Rule Builder’s Guide

Version 3.8
Note

Before using this information and the product it supports, read the information in “Notices” on page 429.
Contents

Preface .............................................. ix
Who Should Read This Guide .................... ix
Publications ........................................ ix
IBM Tivoli Enterprise Console Library ........ ix
Prerequisite Documents ........................... x
Related Documents ................................ x
Accessing Publications Online .................. xi
Providing Feedback about Publications ....... xi
Accessibility ......................................... xi
Contacting Customer Support .................... xi
Conventions Used in This Guide ................. xii
Typeface Conventions ............................... xii
Operating System-dependent Variables and
Paths .................................................... xii

Chapter 1. Rule Development
Fundamentals ........................................ 1
Designing Rules ..................................... 1
Events ................................................. 2
  Event Adapters ................................... 3
    Event Attributes Supplied by an Event Adapter
    Events Sent from Other Products or the Command
    Line .............................................. 5
  Event Filtering ................................... 5
Special Characters in Events ..................... 6
  Event Attribute Values Supplied by the Event Server
  Event Cache ...................................... 7
Rules ................................................. 8
  Rule Types ....................................... 8
General Rule Structure ............................. 10
  What Causes a Rule to Run ...................... 10
Example Actions You Can Take with Rules ..... 10
Rule Bases .......................................... 11
  Rule Base Targets ................................ 12
  Rule Sets and Rule Packs ....................... 13
  Rule Pack Example ................................ 16
  Modifying Rule Sets ............................. 17
Event Classes ...................................... 17
Predicates ......................................... 17
Rule Base Example ................................. 18
  Requirements for Rule Base Example .......... 18
  Assumptions for Rule Base Example .......... 18
  Procedure for Rule Base Example ............. 21
Rule Base Manipulation Procedures Using the
Rule Builder ....................................... 22
  Creating a New Rule Base ...................... 22
  Editing an Existing Rule Base ............... 24
  Listing All Known Rule Bases ................. 24
  Listing the Loaded Rule Base ................. 24
  Listing Rule Sets and BAROC Files in a
  Rule Base ...................................... 26
  Copying a Rule Base ............................ 27
  Deleting a Rule Base ............................ 28
  Importing a Rule Set into a Rule Base ....... 29
  Importing BAROC Files into a Rule Base ... 31
  Compiling a Rule Base ......................... 33
  Loading and Activating a Rule Base .......... 35
  Loaded and Active Rule Bases ................ 35
  UNIX Directory Permissions for Rule Bases. 36
  Testing, Debugging, and Analyzing Rules .... 36
  Creating Rules in International Environments 37
  Using Non-English Text in Rules ............. 37
  Passing Data to Tasks and Programs ......... 37
    Passing Data from exec_program and
    exec_program_local Predicates ............... 38
    Data for exec_task and exec_task_local
    Predicates ................................... 38
    UTF-8 Data and Local-Encoded Data
    Compatibility .................................. 38

Chapter 2. Event Class Concepts ............... 39
Designing Event Classes .......................... 39
Event Class Definitions .......................... 40
Event Attribute Definitions ...................... 41
  Attribute Names ................................ 42
  Attribute Data Types ........................... 42
  Attribute Facets ............................... 43
Inheritance ........................................ 44
  Inheritance Example ............................ 46
Enumeration Definitions .......................... 46
Design Recommendation for Using Enumerations
Instead of Event Class Definitions ............. 47
Overview of the root.baroc and tec.baroc Files . 47
BAROC File Parser Grammar ....................... 47
Re-Importing Modified BAROC Files ............. 48
BAROC Files in International Environments ... 48

Chapter 3. Rule Engine Concepts ............... 49
Forced Cleaning of the Event Cache ............. 49
Processing Events .................................. 49
  Processing Timer Expiration .................. 51
  Processing the Next Transaction .............. 51
Performance Tips ................................. 51

Chapter 4. Rule Language Reference .......... 53
Naming Conventions ............................... 53
Variables .......................................... 54
  Regular Variables ............................... 55
  Global Variables ............................... 55
    Manipulating Global Variables ............... 56
    Global Variable Example ..................... 57
Constants ......................................... 59
Comments ......................................... 59
Rule Structure ..................................... 60
  Rule Parser Grammar ............................ 60
  General Structure of a Plain Rule .......... 61
  Usage Notes .................................... 61
  Examples ....................................... 61
  General Structure of a Change Rule ......... 63
  Usage Notes .................................... 63

© Copyright IBM Corp. 2002
<table>
<thead>
<tr>
<th>Example</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Structure of a Timer Rule</td>
<td>65</td>
</tr>
<tr>
<td>Usage Notes</td>
<td>65</td>
</tr>
<tr>
<td>Examples</td>
<td>66</td>
</tr>
<tr>
<td>General Structure of a Simple Rule</td>
<td>67</td>
</tr>
<tr>
<td>Usage Notes</td>
<td>67</td>
</tr>
<tr>
<td>Example</td>
<td>68</td>
</tr>
<tr>
<td>General Structure of a Correlation Rule</td>
<td>69</td>
</tr>
<tr>
<td>Usage Notes</td>
<td>69</td>
</tr>
<tr>
<td>Example</td>
<td>69</td>
</tr>
<tr>
<td>Filters</td>
<td>71</td>
</tr>
<tr>
<td>Event Filters</td>
<td>71</td>
</tr>
<tr>
<td>Attribute Conditions</td>
<td>73</td>
</tr>
<tr>
<td>Change Rule Filters</td>
<td>74</td>
</tr>
<tr>
<td>Sender Filter</td>
<td>74</td>
</tr>
<tr>
<td>Attribute Change Filter</td>
<td>75</td>
</tr>
<tr>
<td>Timer Rule Filters</td>
<td>76</td>
</tr>
<tr>
<td>timer_info Filter</td>
<td>76</td>
</tr>
<tr>
<td>timer_duration Filter</td>
<td>77</td>
</tr>
<tr>
<td>Actions</td>
<td>78</td>
</tr>
<tr>
<td>Action Types</td>
<td>78</td>
</tr>
<tr>
<td>Directives</td>
<td>79</td>
</tr>
<tr>
<td>Rule Language Predicates</td>
<td>80</td>
</tr>
<tr>
<td>Rule Language Predicate Online Information</td>
<td>80</td>
</tr>
<tr>
<td>Quick Reference of Rule Language Predicates</td>
<td>80</td>
</tr>
<tr>
<td>Event Correlation</td>
<td>80</td>
</tr>
<tr>
<td>Event Cache Searching</td>
<td>81</td>
</tr>
<tr>
<td>Usage Notes</td>
<td>82</td>
</tr>
<tr>
<td>What Is a Duplicate Event?</td>
<td>82</td>
</tr>
<tr>
<td>Flow Control</td>
<td>83</td>
</tr>
<tr>
<td>BAROC Manipulation</td>
<td>83</td>
</tr>
<tr>
<td>Event Criteria</td>
<td>84</td>
</tr>
<tr>
<td>Event Activity</td>
<td>84</td>
</tr>
<tr>
<td>Thresholds</td>
<td>84</td>
</tr>
<tr>
<td>Regular Expressions</td>
<td>85</td>
</tr>
<tr>
<td>Global Variables</td>
<td>86</td>
</tr>
<tr>
<td>Time Manipulation</td>
<td>86</td>
</tr>
<tr>
<td>Launching Programs and Tasks</td>
<td>86</td>
</tr>
<tr>
<td>Attribute and Status Manipulation</td>
<td>87</td>
</tr>
<tr>
<td>General Purpose</td>
<td>87</td>
</tr>
<tr>
<td>Debug</td>
<td>88</td>
</tr>
<tr>
<td>Alphabetical Listing of Rule Language Predicates</td>
<td>88</td>
</tr>
<tr>
<td>add_to_repeat_count</td>
<td>89</td>
</tr>
<tr>
<td>all_clear_targets</td>
<td>90</td>
</tr>
<tr>
<td>all_duplicates</td>
<td>91</td>
</tr>
<tr>
<td>all_instances</td>
<td>93</td>
</tr>
<tr>
<td>any_clear_target</td>
<td>94</td>
</tr>
<tr>
<td>any_clearing_event</td>
<td>95</td>
</tr>
<tr>
<td>attr_condition</td>
<td>96</td>
</tr>
<tr>
<td>attr_exception</td>
<td>98</td>
</tr>
<tr>
<td>attr_sequence</td>
<td>99</td>
</tr>
<tr>
<td>bo_add_at_slotval_begin</td>
<td>100</td>
</tr>
<tr>
<td>bo_add_at_slotval_end</td>
<td>101</td>
</tr>
<tr>
<td>bo_get_class_of</td>
<td>102</td>
</tr>
<tr>
<td>bo_get_class_slots</td>
<td>103</td>
</tr>
<tr>
<td>bo_get_enum_options</td>
<td>105</td>
</tr>
<tr>
<td>bo_get_slotval</td>
<td>106</td>
</tr>
<tr>
<td>bo_is_defined_for_class</td>
<td>107</td>
</tr>
<tr>
<td>bo_is_direct_super_of</td>
<td>108</td>
</tr>
<tr>
<td>bo_is_super_of</td>
<td>109</td>
</tr>
<tr>
<td>bo_remove_from_slotval</td>
<td>110</td>
</tr>
<tr>
<td>bo_reset_default_slotval</td>
<td>112</td>
</tr>
<tr>
<td>bo_set_slotval</td>
<td>113</td>
</tr>
<tr>
<td>cancel_all_timers</td>
<td>114</td>
</tr>
<tr>
<td>cancel_timer</td>
<td>115</td>
</tr>
<tr>
<td>change_event_administrator</td>
<td>117</td>
</tr>
<tr>
<td>change_event_severity</td>
<td>118</td>
</tr>
<tr>
<td>change_event_status</td>
<td>119</td>
</tr>
<tr>
<td>check_all_thresholds</td>
<td>120</td>
</tr>
<tr>
<td>check_and_increment_count</td>
<td>121</td>
</tr>
<tr>
<td>check_event_criteria</td>
<td>123</td>
</tr>
<tr>
<td>check_threshold</td>
<td>125</td>
</tr>
<tr>
<td>clear_closed_events</td>
<td>127</td>
</tr>
<tr>
<td>clears</td>
<td>128</td>
</tr>
<tr>
<td>commit_action</td>
<td>130</td>
</tr>
<tr>
<td>commit_rule</td>
<td>131</td>
</tr>
<tr>
<td>commit_set</td>
<td>132</td>
</tr>
<tr>
<td>convert_ascii_time</td>
<td>133</td>
</tr>
<tr>
<td>convert_local_time</td>
<td>134</td>
</tr>
<tr>
<td>create_cache_search_criteria</td>
<td>136</td>
</tr>
<tr>
<td>create_clearing_event</td>
<td>138</td>
</tr>
<tr>
<td>create_event_criteria</td>
<td>141</td>
</tr>
<tr>
<td>create_event_sequence</td>
<td>144</td>
</tr>
<tr>
<td>create_threshold</td>
<td>149</td>
</tr>
<tr>
<td>decrement_slot</td>
<td>150</td>
</tr>
<tr>
<td>drop_change_request</td>
<td>151</td>
</tr>
<tr>
<td>drop_received_event</td>
<td>152</td>
</tr>
<tr>
<td>erase_globals</td>
<td>153</td>
</tr>
<tr>
<td>exec_program</td>
<td>154</td>
</tr>
<tr>
<td>exec_program_local</td>
<td>156</td>
</tr>
<tr>
<td>exec_task</td>
<td>159</td>
</tr>
<tr>
<td>exec_task_local</td>
<td>162</td>
</tr>
<tr>
<td>first_causal_event</td>
<td>166</td>
</tr>
<tr>
<td>first_duplicate</td>
<td>167</td>
</tr>
<tr>
<td>first_effect_event</td>
<td>169</td>
</tr>
<tr>
<td>first_instance</td>
<td>170</td>
</tr>
<tr>
<td>first_related_event</td>
<td>172</td>
</tr>
<tr>
<td>forward_event</td>
<td>174</td>
</tr>
<tr>
<td>generate_event</td>
<td>176</td>
</tr>
<tr>
<td>get_attributes</td>
<td>177</td>
</tr>
<tr>
<td>get_config_param</td>
<td>178</td>
</tr>
<tr>
<td>get_global_grp</td>
<td>180</td>
</tr>
<tr>
<td>get_global_var</td>
<td>181</td>
</tr>
<tr>
<td>get_gm_time</td>
<td>182</td>
</tr>
<tr>
<td>get_gm_time</td>
<td>183</td>
</tr>
<tr>
<td>get_local_time</td>
<td>184</td>
</tr>
<tr>
<td>get_time</td>
<td>185</td>
</tr>
<tr>
<td>global_exists</td>
<td>186</td>
</tr>
<tr>
<td>increment_slot</td>
<td>187</td>
</tr>
<tr>
<td>init_count</td>
<td>188</td>
</tr>
<tr>
<td>init_event_activity</td>
<td>190</td>
</tr>
<tr>
<td>ip_node_unreachable</td>
<td>192</td>
</tr>
<tr>
<td>is_clearing_event</td>
<td>193</td>
</tr>
<tr>
<td>link_effect_to_cause</td>
<td>194</td>
</tr>
<tr>
<td>loadGlobals</td>
<td>196</td>
</tr>
<tr>
<td>log_error</td>
<td>197</td>
</tr>
<tr>
<td>place_change_request</td>
<td>199</td>
</tr>
<tr>
<td>print_cache</td>
<td>200</td>
</tr>
<tr>
<td>print_class_tree</td>
<td>202</td>
</tr>
<tr>
<td>print_event_activity</td>
<td>203</td>
</tr>
<tr>
<td>re_after_match</td>
<td>205</td>
</tr>
</tbody>
</table>
Chapter 5. Correlation Examples . . . . 237
Recording Causal Relationships between Events 237
Special Cases when Recording Causal Relationships . . . . . 238
Multiple Events of the Same Class but Different Meanings . . . . 238
An Event Is Not Part of a Sequence . . . . 239
Using Causal Relationship Information . . . . 239
Cause Events . . . . . 239
Effect Events . . . . 240
Clearing Events . . . 240
Processing Events Based on Causal Relationships . . . . 241
Processing Cause Events . . . . 241
Processing Effect Events . . . . 241
Processing Clearing Events . . . 242
Event Correlation Rule Language Predicates . . . . 242
Defining Event Relationship Information . . . . 243
How Event Criteria Is Evaluated . . . . 243
Using Event Relationship Information to Find Related Events . . . . . 244
Putting It All Together . . . . 245
Creating the Event Relationship Information . . . . 245
The Rules for Processing the Events . . . . 246
The process_problem_events Rule . . . . 246
The process_clearing_event Rule . . . . 248
Alternatives . . . 249

