
correlation, specifying with compound rule 260
counters 109, 172
debug
    logging error messages 180, 211
    tracing rules 234
    tracing user-defined Prolog predicates 213
directives
described 69
  fire_on_non_leaf 69
  profile 69
  trace 69
distributed rule base 12
events, logging 180, 211
event filters
    See event filters or change rule filters or timer rule filters 60
event sequence 132
examples
    correlation with event sequences 217
    rule builder 267
    executing a program 142, 144
    executing a task 147, 149
    flow control 118, 119, 120
    free, described 349
global variables, described 51
    See also global variables 51
ground, described 349
instantiated, described 349
language 9
launching a program 142, 144
launching a task 147, 149
naming conventions 49
parser grammar 53
plain rules 53
    See also plain rules 53
See also rule language predicates or Prolog predicates 10

See also Prolog 9

See regular expressions 188

See state correlation 7

See testing rules 236

See timer rules 47, 55

See also timer rules 47

See simple rules 11

See simple type 38

See slashes, converting 197

See state correlation 7

See upgrading from previous releases 243

See variables 51

See uppercase names 50

See writing to file 50

See event cache 9

See event conditions 57, 253

See event actions 57, 255

See severity attribute 106, 208

See severity, changing 106, 208

See severity, setting 208

Simple rules

creating a simple rule, rule builder procedure 249

described 11

event actions 57, 255

event conditions 57, 253

structure 57

simple_type 38

slashes, converting 197

software support

contacting xii

sort Prolog predicate 431

sprintf Prolog predicate 432

starting the event server 266

starting the rule builder 246

startsWith string 329

state correlation 7

aggregate values 291

allEvents mode 289

attributes, common 288

attributes, examples 313

CDATA tag 305

cloning 296, 308

collector rules 292
duplicates rules 289

eventType attribute 288

firstEvent mode 289

forwardEvents mode 290

id attribute 288

lastEvent mode 289

matching rules 288

messages 340

overview 283
state correlation (continued)
predicate library 312
rules 283
sample.xml file 283
state machines 283
summary action 285
syntax 305
TECSummary 285
threshold rules 289
thresholdCount attribute 290
timeInterval attribute 290
timeIntervalMode attribute 290
transitions 289, 290, 292
triggerMode attribute 289
value types 312
variable declaration 311
XML 283
state machines
definition 283
stateless rule
match 284
status attribute
changing the value 107
described 8
setting the value 209
status, changing 107, 209
status, setting 209
stopping the event server 266
string functions 329
string predicates 326
string set functions 337
string set predicates 335
String variable 311
string.concat 329
string.contains 328
string.eq 326
string.everyRuleId 331
string.ge 327
string.getEventType 331
string.gt 326
string.hourOfDay 331
string.iceq 328
string.icne 328
string.le 327
string.length 332
string.lt 327
string.month 332
string.monthString 333
string.ne 326
string.startsWith 329
string.substring 333
string.trim 334
stringset.areDisjoint 336
stringset.eq 335
stringset.equals 336
stringset.isMemberOf 336
stringset.isSubsetOf 336
stringset.join 337
stringset.ne 336
stringset.previousValues 338
strip Prolog predicate 438
subclass, described 40
subset Prolog predicate 436
substring Prolog predicate 437
substring string 333
subtract Prolog predicate 438
summary actions 283, 285
superclass checking 97, 98
superclass, described 40
target events, clearing 79, 83
targets, rule base
See rule base targets 14
tasks, executing from rules 147, 149
tec_agent_demo program 231
TEC_UMASK environment variable 32
tec.baroc file 18, 43
tecroot.xml file 283
tecsce.dtd file 283, 305
TECSummary action 285
template
See rule language predicates or Prolog predicates or
terms, Prolog 345
test data type and variable Prolog predicate 351
testing rules 231
See also rules, test 231
text editor, use with creating rules 243
time manipulation rule language predicates 75
timeInterval attribute 290
timeIntervalMode attribute 290
timeNow function 320
timer rule filters
described 66
timer_duration filter 66
timer_info filter 66
timer rules
described 10
processing of 47
structure 55
timer_duration filter 66
timer_info filter 66
timers
canceling 102, 103
described 10
setting a timer 212
state correlation 289, 292
timeSpan function 320
Tivoli Availability Intermediate Manager
rule base example 19
Tivoli Enterprise Console Rule Builder
See rule builder 243
Tivoli Software Information Center xi
toLowerCase string 334
toUpperCase string 333
trace 204, 234
trace directive 69
trace_it rule language predicate 213
tracing rule execution 69, 234
tracing user-defined Prolog predicates 213
transaction processing, rule engine 47
transitions 283, 289, 290, 292
trigger events 283
triggerMode attribute 289
trim string 334
troubleshooting
  state correlation 340
typeface conventions xiii

U
unification, described 349
union Prolog predicate 440
UNIX directory permissions for rule bases 32
unlink_from_cause rule language predicate 214
unlinking cause and effect events 214
update_event_activity rule language predicate 215
upgrade_gui.sh command 243
upgrading rule bases 13
upgrading rules from previous releases 243
user_predicates file 442
UTF-8 encoding 305
UTF-8 support 33, 44, 243

V
value types 312
variables
  _ (Prolog anonymous variable) 61
  _class 61
  _event 60
  environment variables, TEC_UMASK 32
  event filters, in 60
  global variables
    See global variables 51
    Prolog variables, described 345
    regular variables 51
    rules 50
    See also rules, variables 50
    scope in rules 51
    state correlation 311
    TEC_UMASK environment variable 32
variables, notation for xiv

W
where clause, event filter 63
wpostemsg command 6, 231
wrb command
  creating rule base targets 14
  creating rule bases 16
  usage recommendation 13
wrbudgrade command 13
write Prolog predicate 441

X
XML files 283, 305