Chapter 6. Testing, Tracing, and Profiling Rules . . . . . 253
Testing Rules . . . . . 253
Sending Events with Commands . . . . 253

Sending Events with the tec_agent_demo Program . . . . . . . . 253
Configuration Prerequisites for Using the tec_agent_demo Program . . . . 253
Using the tec_agent_demo Program . . . . 254
Sending Events with the SendEvents Program . . . . 254
Creating Event Files from Events in the Reception Log . . . . . . . . . . 255
Tracing Rules . . . . . 256
Trace Granularity . . . . 257
Tracing a Rule Set . . . . 257
Tracing a Rule . . . . 258
Trace Information . . . . . 258
Rule Tracing Example . . . . . 259
Tracing a Rule Event Filter . . . . . 259
Tracing Rule Actions . . . . 260
Tracing Multiple Solution Event Filters in Predicates . . . . . . . 261
Profiling Rules . . . . . 261
Profile Granularity . . . . 263
Profiling a Rule Set . . . . 263
Profiling a Rule . . . . 264

Chapter 7. Synchronization and Correlation with the NetView Integrated TCP/IP Services
Component . . . . . . . . . . . 265
Events Sent From the NetView Component . . . . . 265
Synchronization with the NetView Component . . . . 266
Synchronization of Acknowledged Events . . . . 266
Synchronization of Closed Events . . . . . 266
Clearing NetView Events . . . . . 267
Clearing TEC_ITS_SA_Status Events . . . . 267
Correlation with the NetView Component . . . . 268
Correlation of Network Interface Events with Node Status Events . . . . . 268
Correlation of Subnet Events with Router Status Events . . . . 268
Correlation of IBM Tivoli Switch Analyzer Status Events with Node, Router, and Interface Events . . . . . . . . . 269
Correlation of Non-Network Events and Network Events . . . . 269

Chapter 8. The Default Rule Base. . . . . . . . . . . 271
Configuration Rules . . . . . . . . . . . 272
A Word about Record Predicates . . . . . 272
The Startup Rule Set . . . . . . . . . 272
Redirect Prolog Messages to Files . . . . 272
Activate Detailed Debugging Output for Rule Tracing . . . . . . . 272
Prevent Display of Nested Predicates at an Event Console . . . . 273
Restoring the Event Cache . . . . 273
Event Activity . . . . . 273
Event Filtering . . . . 273
Event Thresholds . . . . 274
Heartbeat Monitoring . . . . 274
Maintenance . . . . 275
Correlation . . . . 276
Event Forwarding 276
Shutdown 277

Chapter 9. Creating Rules with the Graphical Rule Builder 279

Overview of the Rule Builder 279
Graphical Rule Builder Considerations for International Environments 279
Using the Rule Builder 280
Starting the Rule Builder 282
Editing an Existing Rule Base 283
Creating a New Rule Set 283
Editing a Rule Set 284
Creating a Simple Rule 285
Creating a Compound Rule 295
Saving a Rule Set 300
Compiling a Rule Base 301
Stopping and Restarting the Event Server 304

Rule Builder Examples 305
Defining an Escalation Policy 305
The Problem 305
The Solution 305
Get Event Information 305
Build the Rules 306
Monitoring for Potential Security Breaches on UNIX Endpoints 309
The Problem 309
The Solution 309
Modify Default Severity 309
Create the Rules 309
Using Distributed Monitoring as a Passive Monitor 312
The Problem 312
The Solution 312
Configure the Distributed Monitoring Monitors 312
Create the Rules 313
Linking Related Events 316
The Problem 316
The Solution 317

Appendix. Using Prolog in Rules 319
Prolog Predicate Online Information 319
Language Basics 319
Terms 319
Facts 320
Rules 320
The Knowledge Base 321
Comments 322
Data Types and Ranges 322
Data Type Mapping between BAROC and Prolog 323
An Example of Using Prolog in a Tivoli Enterprise Console Rule 324
Quick Reference of Prolog Predicates and Operators 325
Instantiation and Unification 325
Test Data Type and Variable Instantiation 326
Convert Data Types 326
Atom Manipulation 327
Standard Order Comparison 327
Functions 327
Miscellaneous 327
List Manipulation 328
Mathematical Expression Evaluation 328
Arithmetic Comparison 329
Mathematical Operators 329
Pointer Arithmetic 329
Miscellaneous 330
Knowledge Base 330
I/O 330

Alphabetic Listing of Built-In Predicates and Operators 330
;
+ (pointer offset addition) 346
– (pointer offset subtraction) 332
– (pointer subtraction) 334
– (sign reversal) 336
– (subtraction) 337
* 338
/ 339
// 340
@< 341
@=< 342
@> 343
@>= 344
\= 345
+(pointer offset addition) 346
< 348
= 349
=:= 350
=\= 351
=? 352
=< 353
== 354
=> 355
>= 356
abolish 357
append 358
ascii 359
assert 360
atom 362
atomic 363
atomlength 364
atomparg 365
atomconcat 366
atomtolist 367
compile 368
consult 370
delete 372
disjoint 373
empty_list 374
fclose 375
fclose 376
fopen 377
fopen 378
flisting 379
int_to_hex 380
integer 381
is 383

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_list</td>
<td>385</td>
</tr>
<tr>
<td>length</td>
<td>386</td>
</tr>
<tr>
<td>lowertoupper</td>
<td>387</td>
</tr>
<tr>
<td>member</td>
<td>388</td>
</tr>
<tr>
<td>name</td>
<td>390</td>
</tr>
<tr>
<td>nmember</td>
<td>391</td>
</tr>
<tr>
<td>nmembers</td>
<td>392</td>
</tr>
<tr>
<td>not</td>
<td>393</td>
</tr>
<tr>
<td>number</td>
<td>394</td>
</tr>
<tr>
<td>pointer</td>
<td>395</td>
</tr>
<tr>
<td>pointeroffset</td>
<td>396</td>
</tr>
<tr>
<td>pointertoatom</td>
<td>397</td>
</tr>
<tr>
<td>pointertoint</td>
<td>398</td>
</tr>
<tr>
<td>printf</td>
<td>399</td>
</tr>
<tr>
<td>read</td>
<td>401</td>
</tr>
<tr>
<td>readln</td>
<td>403</td>
</tr>
<tr>
<td>real (convert integer)</td>
<td>404</td>
</tr>
<tr>
<td>real (test real type)</td>
<td>405</td>
</tr>
<tr>
<td>realtoatom</td>
<td>406</td>
</tr>
<tr>
<td>reconsult</td>
<td>407</td>
</tr>
<tr>
<td>remove_dups</td>
<td>408</td>
</tr>
<tr>
<td>retract</td>
<td>409</td>
</tr>
<tr>
<td>round</td>
<td>410</td>
</tr>
<tr>
<td>remove</td>
<td>411</td>
</tr>
<tr>
<td>sort</td>
<td>412</td>
</tr>
<tr>
<td>sprintf</td>
<td>413</td>
</tr>
<tr>
<td>strip</td>
<td>415</td>
</tr>
<tr>
<td>subset</td>
<td>417</td>
</tr>
<tr>
<td>substring</td>
<td>418</td>
</tr>
<tr>
<td>subtract</td>
<td>419</td>
</tr>
<tr>
<td>term_type</td>
<td>420</td>
</tr>
<tr>
<td>union</td>
<td>421</td>
</tr>
<tr>
<td>write</td>
<td>422</td>
</tr>
<tr>
<td>Making Your Predicates and Facts Available to the Tivoli Enterprise Console Rule Engine</td>
<td>424</td>
</tr>
<tr>
<td>Loading Predicates and Facts from a File</td>
<td>424</td>
</tr>
<tr>
<td>Using the assert Predicate with a TEC_Start Event</td>
<td>425</td>
</tr>
<tr>
<td>Using the Prolog Interpreter</td>
<td>426</td>
</tr>
<tr>
<td>An Example of Using the Prolog Interpreter</td>
<td>427</td>
</tr>
<tr>
<td>Notices</td>
<td>429</td>
</tr>
<tr>
<td>Trademarks</td>
<td>431</td>
</tr>
<tr>
<td>Glossary</td>
<td>433</td>
</tr>
<tr>
<td>Index</td>
<td>439</td>
</tr>
</tbody>
</table>
Preface

The IBM Tivoli Enterprise Console® Rule Builder’s Guide provides information about developing rules for managing events with the IBM Tivoli Enterprise Console product. 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. Currently, you can develop more flexible rules by manually coding them instead of using the graphical rule builder.

Who Should Read This Guide

The target audience for this guide is system administrators who need to develop rules for automating the management of Tivoli Enterprise Console events received by the Tivoli Enterprise Console event server or Tivoli® Availability Intermediate Manager. Users of this guide should have knowledge of the following:

- UNIX® and Windows NT™ operating systems
- Shell programming
- Tivoli Enterprise Console functionality

Publications

This section lists publications in the IBM Tivoli Enterprise Console library and any other related documents. It also describes how to access Tivoli publications online, how to order Tivoli publications, and how to make comments on Tivoli publications.

IBM Tivoli Enterprise Console Library

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

- IBM Tivoli Enterprise Console Adapters Guide, GC32-0668
  Provides information about the currently available adapters.
- IBM Tivoli Event Integration Facility User’s Guide, GC32-0691
  Discusses how to develop your own event adapters that are tailored to your network environment and your specific needs. Additionally, the guide describes how to filter events at the source.
- IBM Tivoli Enterprise Console Installation Guide, GC32-0823
  Discusses how to install, upgrade, and remove IBM Tivoli Enterprise Console components.
- IBM Tivoli Enterprise Console Reference Manual, GC32-0666
  The IBM Tivoli Enterprise Console Reference Manual provides details about command-line commands applicable to using the IBM Tivoli Enterprise Console product, the predefined tasks shipped in the task library, and the environment variables available to tasks that execute with an event. It also includes an overview of event flow through the event server architecture.
  Discusses how to plan for and configure your event database environment and describes components, roles, and other general information specific to for using the IBM Tivoli Enterprise Console product.
Prerequisite Documents

You should be familiar with the following documents before attempting to use the Tivoli Enterprise Console product:

- **Tivoli Management Framework Planning and Installation Guide, Version 3.6**
- **Tivoli Management Framework User’s Guide, Version 3.6**
- **Tivoli Management Framework Reference Manual, Version 3.6**

These documents contain detailed information about the desktop, managed nodes, administrators, policy regions, profiles, notices, tasks, and scheduling.

- **IBM Tivoli Enterprise Console User’s Guide**

Provides procedures for using the Tivoli Enterprise Console product.

Related Documents

The following list identifies additional resources for developing rule-writing skills. These resources were available when the IBM Tivoli Enterprise Console Rule Builder’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.tivoli.com/services/education/](http://www.tivoli.com/services/education/) for additional information.

- **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 wwbmtiv@us.ibm.com for additional information.

- **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](http://www.orb-data.com).


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


Accessing Publications Online

Publications in the product libraries are included in PDF or HTML formats, or both, on the product CD. To access the publications using a Web browser, open the infocenter.html file, which is located in the appropriate publications directory on the product CD.

When IBM publishes an updated version of one or more online or hardcopy publications, they are posted to the Tivoli Information Center. You can access updated publications in the Tivoli Information Center from the following Tivoli Customer Support Web site: http://www.tivoli.com/support/documents/

The Tivoli Information Center contains the most recent version of the books in the product library in PDF or HTML formats, or both. Translated documents are also available for some products.

Note: If you print PDF documents on other than letter-sized paper, select the Fit to page check box in the Adobe Acrobat Print dialog (which is available when you click File --> Print) to ensure that the full dimensions of a letter-sized page are printed on the paper that you are using.

Providing Feedback about Publications

We are very interested in hearing about your experience with Tivoli products and documentation, and we welcome your suggestions for improvements. If you have comments or suggestions about our products and documentation, contact us in one of the following ways:

• Send an e-mail to pubs@tivoli.com.
• Complete our customer feedback survey at the following Web site: http://www.tivoli.com/support/survey/

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. For additional information, see the Accessibility Appendix in the IBM Tivoli Enterprise Console User’s Guide.

Contacting Customer Support

If you have a problem with any Tivoli product, you can contact Tivoli Customer Support. See the Tivoli Customer Support Handbook at the following Web site:

http://www.tivoli.com/support/handbook/

The handbook provides information about how to contact Tivoli Customer 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
• What information you should gather before contacting support
Conventions Used in This Guide

Typeface Conventions

The guide uses several typeface conventions for special terms and actions. These conventions have the following meaning:

**Bold**  Commands, keywords, file names, authorization roles, URLs, or other information that you must use literally appear in **bold**. Names of windows, dialogs, and other controls also appear in **bold**.

**Italics** Variables and values that you must provide appear in *italics*. Words and phrases that are emphasized also appear in *italics*, as do new terms defined in the text.

**Monospace**  Code examples, output, and system messages appear in a monospace font.

Operating System-dependent Variables and Paths

This guide uses the UNIX convention for specifying environment variables and for directory notation. When using the Windows NT command line, replace `$variable` with `%variable%` for environment variables and replace each forward slash (`/`) with a backslash (`\`) in directory paths.

**Note:** When using the bash shell on Windows NT, you can use the UNIX conventions.

This guide uses the backslash character (`\`) convention at the end of a line of example text to indicate that the text shown on the following line has wrapped due to space restrictions of the page. The example should be interpreted as being on one line.
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.

In an event management system, the status and performance of hardware and software components are continuously monitored by both active and passive means. For example, the Tivoli Distributed Monitoring product is active, while the monitoring of system log files is passive.

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 will be corrected when the root cause is corrected. For example, if an uninterruptible power supply (UPS) 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 use the rule templates provided in the default rule base. For more information about the default rule base, see Chapter 8, “The Default Rule Base” on page 271.

Chapter 5, “Correlation Examples” on page 237 contains examples of correlating events with rules. For more information about designing event classes, see “Designing Event Classes” on page 39.

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 Documents” on page x.

---

**Events**

The central unit of information for the IBM Tivoli Enterprise Console is the event. An event is 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 server.

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 User’s Guide 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. Values for other attributes can be supplied by the event adapter, but they are not required. Some 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 discussed in Chapter 2, “Event Class Concepts” on page 39. 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 6.

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 wtdumprl 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.

![Windows NT Event Log Entry](image1.png)

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 `wtdumprl` command to display the reception log).

![Tivoli-formatted Event](image2.png)

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

![Event Details](image3.png)
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 Distributed Monitoring, NetView Integrated TCP/IP Services, Security Management, and Workload Scheduler, among others.

You can send events from a shell command line using the `wpostemsg` and `postemsg` commands. These are mostly used for testing purposes but you can also use them in shell scripts. See the IBM Tivoli Enterprise Console Reference Manual 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 for at the event server. Filtering events closest to 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.
  - NetWare 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 product 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 only forwards significant events to a higher-level installation.
Special Characters in Events

The characters Ctrl+A and Ctrl+B can not 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**
An 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”. The re_split_event_id predicate is provided to parse this value in a rule. See “re_split_event_id” on page 213 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>189578295</td>
<td>Tec_Start</td>
</tr>
<tr>
<td>189578296</td>
<td>UPS_Fan_Down</td>
</tr>
<tr>
<td>189578296</td>
<td>universal_oserv</td>
</tr>
<tr>
<td>189578315</td>
<td>UPS_Temp_Degraded</td>
</tr>
<tr>
<td>189578420</td>
<td>NFS_No_Response</td>
</tr>
<tr>
<td>2189578296</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 80.

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 "Using Prolog in Rules" on page 319 provides an overview and reference material covering some useful Prolog features you can use for rule development. Also, "Related Documents" on page x 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 78 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 78 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 60.

```
rule_type: rule_name:
{
    description
    event_filter
    action1
    action2
    ...
}
```

The following figure shows an example of the structure of a plain rule.

![Rule Structure Diagram]

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 71.

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. When one of these events is closed, the state of the other automatically changes.
- 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 of discussion, the terms rule base and rule base target can be used interchangeably. Rule base targets are discussed more in [“Rule Base Targets” on page 12](#).

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 `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 `wrb` command or the rule builder to manipulate rule bases.

**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 Tivoli. See the `wrbupgrade` command in the *IBM Tivoli Enterprise Console Reference Manual* 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.

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 –imptgtrule 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 Reference Manual IBM Tivoli Enterprise Console
Reference Manual 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 12 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 Tivoli with the Default rule base. A default
rule set for the Tivoli Availability Intermediate Manager is also included with the
Default rule base, and an overview of it can be found in Chapter 8, “The Default
Rule Base” on page 271.

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 `–crtp` 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 `–lsrbpack` option with the `–detailed` suboption.

To list the rule sets in a rule base, you can use the `–lsrbrule` 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 Reference Manual* 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 -lsbrule Production
Rule Set files
--------------
ov_default.rls
log_default.rls
from_sem.rls
tecad_snaevent.rls
tecad_nv390fwd.rls
tecad_nv390msg.rls
```

Of the rule sets shown in the preceding list, the first three 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 nine 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 three rule sets into it.
   ```
   $ wrb -imprprule ov_default defaultRules Production
   $ wrb –imprprule log_default defaultRules \ Production
   $ wrb –imprprule from_sem 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 from_sem 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.
**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. 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 -delrbrule rule_set rule_base
   ```
3. Import the new file into the rule base.
   ```
   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.
   ```
   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 will be 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 Reference Manual for using the `wrb` command.

For additional information about BAROC files, see Chapter 2, “Event Class Concepts” on page 39.

**Predicates**

Like event classes, rule language predicates provided by Tivoli and predicates that you may 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 80.
Some of the Prolog built-in predicates can be used as building blocks to create your own predicates. The Prolog predicates are described in “Using Prolog in Rules” on page 319.

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 Reference Manual. 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 a 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 Tivoli will be 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.
**Procedure for Rule Base Example:** To create the Operations rule base:

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

2. Import classes into the rule base.

   ```
   wrb -imprbclass /data/TME/TEC/default_rb/ \\
   TEC_CLASSES/tecad_nt.baroc Operations
   ```

3. Import the rule sets into the rule base.

   ```
   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.

   ```
   $ wrb -lsrbrule Operations
   Rule Set files
   ---------------
   tec_server.rls
   aim_ops_perf.rls
   aim_ops_sec.rls
   aim_pers.rls
   aim_acct.rls
   ```

5. Create the rule pack in the rule base.

   ```
   wrb -crtrp 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 the *Tivoli Availability Intermediate Manager User's Guide* for the procedure.

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/.rtargets/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 Reference Manual.

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 Reference Manual 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 Tivoli. 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 may be easier to copy an existing rule base and modify it to begin creating a new rule base. See “Copying a Rule Base” on page 27 for additional information.

The following table provides the context and Tivoli authorization role required for this procedure:
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 press 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, press **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 9, “Creating Rules with the Graphical Rule Builder” on page 279 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.

Listing the Loaded Rule Base: You can list the rule base currently loaded on the event server. This may or may 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.
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 22 for additional information.
2. If you do not choose the Remove files in destination before copying option, the rule set file will be 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 will be 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.

![Import Into Rule Base dialog](image)

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**. Output similar to the following is displayed.

![Compile Rule Base dialog box](image)

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” 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.
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 perform analyses on 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 48 and “Graphical Rule Builder Considerations for International Environments” on page 279.

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, Version 3.6 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 `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 in the Tivoli Enterprise Console, Version 3.7 with the 3.7.0-TEC-004 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></td>
<td></td>
<td>UNIX</td>
<td>Windows</td>
</tr>
<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>Perl 5.5.x or earlier</td>
<td>Not 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>executable</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>Perl 5.5.x or earlier</td>
<td>Not 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>executable</td>
<td>Not 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’s class is compared to those stored in the active rule base. If the event class exists in the rule base, the event is accepted as valid by the event server. If the class does not exist in the rule base, 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.

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 1. 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 facet 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 a facet, 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 facet, 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_Volatage would then be defined as subclasses to this Cisco_Cat5000_Env_Envent; 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 44 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 6 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, ...;

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 46 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 46.
      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 396.

   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 79 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.

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 Reference Manual 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 :
  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;         
    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,       
    default = WARNING;       
    msg: STRING;              
    msg_catalog: STRING;      
    msg_index: INTEGER;       
    duration: INTEGER,        
      parse = no;            
    num_actions: INTEGER,     
      parse = no;            
  }
```
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;
  }

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;
  }

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
```
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.

**BAROC File Parser Grammar**

An HTML file containing the grammar for the BAROC file parser in Backus Naur Form (BNF) notation is located in two locations. The grammar can be found on the event server at the following location:

`$BINDIR/../generic_unix/TME/TEC/BOOKS/HTML/barocp.html`

The grammar can be found on the product CD in the following location:

`/BOOKS/HTML/barocp.html`
The grammar provides complete structure and syntax for the statements that can be contained within a BAROC file.

**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 desired. 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 will be 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, Version 3.6* 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 37.
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 execution 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 normally 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 \texttt{wpostmsg} or \texttt{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 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.

| Object       | Convention                                                                
<table>
<thead>
<tr>
<th></th>
<th></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 can begin with a lowercase or uppercase letter; however, a rule action name that begins with an uppercase letter must 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 54</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 401 and “write” on page 422 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'  
    ],  
    action: (  
        set_event_message(_event,'msg changed to %s',  
            [TEC_Start])  
    )  
).

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 'Maintenance_start'  
    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],
[[], [], []])
% Retrieve global variable values into regular values
% The default initialization values ('') are null.

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 will be set to that value.

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.

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.
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.

global_var(_groupkey[i], _key[o], _into[o])
Global Variable Example
To understand the usefulness of manipulating global variables, the following example shows a common scenario that ignores events from systems that have been flagged as being under maintenance (for example, someone is actively repairing them). This rule set assumes that when someone starts maintenance on a system, a Maintenance_start event is generated, and when they finish, a Maintenance_end event is generated. All events generated in between these two events are to be ignored; that is, they are dropped.

```plaintext
/********************************************************************
/**RULES **********************************************************/
----------------------------------------------------------------------
Restrictions: Maintenance_EOF is not generated
----------------------------------------------------------------------

/*
The following rule will be fired whenever the TEC restarts and will
set the maintenance mode for any systems with open Maintenance_start
events.
*/
rule:
maintenance_start:
  (
    event: _event of_class 'TEC_Start'
      where [],
    reception_action:
      (
        all_instances(event:
          _maintenance_ev of_class 'Maintenance_start'
            where [
              status: outside ['CLOSED'],
              origin: _origin
            ]
          ),
        set_global_var('Maintenance', _origin, 'on')
      )
  ).

/*
The following rule will turn maintenance mode on.
Note that the commit rule is necessary so that the check maintenance
mode won't drop this event.
*/
rule:
maintenance_started:
  (
    event: _event of_class 'Maintenance_start'
      where [
        origin: _origin
      ],
    reception_action:
      (
        set_global_var('Maintenance', _origin, 'on'),
        commit_rule
      )
  ).
```
The following rule will turn maintenance mode off.

```
rule:
maintenance_stopped:
{
    event: _event of_class 'Maintenance_stop'
    where [
        origin: _origin
    ],
    reception_action:
    {
        set_global_var('Maintenance', _origin, 'off'),
        drop_received_event,
        all_instances(event: _maint_start of_class 'Maintenance_start'
        where [
            origin: within [_origin ],
            status: outside ['CLOSED']
        ]),
        set_event_status(_maint_start, 'CLOSED'),
        commit_rule
    }
}.
```

The following rule will be fired whenever a new event is received to filter out hosts that are in maintenance mode.

```
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
    }
}.
```

/* End of maintenance rules */
### 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 8.

- 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:

```plaintext
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 71].

Examples

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

```plaintext
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 65) are evaluated.

```plaintext
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, '')
}
```

IBM Tivoli Enterprise Console: Rule Builder’s Guide
General Structure of a Change Rule

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

```plaintext
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 71.
- Sender filters and attribute change filters are described in “Change Rule Filters” on page 74.
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).

declare_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 - %s: %s tec_print "The following condition for printer \n%s has been fixed:
	%s",
                      [_new_status, _msg, _hostname, _msg],
                      'YES')

).
General Structure of a Timer Rule

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

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,  
  action: action_name: (  
    action_body  
  )  
).

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 71.
- Timer rule filters are described in “Timer Rule Filters” on page 76.
- Timers are set with the set_timer predicate, which is described on page 233. A plain rule example that sets a timer is shown in example 2 on page 62. A timer rule example that sets a timer is shown in example 2 on page 66.
- 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.
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 61 causes this rule to be evaluated.

   timer_rule: timer_print_assist : {
      event: _event of_class '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: change_event_severity(new_severity)</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: change_event_status(new_status)</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 repeat_count attribute of the existing event is incremented by one and the newly received event is dropped. Syntax: drop_duplicate_event(time_window)</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 [exec_program on page 154] 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>Event Action</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</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 159 for additional information about the arguments for this event action. The arguments for the exec_task rule language predicate are similar. Note: 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.</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 174 for additional information about the tec_forward.conf file.</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 227 for additional information about the arguments for this event action. The arguments for the set_event_message rule language predicate are similar. Note: 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.</td>
</tr>
</tbody>
</table>

- For a simple rule, names of event classes and attributes do not require single quote delimiters unless they have embedded spaces.
- Event filters are described in "Event Filters" on page 71.

**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).
```

- For a simple rule, names of event classes and attributes do not require single quote delimiters unless they have embedded spaces.
- Event filters are described in "Event Filters" on page 71.
**General Structure of a Correlation Rule**

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

```plaintext
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**
    Links 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.
correlation_rule: fan_fail_causes_temp_warn:

description: 'This rule links a fan failure event and a temperature warning event on the same computer.'

event_relation: Fan_Failure_Notification causes Temperature_Warning

within: 10 minutes

when: [
    Fan_Failure_Notification.hostname equals Temperature_Warning.hostname
]

correlation_rule: temp_warning_causes_temp_shutdown:

description: 'This rule links a temperature warning event and a temperature shutdown event on the same computer.'

event_relation: Temperature_Warning causes Temperature_Shutdown

within: 15 minutes

when: [
    Temperature_Warning.hostname equals Temperature_Shutdown.hostname
]
Filters

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.

\[
event: \_event \text{ of}_\text{class} '\text{Printer_Error_Cleared}'
\]

**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).

\[
\_\text{class within} [\text{class_name1},...]
\]

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 may 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:

\[
event: \_\text{event of}_\text{class}
  \text{within} ['\text{Printer_Paper_Out}',
  '\text{Printer_Toner_Low}',
  '\text{Printer_Offline}',
  '\text{Printer_Output_Full}',
  '\text{Printer_Paper_Jam}',
  '\text{Printer_Door_Open}']
\]

**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.

\[
event: \_\text{event of}_\text{class \_class}
  \text{within} ['\text{Printer_Paper_Out}',
  '\text{Printer_Toner_Low}',
  '\text{Printer_Offline}',
  '\text{Printer_Output_Full}',
  '\text{Printer_Paper_Jam}',
  '\text{Printer_Door_Open}']
\]
The following example shows the use of a variable in place of a literal LIST_OF value:

```plaintext
;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:

event: _event of_class
  outside ['Printer_Paper_Out', 'Printer_Toner_Low', 'Printer_Offline',
  'Printer_Output_Full', '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:

  ```plaintext
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
  ```

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

  ```plaintext
  where [status: equals 'OPEN'],
  ```

  ```plaintext
  outside
  Tests whether an attribute’s value is not equal to one of the values in a list. The following example tests whether the value of the status attribute is anything other than CLOSED:
  ```

  ```plaintext
  where [status: outside ['CLOSED']]
  ```

  ```plaintext
  within
  Tests whether an attribute’s value is equal to one of the values in a list. The following example tests whether the value of the status attribute is either OPEN or ACK and instantiates the _status variable with the value:
  ```

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

  **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.

  ```plaintext
  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.

  ```plaintext
  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,

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

The following list operators are valid in an attribute change filter:

outside Tests whether the requested change is not equal to one of the values in a list.

within Tests whether the requested change is equal to one of the values in a 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 may 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 6. 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

The following list operators are valid in a `timer_info` filter:

**outside**

Tests whether the timer information is not equal to one of the values in a list.
within
Tests whether the timer information is equal to one of the values in a 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:
  \[ \text{timer_info: equals 'Level 1',} \]
- The following timer_info filter succeeds if the timer information was specified as either Level 1 or Level 2. The \(_\text{timerinfo}\) variable is instantiated with the value of the timer information.
  \[ \text{timer_info: _timerinfo within ['Level 1','Level 2'],} \]
- The following timer_info filter simply instantiates the \(_\text{timerinfo}\) variable with the timer information:
  \[ \text{timer_info: _timerinfo,} \]

**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\_equals

The following list operators are valid in a timer_duration filter:

**outside**
Tests whether the timer duration is not equal to one of the values in a list.

**within**
Tests whether the timer duration is equal to one of the values in a 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:
  \[ \text{timer_duration: equals 60,} \]
- The following timer_duration filter succeeds if the timer was specified to last either 60 or 90 seconds. The \(_\text{timerduration}\) variable is instantiated with the value of the timer duration.
  \[ \text{timer_duration: _timerduration within ['60','90'],} \]
- The following timer_duration filter simply instantiates the \(_\text{timerduration}\) variable with the timer duration:
  \[ \text{timer_duration: _timerduration,} \]
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 67. Correlation rules use event relationships for their actions, as described in “General Structure of a Correlation Rule” on page 69.

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

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

- `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 216.

```ruby
rule: link_oserv_to_host: {
  description: 'Link the universal_oserv to universal_host if they are related',
  event: _event of_class 'universal_oserv' where [probe_arg: _probe_arg, severity: equals 'WARNING'],
  reception_action: 'oserv_script': {
    exec_program(_event, 'oserv_beep.sh','%s',[_probe_arg], 'YES')
  },
  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)
  }
}
```

**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
wrbc -comrules -profile command. Profiling is not enabled by default. See “Profiling Rules” on page 261 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 wrbc -comrules -trace command or using the
rule builder dialog. Tracing is not enabled by default. See “Tracing Rules”
on page 256 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
predicates that Tivoli has defined 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 53 contains information about
Prolog and how you can create your own predicates.

This section describes the predicates defined by Tivoli. 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.

Rule Language Predicate Online Information

Information in HTML format about each rule language predicate, 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

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
beginning on page add_to_repeat_count.

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 237 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>
<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 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 may 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 73 for additional information.

Normally, 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 usually 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 execute, you may 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><code>commit_action</code></td>
<td>Controls execution within the actions of the current rule.</td>
</tr>
<tr>
<td><code>commit_rule</code></td>
<td>Controls execution within a rule set.</td>
</tr>
<tr>
<td><code>commit_set</code></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><code>bo_add_at_slotval_begin</code></td>
<td>Adds an element to the beginning of an attribute’s list of values.</td>
</tr>
<tr>
<td><code>bo_add_at_slotval_end</code></td>
<td>Adds an element to the end of an attribute’s list of values.</td>
</tr>
<tr>
<td><code>bo_get_class_of</code></td>
<td>Gets the class of an event.</td>
</tr>
<tr>
<td><code>bo_get_class_slots</code></td>
<td>Gets attribute information for an event class.</td>
</tr>
<tr>
<td><code>bo_get_enum_options</code></td>
<td>Gets the elements of an enumeration.</td>
</tr>
<tr>
<td><code>bo_get_slotval</code></td>
<td>Gets the value of an attribute from an event.</td>
</tr>
<tr>
<td><code>bo_is_defined_for_class</code></td>
<td>Checks whether an attribute is defined for an event class.</td>
</tr>
<tr>
<td><code>bo_is_direct_super_of</code></td>
<td>Checks whether an event class is a direct superclass of another event class.</td>
</tr>
<tr>
<td><code>bo_is_super_of</code></td>
<td>Checks whether an event class is a superclass of another event class.</td>
</tr>
<tr>
<td><code>bo_remove_from_slotval</code></td>
<td>Removes a value from an attribute’s list of values.</td>
</tr>
<tr>
<td><code>bo_reset_default_slotval</code></td>
<td>Resets the attribute value for an event to the default value.</td>
</tr>
<tr>
<td>Predicate</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</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. The criteria is used by the check_event_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>
<tr>
<td>print_event_activity</td>
<td>Writes the event activity report defined with the init_event_activity predicate.</td>
</tr>
<tr>
<td>reset_event_activity</td>
<td>Resets the counts for all event reporting criteria to 0.</td>
</tr>
<tr>
<td>update_event_activity</td>
<td>Captures event information for reporting by the print_event_activity_report 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>check_all_thresholds</td>
<td>Applies criteria to events to determine if any threshold is exceeded.</td>
</tr>
<tr>
<td>check_threshold</td>
<td>Applies criteria to events to determine if a specific threshold is exceeded.</td>
</tr>
<tr>
<td>create_threshold</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 discussed in "Global Variables" on page 55.

The following table lists the predicates in this category.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase_globals</td>
<td>Removes all the global variables in a group from the knowledge base.</td>
</tr>
<tr>
<td>get_global_grp</td>
<td>Gets the value of all global variables in a group.</td>
</tr>
<tr>
<td>get_global_var</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>reset_global_grp</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>set_global_var</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>convert_ascii_time</td>
<td>Converts a time structure to an atom.</td>
</tr>
<tr>
<td>convert_local_time</td>
<td>Converts an epoch time number to a time structure in local system time.</td>
</tr>
<tr>
<td>convert_gm_time</td>
<td>Converts an epoch time number to a time structure in Greenwich mean time.</td>
</tr>
<tr>
<td>get_gm_time</td>
<td>Gets the current time represented in Greenwich mean time.</td>
</tr>
<tr>
<td>get_local_time</td>
<td>Gets the current local system time.</td>
</tr>
<tr>
<td>get_time</td>
<td>Gets the current time represented by an integer since the epoch.</td>
</tr>
<tr>
<td>resolve_time</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>exec_program</td>
<td>Launches a program.</td>
</tr>
<tr>
<td>exec_program_local</td>
<td>Launches a program on a local server.</td>
</tr>
<tr>
<td>exec_task</td>
<td>Launches a task from a task library.</td>
</tr>
<tr>
<td>exec_task_local</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>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 backslashes 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 normal and added to the event cache.

rule: printer_problem:
  event: _event of_class
    within ['Printer_Paper_Out',
    'Printer_Toner_Low',
    'Printer_Offline',
    'Printer_Output_Full',
    'Printer_Paper_Jam',
    'Printer_Door_Open'],
  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 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 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.

```c
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:**  
\[ \text{all_duplicates}( \_event, \text{event:} \_duplicate \text{ where } \text{attribute\_conditions}) \]

—OR—

\[ \text{all_duplicates}( \_event, \text{event:} \_duplicate \text{ where } \text{attribute\_conditions}, \_referenceEvent \text{– time\_before – time\_after}) \]

**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 82.

**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 \text{ where } \text{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 71 for additional information.
- \_referenceEvent \text{– time\_before – time\_after}
  The number of seconds before the reference event.
- \_referenceEvent \text{– time\_after}
  The number of seconds after 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_resp:

  event: \_event of_class 'OV\_NODE\_DOWN'
  where [severity: \_severity],

  action: dup_event_severity:
    \text{all_duplicates}(\_event, event: \_dup OV\_ev
    where [status: outside ['CLOSED'] ],
    \_event -300 -300 ),

    set_event_severity(_dup OV\_ev, _severity)
  )
).```
See Also: first_duplicate
**all_instances**
Succeeds once for each event that satisfies the specified class, attribute, and time window conditions.

**Synopsis:**
all_instances(event: _event of class class where attribute_conditions)

—OR—

all_instances(event: _event of class class where attribute_conditions, _referenceEvent
=time_before ~time_after)

**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:**

_**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:**_  _event of class class where attribute_conditions

Specifies an event filter for querying the event cache. _event_ is instantiated with a pointer to each event instance found. See “Event Filters” on page 71 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 action closes all NFS_SERVER_NOT_RESPONDING events whose server attribute has a value of Pascal:

```plaintext
action: (
  all_instances(event: _nfs_ev of class 'NFS_SERVER_NOT_RESPONDING'
    where [server: equals 'Pascal' ]),

  set_event_status (_nfs_ev, 'CLOSED')
)
```

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 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 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](https://example.com), [create_clearing_event](https://example.com), [create_event_sequence](https://example.com)
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 141 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) UPS and Distributed
Monitoring. APC UPS events use the hostname attribute to identify affected
components. 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.

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')
],
),

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 UPS and Distributed Monitoring. APC UPS events use the hostname attribute to identify affected components. 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.

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.

```plaintext
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.

```lisp
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.

```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].
    )
).
```

**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.

```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.
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 Tivoli use.
[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,YES,NO,''),  
 slot(sub_source,SINGLE,STRING,YES,NO,''),  
 slot(origin,SINGLE,STRING,YES,NO,''),  
 slot(sub_origin,SINGLE,STRING,YES,NO,''),  
 slot(hostname,SINGLE,STRING,YES,NO,''),  
 slot(adapter_host,SINGLE,STRING,YES,NO,''),  
 slot(date,SINGLE,STRING,YES,NO,''),  
 slot(status,SINGLE,STATUS,YES,NO,''),  
 slot(administrator,SINGLE,STRING,YES,NO,''),  
 slot(acl,LIST_OF,STRING,NO,NO,''),  
 slot(credibility,SINGLE,INTEGER,NO,NO,''),  
 slot(severity,SINGLE,SEVERITY,YES,NO,''),  
 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.

```
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:** \( \text{bo_is_defined_for_class}(\_\text{attribute}, \_\text{classname}) \)

**Description:** This predicate succeeds if attribute \_\text{attribute} is defined in class \_\text{classname}.

**Arguments:**

\_\text{attribute}  
The attribute to check.

\_\text{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.

```
rule: baroc_example: (  
    event: \_\text{event of class class},  
    \% class \text{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: 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.
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.

```prolog
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:

```
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 sent 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.

```rule
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 233 additional information.
- `_timer_info` The timer information. This value was specified when the timer was set. See "set_timer" on page 233 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 normally. 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 invalid 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:** `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:

```
check_all_thresholds('NT_Performance_Alert', _name, _count)
```

**See Also:** [check_threshold](#), [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.
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: 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 142 for the `create_event_criteria` predicate on page to the event under analysis.

   ```
   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.
_referencEvent 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.

   _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 an 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 141 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 normally 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 may 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
  ),
  commit_action,
  add_to_repeat_count(_dup_down_ev, 1),
  drop_received_event
},

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 may 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.

```plaintext
reception_action:
  {  
    all_instances(_event, 
      event: _dup_toner_ev 
      where [ 
        status: outside ['CLOSED'] 
      ], 
      _event = 600 - 600 
    ), 
    commit_rule, 
    add_to_repeat_count(_dup_toner_ev, 1), 
    drop_received_event 
  },

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 may 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.

```plaintext
reception_action:
    ( all_instances( _event,
        event: _dupper
        where [ status: outside ['CLOSED'] ]),
    drop_received_event,
commit_set,
    add_to_repeat_count(_dupper, 1) ),
```

**See Also:**  None.
**convert_ascii_time**

Converts a time structure to an atom.

**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.

```prolog
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
Converting 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.

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**

Converts an epoch time number to a time structure in local system time.

**Synopsis:** \texttt{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 \texttt{convert\_local\_time}. \_time\_local\_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\_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.

\texttt{get\_time(_time\_epoch),}
\texttt{bo\_set\_slotval(_event, time\_epoch, _time\_epoch),}
\texttt{convert\_local\_time(_time\_epoch, _time\_local\_struct),}
\texttt{convert\_ascii\_time(_time\_local\_struct, _time\_string),}
\texttt{bo\_set\_slotval(_event, time\_string, _time\_string)}

**See Also:** \texttt{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.
yes 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.

```python
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 normal 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 141 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 may assign the name of an affected machine to the probe_arg attribute of an event.

Other monitoring systems may 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 98 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 normally 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 may be absolute conditions specified, but there should at least be one 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.

   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.
create_clearing_event(
    'cpqDa3PhyDrvStatusChange',
    [[ 'cpqDaPhyDrvStatus', equals, 'OK' ]],
    '[cpqDa3PhyDrvStatusChange]',
    ['hostname', [ 'cpqDaPhyDrvStatus', not_equals, 'OK' ]],
    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. A criteria is created once using the create_event_criteria predicate, and is used where needed in any rule action by the check_event_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[197]
- 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 meaning 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 may find the information in "Attribute Conditions" on page[73] 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 less_than less_than_equal not_equals 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 136 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 to execute a leaf class.

   ```java
   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']
   ]
   )
   ``

   **Note:** This example is not a realistic definition. It is simply intended to show the various ways you can define attribute conditions.

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.

   ```java
   create_event_criteria(db_critical,
     'TEC_DB',
     yes,
     [['hostname', equals, 'DB_SRV'],
     ['severity', equals, 'CRITICAL']
   ]
   ```
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.

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 96.

See the attribute_conditions argument description for the "create_event_criteria" on page 141 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 98.

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>
<tr>
<td>attr_exception</td>
<td>Defines an attribute that must match a different attribute in other events in an event sequence.</td>
</tr>
</tbody>
</table>
**Predicate Description**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_sequence</td>
<td>Defines the values of an attribute that change due to a problem event’s position in an event sequence.</td>
</tr>
<tr>
<td>clears</td>
<td>Defines a clearing event for events in an event sequence.</td>
</tr>
</tbody>
</table>

**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 UPS and Distributed Monitoring. The UPS event sequence is illustrated in the figure on page 238.

   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 Distributed Monitoring. Each of the problem events in the sequence has a related clearing event defined with the clears predicate. The UPS events are related if the hostname attributes have the same value.

   The 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 UPS events). This requires mapping of the probe_arg attribute to the hostname attribute of the UPS 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 a UPS event if its severity is FATAL. Furthermore, a universal_host event is a clearing event only of 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.

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

Chapter 4. Rule Language Reference 145
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 238.

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.

create_event_sequence(
    ['cpqTape3PhyDrvStatusChange'],
    ['hostname', ['status', 'outside', ['CLOSED']]],
    attr_sequence('cpqTape3PhyDrvStatusChange',
        'cpqTapePhyDrvCondition'=['Degraded', 'Failed']),
    clears('cpqTape3PhyDrvStatusChange',
        [['cpqTapePhyDrvCondition', equals, 'OK'],
        ['cpqTape3PhyDrvStatusChange']],
        ['cpqTapePhyDrvCondition', equals, 'OK']]
)
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.

```plaintext
create_event_sequence(
  ['cpqHe3ThermalSystemFanDegraded',
   'cpqHe3ThermalSystemFanFailed',
   'cpqHe3ThermalTempDegraded',
   'cpqHe3ThermalTempFailed'],
  [hostname, ['status', equals, 'OPEN']],
  [
    clears('cpqHe3ThermalSystemFanOK',
           [],
           ['cpqHe3ThermalSystemFanDegraded',
            []],
    clears('cpqHe3ThermalTempOK',
           [],
           ['cpqHe3ThermalTempDegraded',
            []],
    cpqHe3ThermalSystemFanOK
  ]
)
```
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.

```java
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, and the requesting user is not Administrator, 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: equals operator(_operator),
  slot: status set_to _new_status within ['ACK', 'CLOSED'],
  action: (_operator \= 'Administrator',
    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.

```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 may 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 Reference Manual 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 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.

_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 quotes. If ‘YES’, the completion status command can be checked from the event console.

Examples: The following example shows a use of the predicate:

exec_program(_event,
   % Pass in the event pointer for access to
   % its environment variables.

   'scripts/send_notice',
   % Program path/name.

   '-m "%s" -s %s',

   154 IBM Tivoli Enterprise Console: Rule Builder’s Guide
The %s format specifiers of the _format_string argument are bound to the msg and severity attributes of the event. The send_notice program will be launched with four command line arguments, such as shown in following example:

```
send_notice -m "Su to root failed for Joe" -s CRITICAL
```

Note that double quotes 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_message**
  Output from the program. This attribute value 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 may 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 Reference Manual 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, Version 3.6 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 quotes). The example in “exec_program” on page 154 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 quotes.

'YES' Generate a TASK_COMPLETE event when the program finishes. This argument must be enclosed in single quotes.

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:
   • The program is given the name of list_tmpdir.
   • There are no additional arguments for the program’s command line.
   • 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',

Chapter 4. Rule Language Reference 157
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 Reference Manual*.

**Note:** Null arguments to `exec_task` may 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 Reference Manual* 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, Version 3.6* 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:

  `-l tasklibname -h hostname -a arg1 -a arg2...`

  The example in [“exec_program” on page 154](#) 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, Version 3.6* 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 quotes. 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 will be 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', '-1 "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 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_severityollection:_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 ADSMIncBackup -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 Reference Manual.

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_message
Output from the task. This attribute value 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 may 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 Reference Manual* 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, Version 3.6* 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 quotes). 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 154 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, Version 3.6* 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 quotes.

‘YES’ Generate a TASK_COMPLETE event when the task finishes. This argument must be enclosed in single quotes.

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.

rule: task_start: {
    event: _event of_class 'TEC_DB'
    where [ ],

    reception_action: start_it: (  
        exec_task_local({  
            'send_dbadmin,  
            _event,  
            'Send_Email',  
            '-l "T/EC Tasks" -h "stumpy" -a "%s"  
             -a "%s",  
             ['joe@company.com','joe@company.com'],  
             'YES'
        })
    )
}

rule: task_result: {
    event: _event of_class 'TASK_COMPLETE'
    where [task_name: _task_name equals 'send_dbadmin',  
      % Test task name. Assign task_name value to  
      % variable if passed.
      task_number: _task_num  
      % Assign task_number attribute value to  
      % variable.
    ],

    reception_action: process_task_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_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:**

`first_causal_event(_effect_event, _cause_event)`

—OR—

`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.

```
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:**  
\[
\text{first_duplicate(} _\text{event}, \text{event: } _\text{duplicate where attribute_conditions})
\]

---OR---

\[
\text{first_duplicate(} _\text{event}, \text{event: } _\text{duplicate where attribute_conditions, } _\text{referenceEvent} _\text{time_before } _\text{time_after})
\]

**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 82.

If the \text{time_before} and \text{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.
- \text{event: } _\text{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 71 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 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
  }
```

Chapter 4. Rule Language Reference 167
See Also: all_duplicates
**first_effect_event**
Sends 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.

```plaintext
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:  

\[
\text{first_instance(} \text{event: \_event of class class where attribute_conditions) }
\]

—OR—

\[
\text{first_instance(} \text{event: \_event of class class where attribute_conditions, \_referenceEvent \text{–time_before \text{–time_after})}
\]

Description:  Succeeds once for the first event that satisfies the specified class, attribute, and time window conditions.

Arguments:

\_referenceEvent

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

\text{event: \_event of class class where attribute_conditions}

Specifies an event filter. See 

“Event Filters” on page 71

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.

```c
rule: escalate: {
  description: 'escalate host down events when causing NFS problems',
  event: _event of class 'NFS_No_Response'
    where [server: _server],
  action: 'increase_sev': {
    first_instance(event: _down_ev of class 'universal_host'
      where [status: outside ['CLOSED'],
               probe_arg: equals _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:**

```plaintext
first_related_event(_referenceEvent, _related_event, _relation)
```

| —OR— |
|———|

```plaintext
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.

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:

```
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:

```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:

```prolog
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.

```none
#.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:** `get_global_grp(_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.
`get_global_grp('Maintenance', _key, _value),`

**See Also:** `get_globals`, `get_global_var`
**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 will be set to `_default` and `_default` will be 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]
**getGlobals**
Gets all global variables.

**Synopsis:**  
\[
\text{getGlobals}(\text{group}, \text{key}, \text{value})
\]

**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:
\[
\text{getGlobals}(\text{group}, \text{key}, \text{value})
\]

**See Also:** [getGlobalGrp](#) [getGlobalVar](#)
**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:

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:

```python
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 will be 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
),
```

IBM Tivoli Enterprise Console: Rule Builder’s Guide
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:

```template
rule: 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
=============================================================

<table>
<thead>
<tr>
<th>Count</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>849</td>
<td>TEC_Tick</td>
</tr>
<tr>
<td>848</td>
<td>TEC_DB</td>
</tr>
<tr>
<td>822</td>
<td>TEC_Notice</td>
</tr>
<tr>
<td>812</td>
<td>TEC_Error</td>
</tr>
</tbody>
</table>

=============================================================
Slot Summary
=============================================================

<table>
<thead>
<tr>
<th>Count</th>
<th>Slot Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>3332</td>
<td>status=OPEN</td>
</tr>
<tr>
<td>590</td>
<td>severity=MINOR</td>
</tr>
<tr>
<td>574</td>
<td>severity=WARNING</td>
</tr>
<tr>
<td>564</td>
<td>severity=CRITICAL</td>
</tr>
<tr>
<td>550</td>
<td>severity=UNKNOWN</td>
</tr>
<tr>
<td>544</td>
<td>severity=HARMLESS</td>
</tr>
<tr>
<td>510</td>
<td>severity=FATAL</td>
</tr>
<tr>
<td>12</td>
<td>hostname=midnight.austin.lab.tivoli.com</td>
</tr>
<tr>
<td>12</td>
<td>source=69.1.3.30</td>
</tr>
<tr>
<td>11</td>
<td>hostname=dhcp12-235.austin.lab.tivoli.com</td>
</tr>
<tr>
<td>11</td>
<td>source=69.1.12.235</td>
</tr>
<tr>
<td>11</td>
<td>hostname=stingray.austin.lab.tivoli.com</td>
</tr>
<tr>
<td>11</td>
<td>source=69.1.5.82</td>
</tr>
<tr>
<td>10</td>
<td>hostname=austin.lab.tivoli.com</td>
</tr>
<tr>
<td>10</td>
<td>source=69.1.1.6</td>
</tr>
</tbody>
</table>

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 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 Integrated TCP/IP Services component 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.

**Examples:**  
The following example rule uses this predicate to correlate TEC_ITS_UNREACHABLE events with TEC_ITS_SUBNET_STATUS events.

```rule
unreachable_correlate_subnet: ( event: _event of_class 'TEC_ITS_UNREACHABLE'  
where [ ip_unreachable: _ip_unreachable ],  
reception_action: unreachable_correlate_subnet: (  
ip_node_unreachable(_ip_unreachable, _subnet_event),  
(subevent_status(_event, 'CLOSED') ; true ) ) ) ) ) .
```

**See Also:**  
link_effect_to_cause
**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.

```plaintext
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:

```
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 Tivoli 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 233 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.

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 Æ writes the entire event cache
• With an event filter Æ 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.

```plaintext
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: None.
```
**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:** None.
**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 190.

```prolog
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:

```prolog
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 invalid. 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:

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:
```
re_create(test, 'h.*i')
¾ Create regular expression test.

re_search_string(test, 'chair')
¾ Search 'chair' using regular expression test.
¾ Succeeds, matches 'hai'.
```

**See Also:**  re_create
**re_send_event_conf**

Sends an event to a remote event server.

**Synopsis:**  \( \text{re_send_event_conf}(_\text{conf\_file}, _\text{event}) \)

**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 Reference Manual](#) 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 7](#). 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:

```prolog
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 may 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 may 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 may have been caused by the disk being full:

```plaintext
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:

```%
% 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:

```plaintext
timer_rule: reset_event_activity: {
    event: _event of_class _class
    where [],
    timer_info: equals 'Event Activity Report',
    timer_duration: _rep_freq,
    action: reset_activity: {
        % recorded(event_activity,active),
        % Line above used with im.rls (intermediate mgr. rules)
        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:  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.
reset_global_grp('Maintenance', 'off'),

See Also: 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 will be 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:

#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: bo_set_slotval, get_local_time
**save Globals**
Writes all global variables from a group to a file.

**Synopsis:** save Globals(_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:
save Globals('/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, 600, 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.

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 6 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 7.

  **Notes:**
  1. You cannot change a status of CLOSED.
  2. A change from ACK to OPEN status is invalid for the `new_status` argument.

**Examples:**  
The following example shows how to set the status attribute of the event under analysis to ACK:

```plaintext```
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:
```
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.

 froze_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: `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:

`predicate_name`
The name of the predicate to trace.

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 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_detailed_debugging`
**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:

```percival
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 1 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:

---

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 may be generated due to a loss of electrical power from the utility company.
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 may 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 238. 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 238, 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 238, 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 238, 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 238, 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”
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 241.

**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 241.

- 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 144 and "create_clearing_event" on page 138 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 Alphabetic Listing of Rule Language Predicates beginning on page 88.

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.

```plaintext
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'], []),
            clear('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']),
            clear('cpqTape3PhyDrvStatusChange',
                ['cpqTapePhyDrvCondition', equals, 'OK'],
                ['cpqTape3PhyDrvStatusChange'],
                [])
        ),

        create_clearing_event(
            'CiscoLinkUp', [], ['CiscoLinkDown', ['origin'], no],

        create_clearing_event(
            'cpqDa3PhyDrvStatusChange',
            ['hostname', ['cpqDaPhyDrvStatus', not_equals, 'OK'],
             ['cpqDa3PhyDrvStatusChange'],
             ['hostname', ['cpqDaPhyDrvStatus', equals, 'OK'],
             ['cpqDa3PhyDrvStatusChange'],
             ['hostname', ['cpqDaPhyDrvStatus', not_equals, 'OK'],
             no],
```
The 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 245, as well as any other events that require the same processing.

The 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 may have 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 may 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.

**Note:** The process_problem_events rule spans two pages.

```plaintext
rule: '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, 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_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', [_ev], no),
    exec_program(_ev, 'scripts/notify_admin.sh',
                 'admin_group', [], no),
    commit_set
).

The **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.

```
rule: 'process_clearing_event':(
    event: _clr_ev of_class 'EVENT',
    reception_action: 'clear_all_targets'(
        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” on page 248. 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 “The process_problem_events Rule” on page 246 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.
rule: 'process_clearing_event': (  
    event: _clr_ev of_class 'EVENT',
    reception_action: 'clear_all_targets'
      is_clearing_event(_ev),
      (  
        all_clear_targets(_clr_ev, _target, 3600, 0),
        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
    )
    ).
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 `wpostemsg` and `postemsg` commands
- The `tec_agent_demo` program
- The SendEvents program

Sending Events with Commands

You can test rules with the `wpostemsg` and `postemsg` command-line commands. These commands send events whose characteristics are specified as arguments. The `wpostemsg` command simulates an event from a TME adapter and uses Tivoli Management Framework services for communication with the event server. The `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 *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.

```
TEC_Start
NT_NAV_start
NT_NAV_stop
NT_Perf_Alert
TEC_Stop
```

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 254 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 255 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)
   ```bash
   export TEC_BIN_DIR=$BINDIR/TME/TEC
   ```
2. Enter the following command to send the events listed in the control file:
   ```bash
   $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 256 including the events_list and time_list files. Use the procedure in “Creating Event Files from Events in the Reception Log” on page 255 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.

```
$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.

```
wtdump rl | $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 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.
Next, 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:

```plaintext
directive: trace   % Start trace.
rule: rule1: ( ...
  ).   % End rule1.
rule: rule2: ( ...
  ).   % End rule2.
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:

```plaintext
rule: test_rule: ( 
directive: trace,
  event: _evt of_class within [ 'NT_NAV' ] where [],

  reception_action: action0:(
    drop_received_event
  )
).
```

257
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.
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:

- call condition
- call usage: _125
- exit usage: 95
- call hostname: _126
- exit hostname: another_host
- call hostname: another_host equals my_server
- fail hostname: another_host equals my_server
- 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.

- call action auto_ack
- call set_event_status(0x2c0b88, ACK)
- exit set_event_status(0x2c0b88, ACK)
- call set_event_administrator(0x2c0b88, john)
- exit set_event_administrator(0x2c0b88, john)
- exit action auto_ack
- call action page_administrator
- call exec_program(0x2c0b88, send_cpu_usage_to_pager, 'john my_server %d', [95], NO)
- exit exec_program(0x2c0b88, send_cpu_usage_to_pager, 'john my_server %d', [95], NO)
- 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(0x2c0ae0, 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:

```c
rule: test_rule:
    directive: profile,
    event: _evt of_class within ['NT_NAV'] where [],
    reception_action: action0:
        drop_received_event

```
Chapter 7. Synchronization and Correlation with the NetView Integrated TCP/IP Services Component

The NetView® Integrated TCP/IP Services component (hereafter referred to as the NetView component) monitors network resources and can generate IBM Tivoli Enterprise Console events about those resources. Whenever the status of a network resource changes, the NetView component sends the event server an event notifying it of the change.

The IBM Tivoli Enterprise Console product provides a rule set, netview.rls, that helps manage these events generated by the NetView component. The NetView component also receives and processes events from the IBM Tivoli Switch Analyzer product, some of which are sent to the event server. Events from the IBM Tivoli Switch Analyzer product convey information about low level network resources to the event server. These rules enable the correlation of these low level network failures with other network failures reported by NetView to find the root cause event.

By closing obsolete or effect events, these rules ensure that only relevant events from the NetView component remain open in the event server. These rules also send information about status changes to these events back to the NetView component to keep information displayed there synchronized with information displayed on the event console. This chapter is an overview of these rules.

For the particulars of each rule, see the comments in the netview.rls file. For information about NetView events or NetView event handling, see the NetView documentation library.

This rule set is active by default. Before using the rule set, create the NetView event groups using the wcrtnvgroups command. For more information, see the IBM Tivoli Enterprise Console Reference Manual and the wcreatnvgroups command.

This functionality is supported in NetView Integrated TCP/IP Services component, version 7.1.2 and higher.

Events Sent From the NetView Component

The following table lists and briefly explains the events that are received from the NetView component.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEC_ITS_INTERFACE_STATUS</td>
<td>The status associated with a network interface. Values are UP, DOWN, ADMIN_DOWN, and UNREACHABLE.</td>
</tr>
<tr>
<td>TEC_ITS_ISDN_STATUS</td>
<td>The status associated with a ISDN interface. Values are ACTIVE and DORMANT.</td>
</tr>
<tr>
<td>TEC_ITS_12_STATUS</td>
<td>This is an internal event sent by the NetView component to assist in the processing of TEC_ITS_SA_STATUS events. Values are UP, DOWN, and MARGINAL.</td>
</tr>
</tbody>
</table>
### Event Description

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEC_ITS_NODE_STATUS</td>
<td>The status associated with a network node. Values are UP, DOWN, and MARGINAL.</td>
</tr>
<tr>
<td>TEC_ITS_ROUTER_STATUS</td>
<td>The status associated with a network router. Values are UP, DOWN, and MARGINAL.</td>
</tr>
<tr>
<td>TEC_ITS_SA_STATUS</td>
<td>The status reported by the IBM Tivoli Switch Analyzer product. Values are ifDown, nodeDown, nodeMarginal, ifUp, ifUnmanaged, ifDeleted, nodeUp, nodeUnmanaged, nodeResolved, and nodeDeleted.</td>
</tr>
<tr>
<td>TEC_ITS_SNMPCOLLECT_THRESHOLD</td>
<td>Reports threshold violations and threshold resets (also called rearms) generated by the NetView SNMP collect daemon. Values are THRESHOLD_EXCEEDED, and REARMED.</td>
</tr>
<tr>
<td>TEC_ITS_SUBNET_CONNECTIVITY</td>
<td>The connectivity information associated with a subnet. Values are UNREACHABLE, and REACHABLEAGAIN.</td>
</tr>
</tbody>
</table>

### Synchronization with the NetView Component

Synchronization rules send the NetView component notification when an event that was originally received from the NetView component is closed or acknowledged in the IBM Tivoli Enterprise Console product. Therefore, the NetView display remains synchronized with the event console.

#### Synchronization of Acknowledged Events

When an event is acknowledged in the event console, that event is automatically acknowledged in the NetView component. The rules synchronize acknowledgements for the following events:

- TEC_ITS_INTERFACE_STATUS
- TEC_ITS_NODE_STATUS
- TEC_ITS_ROUTER_STATUS

To synchronize these events with the NetView component, these rules send SNMP traps to the Netview component whenever these events are acknowledged. Upon receiving the trap, the NetView component updates its display with an acknowledged status.

#### Synchronization of Closed Events

These rules will cause the following events that have been closed in the event console to immediately update their status at the NetView console:

- TEC_ITS_INTERFACE_STATUS
- TEC_ITS_NODE_STATUS
- TEC_ITS_ROUTER_STATUS

This is done by sending a trap to the NetView component which causes it to immediately poll the network object and then update the topology map accordingly.

For example, suppose an event console operator sees a TEC_ITS_NODE_STATUS event with a status of DOWN and sends someone to repair the device. When the
repair is completed, the operator closes the event. The operator knows that the event was closed only after the device was fixed, and so expects the node’s status to be updated in the NetView console to reflect this change. These rules cause that update to occur.

Clearing NetView Events

When NetView resource status events are sent to the IBM Tivoli Enterprise Console product, old status events about the same resource become obsolete. When such a new event is received, old status events in the event cache for the same resource are automatically closed.

For example, if a router becomes unreachable, the NetView component sends a TEC_ITS_ROUTER_STATUS event with a status of DOWN. Later, when the router becomes available again, the NetView component sends another TEC_ITS_ROUTER_STATUS event, this time with a status of UP. In this instance, the rules close the first TEC_ITS_ROUTER_STATUS event, because it is made obsolete by the second event. The events that can be automatically closed are as follows:

- TEC_ITS_INTERFACE_STATUS
- TEC_ITS_ISDN_STATUS
- TEC_ITS_SNMPCOLLECT_THRESHOLD_STATUS
- TEC_ITS_NODE_STATUS
- TEC_ITS_ROUTER_STATUS
- TEC_ITS_SUBNET_CONNECTIVITY
- TEC_ITS_L2_STATUS
- TEC_ITS_SA_STATUS

Clearing TEC_ITS_SA_Status Events

In addition to closing an old status event when a new TEC_ITS_SA_STATUS event is received for a resource, these rules also set the severity of the old event. To do this, the sastatus attributes of the new and cached events are compared. If they match, the obsolete event is closed and its severity is set. The severity of the event is based on the value of the sastatus attribute.

The following table shows the mapping between the sastatus attribute and the severity of the closed event:

<table>
<thead>
<tr>
<th>sastatus value</th>
<th>New Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifDown, nodeDown, nodeMarginal</td>
<td>CRITICAL</td>
</tr>
<tr>
<td>ifUp, nodeUp, nodeResolved</td>
<td>HARMLESS</td>
</tr>
<tr>
<td>ifUnmanaged, ifDeleted, nodeUnmanaged, nodeDeleted</td>
<td>WARNING</td>
</tr>
</tbody>
</table>

For example, suppose that the event server receives a TEC_ITS_SA_STATUS event with a sastatus value of ifDown. The cache is searched for another TEC_ITS_SA_STATUS event with a sastatus of ifDown. If such an event is found, the old TEC_ITS_SA_STATUS event is closed and is assigned a severity of CRITICAL.
Correlation with the NetView Component

When the availability of one network resource is affected by the availability of another resource, the event server might receive multiple events from the NetView component for one problem. These rules close effect events and link them to the cause events. This way, an effect event need not be processed separately from its cause event. For more information about correlation and linking events, see Chapter 5, “Correlation Examples” on page 237.

Correlation of Network Interface Events with Node Status Events

When network interfaces become unavailable, they can cause nodes or routers to fail. When this happens, the event server might receive events for both the interface failure and the node or router failure. Because this is really only one problem, the interface failure event is set as a cause event and the node or router failure events are closed as effect events.

The following events can be effect events of a TEC_INTERFACE_STATUS event with a status of DOWN or ADMIN_DOWN:

- TEC_ITS_NODE_STATUS event with a status of DOWN or MARGINAL
- TEC_ITS_ROUTER_STATUS event with a status of DOWN or MARGINAL

For example, as all of a router’s interfaces become unavailable over a period of time, the NetView component sends a TEC_ITS_INTERFACE_STATUS event with a status of DOWN for each interface as it becomes unavailable. Once all of the router’s interfaces are unavailable, this means that the router is unavailable. Therefore, the NetView component also sends a TEC_ITS_ROUTER_STATUS event with a status of DOWN. When the event server receives the router-down event, and the TEC_ITS_INTERFACE_STATUS event is still in the event cache, the TEC_ITS_INTERFACE_STATUS event is linked as a cause event of the TEC_ITS_ROUTER_STATUS event. The TEC_ITS_ROUTER_EVENT event is then closed. This automatic processing determines the root cause of the event and avoids treating the router and interface events as multiple problems.

Correlation of Subnet Events with Router Status Events

When routers become unreachable, the event server receives events for both the subnet being unreachable and the router failure. If multiple events for subnet and router failure are sent, these rules set the router event as the cause event and close the subnet event. That is, these rules link each TEC_ITS_ROUTER_STATUS event with status DOWN or MARGINAL as a cause event of TEC_ITS_SUBNET_CONNECTIVITY events with status UNREACHABLE.

These rules correlate all subnet unreachable events from the same NetView server to all TEC_ITS_ROUTER_STATUS events with status DOWN or MARGINAL from the same NetView server. To obtain further information about the cause of the problem, launch the NetView console.

For example, suppose that the NetView component detects a router failure while monitoring a subnet and sends a TEC_ITS_ROUTER_STATUS event to the event server. Because a router is unavailable, the status of the subnet becomes UNREACHABLE. Therefore, the NetView component sends a TEC_ITS_SUBNET_CONNECTIVITY event to the event server with a status of UNREACHABLE. When the correlation rules are evaluated, the TEC_ITS_SUBNET_CONNECTIVITY event is then closed and the router event is linked to the subnet event as its cause event.
Correlation of IBM Tivoli Switch Analyzer Status Events with Node, Router, and Interface Events

The IBM Tivoli Switch Analyzer product monitors and reports on low-level network failures, such as switches. When a network resource that is monitored by the IBM Tivoli Switch Analyzer product fails, the NetView component sends the event server a TEC_ITS_SA_STATUS event. Because low level network failures can cause other network devices to fail, the NetView component might also send events indicating an interface, router, or node failure. These rules identify the TEC_ITS_SA_STATUS event as the root cause of these failures and close the effect events.

For example, suppose that the IBM Tivoli Switch Analyzer product detects a switch failure and notifies the NetView component, which then sends a TEC_ITS_SA_STATUS event to the event server. Simultaneously, the NetView component detects a router failure and sends the event server a TEC_ITS_ROUTER_STATUS event. This event has a status of DOWN and the same hostname attribute value as the TEC_ITS_SA_STATUS event. The IBM Tivoli Enterprise Console sets the TEC_ITS_SA_STATUS event as the cause event and closes the router event as an effect event.

Correlation of Non-Network Events and Network Events

The netview.rls rule set correlates subnet status information from the NetView component with information from other applications monitoring network resources. If an event reporting that a resource is unavailable arrives after the NetView component reports that a subnet is unavailable, the subnet event is designated as the cause event, and the resource event is closed as the effect event.

For example, suppose that the IBM Tivoli Monitoring product is monitoring resources on a subnet. When these resources become unreachable, a Heartbeat_Unreachable event is sent from IBM Tivoli Monitoring to the event server. Simultaneously, the NetView component sends a TEC_ITS_SUBNET_CONNECTIVITY event with a status of UNREACHABLE. These rules compare the IP addresses of the unreachable resources with the subnet information and if they are in the subnet that is unreachable, the Heartbeat_Unreachable event is closed as an effect event and the TEC_ITS_SUBNET_CONNECTIVITY event is designated as the cause event.

For an event to be correlated with a TEC_ITS_SUBNET_CONNECTIVITY event, the event must be a subclass of a TEC_ITS_UNREACHABLE event and must contain the IP address of the failed network resource in the ip_unreachable attribute of the event.

By default, the IBM Tivoli Monitoring product, version 5.1 and higher, supports sending heartbeat failure events to the event server. This allows event console operators to see the correlation between application failures, detected by IBM Tivoli Distributed Monitoring product, and network failures, detected by the NetView component.
Chapter 8. The Default Rule Base

This chapter provides an overview of rule sets contained in the default rule base. In addition to the rule sets discussed here, the default rule base contains rule sets that were shipped in previous releases. These are included for backwards compatibility, and are not be discussed here.

Sample rule sets containing rules for the processing of events are included with the IBM Tivoli Enterprise Console default rule base. The rules provided are template rules, and need to be customized for your specific event management requirements. Editing of these rule sets must be done with a text editor instead of with the rule builder. The rule sets are commented with helpful examples, descriptions, and tips. Make use of the comments as they provide specific details for customizing the rule set.

Each rule set contains a configuration rule and an operation section. With the exception of the startup.rls file, the configuration rule configures parameters for the rules that follow in the operation section. The configuration rule in startup.rls configures global parameters, such as debugging and cache activation. Configuration rules are processed when the event server is started; that is, a configuration rule is triggered by a TEC_Start event.

The rules are grouped by function. Each group can be activated or deactivated independently of other groups. For example, you can activate event filtering rules and deactivate maintenance rules by activating or deactivating their respective rule sets.

The following table lists the function groupings and file names of the rule sets:

<table>
<thead>
<tr>
<th>Function</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>startup.rls</td>
</tr>
<tr>
<td>Event activity</td>
<td>event_activity.rls</td>
</tr>
<tr>
<td>Event filtering</td>
<td>event_filtering.rls</td>
</tr>
<tr>
<td>Heartbeat monitoring</td>
<td>heartbeat.rls</td>
</tr>
<tr>
<td>Maintenance</td>
<td>maintenance_mode.rls</td>
</tr>
<tr>
<td>Event thresholds</td>
<td>event_thresholds.rls</td>
</tr>
<tr>
<td>Correlation</td>
<td>correlation.rls</td>
</tr>
<tr>
<td>Forwarding</td>
<td>forwarding.rls</td>
</tr>
<tr>
<td>Shutdown</td>
<td>shutdown.rls</td>
</tr>
</tbody>
</table>

Note: The rule sets must be added to a rule base in the same order as they are listed above. Since rules are designed to be activated in a particular order, failure to add rule sets in this order can result in unexpected behavior.

Each operation section contains rules that perform processing based on the parameters set in the configuration section. To learn about the rules, browse the rule set in a text editor to analyze rule logic and read the comments associated with each rule. The remainder of this chapter mainly discusses the configuration rules.
Configuration Rules

Each configuration rule configures the parameters for its rule set. The predicates discussed in this topic are included in the rule sets in the proper locations.

A Word about Record Predicates

The rule sets in the default rule base make extensive use of Prolog built-in record predicates. You must to modify some of the arguments to these predicates in the configuration section of the rule set, but a short explanation of their purpose can be helpful in understanding the rule set. If you want detailed information about record predicates, refer either to the Prolog manuals mentioned in the Preface or your own selections. Record predicates are used to manipulate global data. The predicates and their basic uses are as follows:

<table>
<thead>
<tr>
<th>Record Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>Store data</td>
</tr>
<tr>
<td>rerecord</td>
<td>Update data</td>
</tr>
<tr>
<td>recorded</td>
<td>Retrieve data</td>
</tr>
<tr>
<td>erase</td>
<td>Delete data</td>
</tr>
</tbody>
</table>

The Startup Rule Set

The startup rule set configures global parameters.

The possible configuration actions are as follows:
- Redirect Prolog messages to files
- Activate detailed debugging output for rule tracing
- Display nested predicate errors only at an event console
- Restore the event cache

Redirect Prolog Messages to Files

Provide paths and file names to the predicates shown in the following example. The tell_err predicate redirects Prolog error messages typically written to stderr to a file. The tell predicate redirects Prolog messages typically written to stdout to a file.

```prolog
tell_err('/tmp/prolog.err'),
% Route stderr to file

tell('/tmp/out'),
% Route stdout to a file
```

Activate Detailed Debugging Output for Rule Tracing

Set this parameter to the value on to provide additional debugging information in the rule trace file; otherwise, set it to the value off. Many of the newer predicates included with the Tivoli Enterprise Console product have tracing statements within them so that you can analyze why a predicate at a nested level (called from within the execution of an outer level predicate) has succeeded or failed. Older predicates provide trace information only when entering and exiting a top-level predicate. Be sure to enable rule tracing so trace information is generated to the rule trace file. The following example shows how to activate detailed debugging:

```prolog
set_detailed_debugging(on),
```
Prevent Display of Nested Predicates at an Event Console
This rule is to help with rule development. The drop_internal_error_event rule prevents TEC_Error events, generated from errors by nesting predicates, from being displayed at an event console. No action is performed on the events.

For example, a TEC_Error event occurs when the server detects abnormal conditions caused by the rules, such as when a non-existent predicate has been called. Creating rules based on these types of events is useful to a customer who is developing rules in a test environment. During testing, the drop_internal_error_event rule is disabled to allow the customer’s diagnostic rules to fire. When these rules move into a production environment, the drop_internal_error_event is enabled, preventing internal TEC_Error events from showing in the console.

To enable this rule, uncomment the rule in the netview.rls file. After doing this, the rule base must be recompiled and reloaded.

Restoring the Event Cache
You can reload old cached events at startup by using the restore_event_cache predicate. Since the predicate can fail, a Prolog safeguard is used to ensure that this does not disrupt the flow of the rule. The following example shows how to restore an event cache:

```
get_execution_mode(_mode),
(  _mode \== 'tec_server_mode' -> % Restores the cache if running on
    % a TEC event server
    restore_event_cache;
    true
)
```

Event Activity
With these parameters, you can configure an event activity report. You can specify items such as:
- How often to write the report
- The path name and file name for the report
- Events to exclude from the report
- Attribute criteria to include in the report
- Thresholds; for example, do not include event counts of less than 5 in the report

These parameters are actually arguments to the init_event_activity predicate, which is described in detail on page 190.

Event Filtering
With these parameters, you can define the criteria for determining the state of an event. These parameters are actually arguments to the create_event_criteria predicate, which is described in detail on page 141.

The event criteria must be specified in the second argument of the record predicate. For example, if you defined a criteria named harmless_maintenance and a criteria named harmless_heartbeat, you would need to specify them as shown in the following example:

```
record(event_filter_criteria,  
    [harmless_maintenance,  
      harmless_heartbeat])
```
Event Thresholds

With these parameters, you can define criteria for querying the event cache to check whether a certain number of events have been received within a time window.

These parameters are actually arguments to the predicates listed in the following table.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Described in Detail on Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_cache_search_criteria</td>
<td>136</td>
</tr>
<tr>
<td>create_event_criteria</td>
<td>141</td>
</tr>
<tr>
<td>create_threshold</td>
<td>149</td>
</tr>
</tbody>
</table>

These three parameters are required because the create_threshold predicate requires the name of a search that has been defined by the create_cache_search_criteria predicate. Similarly, the create_cache_search_criteria predicate requires the name of a criteria that has been defined by the create_event_criteria predicate. The following example shows how a threshold criteria is defined.

First, an event criteria is created with the create_event_criteria predicate.

```plaintext
create_event_criteria(all_critical, % Name of criteria
  'EVENT', % Class
  yes, % Fire on non-leaf
  [['severity', equals, 'CRITICAL']] % Attribute conditions
),
```

Then, a cache search criteria is created. The following creates a cache search that searches for an event whose severity is CRITICAL.

```plaintext
create_cache_search_criteria(critical_severity_search, % Name of the search
  all_critical, % Event criteria to use
  [hostname], % These attribute values must
  no % Compare duplicate detect slots
),
```

Finally, a threshold criteria is created, using the cache search that was previously defined. If more than five CRITICAL events are in the event cache within a window of one minute, this threshold is exceeded. If this threshold remains exceeded for more than 300 seconds, it is reported again as exceeded.

```plaintext
create_threshold(critical_threshold, % Name of threshold
  critical_severity_search, % Cache search to use
  60, % Reception period (in seconds)
  5, % Event threshold count
  300 % Maximum reporting frequency
),
```

Heartbeat Monitoring

These parameters specify how often to expect a pulse in the form of a TEC_Heartbeat event from a host. They also specify what severity to assign a TEC_Heartbeat internally-generated event when a pulse is missed. Each monitored host must systematically send a TEC_Heartbeat event to the event server where the pulse is being monitored.
The level attribute in a TEC_Heartbeat event determines how often the pulse is checked. The level attribute is defined as an ENUMERATION type in the tec.baroc file with the following possible values: ONE, TWO, THREE, FOUR, and FIVE. The default is FIVE. If more levels are needed, you must modify the tec.baroc file and add the new levels to the heartbeat monitoring configuration section.

You assign a monitoring frequency, in seconds, for each level. Additionally, you assign a value for the severity attribute of any TEC_Heartbeat event generated when a pulse is missed by the event server doing the heartbeat monitoring.

The following example shows how to configure these parameters:

record(heartbeat,'ONE',[60,'FATAL'])),
% Level ONE, check pulse every 60 seconds, FATAL severity
% if pulse missed.

record(heartbeat,'TWO',[300,'CRITICAL'])),
% Level TWO, check pulse every 300 seconds, CRITICAL severity if pulse missed.

record(heartbeat,'THREE',[600,'MINOR'])),
% Level THREE, check pulse every 600 seconds, MINOR severity if pulse missed.

record(heartbeat,'FOUR',[1800,'WARNING'])),
% Level FOUR, check pulse every 1800 seconds, WARNING severity if pulse missed.

record(heartbeat,'FIVE',[3600,'HARMLESS'])),
% Level FIVE, check pulse every 3600 seconds, HARMLESS severity if pulse missed.

**Maintenance**

Maintenance mode means that a host is undergoing activities such as software updates, restarts, power-off/power-on sequences, and so forth. These activities can cause events to be generated that you do not want the rule engine to process. Placing a host in maintenance mode causes the rule engine to ignore all events for the host, except those that are of the class TEC_Maintenance.

With these parameters, you can specify how long a host should remain in maintenance mode before generating an internal event of class TEC_Maintenance with the mode_status attribute set to a value of Timeout. Additionally, you can assign a value for the severity attribute of any TEC_Maintenance internal event that is generated when the host has been in maintenance mode for too long.

Each host going into maintenance mode must send a TEC_Maintenance event with the current_mode attribute set to a value of ON to the event server. A host can be taken out of maintenance mode by sending a TEC_Maintenance event with the current_mode attribute set to a value of OFF.

**Note:** When the timer for a host in maintenance mode expires, the host remains in maintenance mode until explicitly taken out of maintenance mode with a TEC_Maintenance event whose current_mode attribute is set to a value of OFF. The internal TEC_Maintenance event whose mode_status attribute is set to a value of Timeout is simply an informational event.

To specify the time, in seconds, that a host should be in maintenance mode, assign a value to the max_time argument, as shown in the following example:
record(maintenance_mode, max_time, 2600),
% Set maintenance mode to 1 hour.

to specify the severity for the TEC_Maintenance event generated when a host has been in maintenance mode past its defined time, assign a value to the severity argument, as shown in the following example:
record(maintenance_mode, severity, 'WARNING'),
% Set severity for event generated when a host has been in maintenance mode past max_time.

**Correlation**

With these parameters, you can define logical relationships between events. The examples shown in this section define relationships between events that can be generated from an APC power supply source, and are used by the rules defined in the correlation operation section. You can modify the existing examples or add new entries to this section for relationships you create.

**Note:** In order to run this example, include the APC event classes defined in the following file:

```plaintext
.../TEC/samples/correlation/apc_ups/apc.baroc
```

These parameters are actually arguments to the create_event_sequence and create_clearing_event predicates, which are described in detail on pages 144 and 138, respectively. The following is an example of how to use these predicates to define correlation logic.

```plaintext
% commLost %
% This defines a clearing relationship between the communicationLost event and the communicationsEstablished event
create_clearing_event('communicationEstablished', % clearing event
[ ], % clearing conditions
[ 'communicationLost' ], % cleared event
[ hostname ] ), % cleared attribute conditions
%

% upsSupplyingPower %
% This defines a cause-effect correlation sequence between the upsOnBattery, lowBattery, and upsDischarged events.
% cause chain of events
create_event_sequence([ 'upsOnBattery', 'lowBattery', 'upsDischarged' ],
% attribute conditions for the cause chain of events
[ hostname, [ 'status', not_equals, 'CLOSED' ] ],
[ % clearing relationships in the cause chain of events
clears('powerRestored', [ ], [ 'upsOnBattery' ], [ ] ),
clears('returnFromLowBattery', [ ], [ 'lowBattery' ], [ ] ),
clears('dischargeCleared', [ ], [ 'upsDischarged' ], [ ] ),
]),
```

**Event Forwarding**

There is one rule in this group and a configuration parameter to activate or deactivate this function. The rule forwards all events that have not been dropped (except TEC_Start, TEC_Maintenance, and TEC_Stop events which are not forwarded) to the event server specified in the tec_forward.conf file. This rule uses the forward_event predicate. See [forward_event](#) on page 174 for additional information about configuring the tec_forward.conf file.
Shutdown

This rule set is provided as a placeholder and can be used to add functions that are performed when the TEC_Stop event is received. Typical functions added include clean up and maintenance actions, such as file logging, the removal of debugging information, preparing the event cache for when the rules engine restarts, or closing a database.
Chapter 9. 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 –imptgrule` 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 Reference Manual 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, Version 3.6 for additional information about the
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.
Callout | Location for Task Information
--- | ---
1 | “Starting the Rule Builder” on page 282
2 | “Creating a New Rule Base” on page 22
3 | “Editing an Existing Rule Base” on page 283
4 | “Creating a New Rule Set” on page 283
5 | “Saving a Rule Set” on page 300
6 | “Editing a Rule Set” on page 284
7 | “Creating a Simple Rule” on page 285
8 | 
9 | 
A | 
B |
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, Version 3.6 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]

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 13.

![New Rule Set Creation](image)
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.

![Image of T/EC Rule Base window with new rule set structure](image1.png)

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 rule set name selected](image2.png)

**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 79 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:

![Simple Rule window with selected event class](image)

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.

![Simple Rule window with description](image)

3. Use the following procedure to enter the event class. Event classes are discussed in Chapter 2, “Event Class Concepts” on page 39.

   a. From the Simple Rule Window, click Event Class to display the Select Class window.

   ![Select Class window](image)

   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 will be 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.

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 discussed in “Attribute Conditions” on page 73. 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.

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.

The two distinct parts of a rule action are as follows:

<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>
<tr>
<td>Rule Action</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Action(s)</td>
<td>Specifies what to do when the event filter conditions are met.</td>
</tr>
</tbody>
</table>

**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 174 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. 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 desired task 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.</td>
</tr>
</tbody>
</table>

![Task Selection Window](image)
<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.</td>
</tr>
</tbody>
</table>

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.

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.

![Edit Arguments window](image)

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, Version 3.6* 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.

![Simple Rule window](image)
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 cannot 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.

   ![Compound Rule: New Compound Rule window](image1)

   Type a brief descriptive comment about the rule in the Description text field. This description is used to identify the rule in other dialogs.

   ![Compound Rule: New Compound Rule window](image2)

2. Use the following procedure to enter the event classes. Event classes are discussed in Chapter 2, “Event Class Concepts” on page 39.

   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.

   ![Select Class window](image3)
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-cancelled-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.
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 may 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.

e. 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.

f. 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.
g. Click **Add** to apply the changes and show the updated part of the rule in the **Correlation Synopsis** field.

![Correlation Rule: New Compound Rule](image1.png)

h. Repeat steps e, f, and g for each comparison that is needed.

i. Click **OK** to apply these changes and display the updated **Compound Rule window**. The rule being created is shown in the **Rule Synopsis** field.

![Compound Rule: New Compound Rule](image2.png)

j. 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.

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.

4. Click **Yes** to save the rule set and return to the Event Server Rule Bases window.

### 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 256 for additional information.
2. Click **Compile**. Output similar to the following is displayed.
3. Click Close to display the Event Server Rule Bases window.

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.

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

**The 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 Tivoli’s 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.

**The 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 Reference Manual* describes the tasks shipped by Tivoli 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.

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.

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

The 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.
```

The 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:
   When event is received

Actions:
   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:
   When event is... received

Actions:
   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

The 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.*

The Solution

**Configure the Distributed Monitoring Monitors:** The first step to implement this policy is configuring the Distributed Monitoring monitors to send the desired 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 an 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>
<tr>
<td>Attribute</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sub_source</td>
<td>Policy region containing the Distributed Monitoring profile</td>
</tr>
<tr>
<td>collection</td>
<td>Name of the Distributed Monitoring profile</td>
</tr>
<tr>
<td>monitor</td>
<td>Name of the individual monitor</td>
</tr>
<tr>
<td>probe_arg</td>
<td>Arguments passed to the monitor program</td>
</tr>
</tbody>
</table>

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 Pane](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 executes 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:

![Rule Set Screen](image)

**Linking Related Events**

**The Problem**

If a node goes down, you get multiple events from the HP OpenView adapter. 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.*
The Solution
Something that makes this problem potentially difficult to solve is the fact that the events may 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:

[OV_Node_Down] causes [OV_IF_Down] within a 10 minute period if:

[OV_Node_Down].status equals [OV_IF_Down].status
[OV_Node_Down].origin equals [OV_IF_Down].origin

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.

The HP OpenView adapter is configured by default to send in an event of class OV_Node_Down with a status of CLOSED whenever the node comes back up. This produces the preferred effect of automatically closing all of the OV_Node_Down events for that system due to an internal rule in the rule engine. Because of this rule, the associated OV_IF_Down events also close.
Appendix. 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 Documents" on page x 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):

```
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:

```
['Austin', 'Dallas', 'Houston']
```

An empty list, or null list, is written as follows:

```
[]
```

**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:

```
city('Austin')
```

The knowledge that Austin is a city in the state of Texas can be represented by the following fact:

```
in_state('Austin', 'Texas')
```

The knowledge that Texas is a state in the USA can be represented by the following fact:

```
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:

```
in_usa(_city) :-
    city(_city),
    in_state(_city, _state),
    in_country(_state, 'USA').
```

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.

The 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:

```prolog
assert(city('Austin'))
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:

```prolog
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:

```prolog
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:

```prolog
in_usa('Austin')
```

The predicate `in_usa('Austin')` runs the following queries in the order shown:

```prolog
city(_city)
in_state(_city, _state)
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 `in_state('Austin', 'Texas')` and the value Texas is assigned to the _state variable, which can then be used in subsequent queries.

```prolog
in_state('Austin', _state)
```

**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.

**Data Types and Ranges**

The following list describes the range for each Prolog data type:

- **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.

- **Atomic**
  - A general term that refers to an integer, real, pointer, or an atom data type.

- **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 `base'number`. For example, 2'111011, 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.

- **List**
  - The length of a list is limited only by the available memory of the heap.

- **Number**
  - A general term that refers to either an integer or real data type.

- **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.

- **Predicate**
  - The maximum arity of a predicate is 64.

- **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 ±2.2250738585072014e-308 through ±1.79769313486231570e+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>Simple</td>
<td></td>
</tr>
<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>Complex</td>
<td></td>
</tr>
<tr>
<td>LIST</td>
<td>list</td>
</tr>
</tbody>
</table>
An 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 325 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>
<tbody>
<tr>
<td>==</td>
<td>Term inequality operator</td>
</tr>
<tr>
<td>=</td>
<td>Term unification operator</td>
</tr>
<tr>
<td>^=?</td>
<td>Expression evaluation</td>
</tr>
<tr>
<td>==</td>
<td>Term equality operator</td>
</tr>
<tr>
<td>not</td>
<td>Term negation predicate</td>
</tr>
</tbody>
</table>

**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>intoatom</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>realtoatom</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>–</td>
<td>Subtract 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**

`_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`
  
  A number. Must be integer or real.

- `_number2`
  
  A number. Must be integer or real.

**Examples**

The following example shows various uses of the predicate:

- `_integer1` is 100,
- `_integer2` is 30,
- `_real1` is 2.1,
  
  % Assign values.

- `_sum1` is `_integer1 + _integer2`,
  
  % `_sum1` is unified with 130.

- `_sum2` is `_real1 + _integer2`
  
  % `_sum2` is unified with 3.210000000000000e+01.

**See Also**

None.
– (pointer offset subtraction)

Pointer offset subtraction operator.

**Synopsis**

`_pointer – _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**

`_pointer`

A pointer.

`_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.
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

Appendix. Using Prolog in Rules 335
– (sign reversal)

Unary negation operator.

Synopsis

–_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:

_reall 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**

\[ \_\text{number1} \times \_\text{number2} \]

**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} is 100,  
\_\text{integer2} is 30,  
\_\text{real1} is 2.1,  
\% Assign values.

\_\text{product1} is \_\text{integer1} \times \_\text{integer2},  
\% \_product1 is unified with 3000.

\_\text{product2} is \_\text{integer1} \times \_\text{real1}  
\% \_product 2 is unified with 2.10000000000000e+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 \_real2 / \_real1},_
  \% _\text{quotient1 is unified with 5.000000000000000e+00}_.

  _\text{quotient2 is \_integer1 / \_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 example 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.

_term1  = 'tec.tivoli.com',
_term2  = 'tec.tivoli.com',

% Assign values.

_term1 \== 'arrakis.tivoli.com'

% Succeeds.

_term1 \== _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).
  }
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:

_integer1 is 100,
_integer2 is 30,
_integer3 is 10,
_real1 is 2.1,

% Assign values.

_integer3 < _integer2,
% Succeeds.

int(_real1) < _real1,
% Succeeds.

_integer1 < _integer3
% Fails.

**See Also**

None.
Term unification operator.

**Synopsis**

\(_\text{term1} = \_\text{term2}\)

**Description**

The Prolog assignment operator. Succeeds if \(_\text{term1}\) and \(_\text{term2}\) can be unified with each other. Note that the \(=\) operator is not the same as the \(==\) operator, which tests for equality.

**Arguments**

\(_\text{term1}\) Any Prolog term.

\(_\text{term2}\) Any Prolog term.

**Examples**

The following example unifies the atom tec.tivoli.com with the variable \(_\text{hostname}\):

\(_\text{hostname} = \textquote{'tec.tivoli.com'}\)

**See Also**

\(==\) \(\neq\) \(\text{is}\)
Mathematical equal to.

Synopsis

\_arithExp1 =:= \_arithExp2

Description

Succeeds if the evaluation of \_arithExp1 is equal to the evaluation of \_arithExp2.

Arguments

\_arithExpr1

Any arithmetic expression. Must be instantiated.

\_arithExpr2

Any arithmetic expression. Must be instantiated.

Examples

\_integer1 is 100,
% Assign value.

\_integer1 =:= \_integer2 + 70,
% Succeeds, \_integer2 is unified with 30.

\_integer1 =:= \text{real(\_integer1)},
% Succeeds.

\_integer1 =:= - \_integer1
% Fails.

See Also

None.
Expression evaluation.

**Synopsis**

\[ \text{result} =? \text{function} \]

**Description**

Term \text{function} is evaluated and the result is unified with \text{result}. Use this operator when \text{function} is a computable function, such as those listed in “Functions” on page 327.

**Arguments**

- \text{function} A computable function.
- \text{result} Any Prolog term.

**Examples**

The following example shows that the substring function is evaluated and its result is placed in the variable \text{msg}:

\[ \text{msg} =? \text{substring(msg, start, length)} \]

**See Also**


Mathematical not equal to.

**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.

- `_integer1` is 100,
  - Assign value.

- `_integer1 =\= _integer1 + 1`,  
  - Succeeds.

- `- _integer1 =\= -100`  
  - Fails.

**See Also**

None.
Mathematical less than or equal to.

**Synopsis**

_arithExp1 <= _arithExp2

**Description**

Succeeds if the evaluation of _arithExp1 is less 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:

_integer1 is 100, 
_real1 is 2.1,
% Assign values.

_integer1 <= _integer1 + 1,
% Succeeds.

_real1 <= _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**

_arithExp1

Any arithmetic expression. Must be instantiated.

_arithExp2

Any arithmetic expression. Must be instantiated.

**Examples**

The following example shows various uses of the operator:

_integer1 is 100,
_integer2 is 30,
% Assign values.

_integer1 > _integer2,
% Succeeds.

_integer2 * 2 > _integer1,
% Succeeds.

_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:

_integer1 is 100,
_integer2 is 30,
_integer3 is 10,
% Assign values.

_integer1 >= _integer3 + 90,
% Succeeds.

_integer2 >= _integer1 - _integer1,
% Succeeds.

_integer3 >= _integer2
% Fails.

**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

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

```prolog
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:

```prolog
ascii(a, _code1),
\% Succeeds and _code1 is unified with 97.
```

```prolog
ascii(\n, _code2),
\% Succeeds and _code2 is unified with 10.
```

```prolog
ascii(_char1, 65),
\% Succeeds and _char1 is unified with 'A'.
```

```prolog
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 424 for additional information.

Arguments

_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.
   
   % 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. Tivoli 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.
   
   b. The Prolog interpreter input and output to test the asserted predicate. See
      "Using the Prolog Interpreter" on page 426 for additional information.
   
   c. A Tivoli Enterprise Console rule fragment that uses the asserted predicate in
      an action.
   
   ```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
   ```
atomlength(_substring, _sublen),
atompart(_string, _substring, _startsub, _len),
  _len1 is _startsub - 1,
atompart(_string, _left, 1, _len1),
  _startr is _len1 + _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:

```prolog
_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:

```prolog
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**

\[ \text{atompart}(_\text{atom}, _\text{atomPart}, _\text{startPos}, _\text{length}) \]

**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 \hspace{1cm} The original string. Must be an atom.

\_atomPart \hspace{1cm} A substring in \_atom. Must be an atom.

\_length \hspace{1cm} The length of the substring \_atomPart. Must be an integer.

\_startPos \hspace{1cm} 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.

\[ \text{atompart}(_\text{msg}, \ ' ', _\text{startOfSpace}, _\text{lenOfSpace}), \]

\% 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.

\[ \text{atompart}(_\text{msg}, _\text{host}, 1, _\text{lenOfHost}) \]
\% _host is unified with 'pikes-peak'.

**See Also**

substring
atomconcat

Concatenate atoms.

Synopsis

\[
\text{atomconcat}(\text{ListOfAtomics}, \text{concatAtom})
\]

\[\text{OR}\]

\[
\text{atomconcat}(\text{atom1}, \text{atom2}, \text{concatAtom})
\]

Description

Variable \text{concatAtom} is the atom constructed by concatenating all items from the list \text{ListOfAtomics} in the order of the list, or by concatenating the instantiated values of \text{atom1} and \text{atom2}. The values in \text{ListOfAtomics}, \text{atom1}, and \text{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

\text{atom1} An atom, integer, real, or pointer that \text{atom2} will concatenate to.

\text{atom2} An atom, integer, real, or pointer that will concatenate to \text{atom1}.

\text{concatAtom} The results of concatenating the arguments. This value is an atom.

\text{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:

   \[
   \text{hostname} = \text{'shinai'},
   \text{repeat_count} \text{ is } 5,
   \%
   \text{ Assign values.}
   \%
   \]

   \[
   \text{atomconcat}([\text{'Host '}, \text{hostname}, \text{'} unavailable ',
   \text{repeat_count}, \text{'} times.'], \text{new_msg})
   \%
   \text{ The variable new_msg is unified with the}
   \%
   \text{ atom 'Host shinai unavailable 5 times.'}
   \]

2. The following example shows various uses of atomconcat with three arguments:

   \[
   \text{hostname} = \text{'shinai'},
   \%
   \text{ Assign value.}
   \%
   \]

   \[
   \text{atomconcat}(_\text{hostname}, \text{' is down.'}, \text{down_msg}),
   \%
   \text{ down_msg is unified with 'shina is down'}.\%
   \]

   \[
   \text{msg} = \text{'shinai is back up.'},
   \%
   \text{ Assign value.}
   \%
   \]

   \[
   \text{atomconcat}(\text{up_hostname}, \text{' is back up.'}, \text{msg})
   \%
   \text{ up_hostname is unified with 'shinai'}.\%
   \]

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:
   ```bash
   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:
   ```bash
   odadmin environ set < /tmp/oserv.env
   ```

See the *Tivoli Management Framework Reference Manual, Version 3.6* 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 424 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 `TECpcomp` 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:

\[
\text{\_unixHosts} = ['arrakis', 'rooster', 'dune', 'perro'],
\text{\_ntHosts} = ['charon', 'scary', 'mach5', 'callisto'],
\]

% Assign values.

\[
\text{append(\_unixHosts, \_ntHosts, \_allHosts),}
\]

% \_allHosts is unified with the list  
% [arrakis, rooster, dune, perro, charon,  
% scary, mach5, callisto].

\[
\text{delete(\_allHosts, \_arrakis, \_residue),}
\]

% \_residue is unified with the list  
% [rooster, dune, perro, charon, scary, mach5, callisto].

\[
\text{delete(\_allHosts, \_host, \_residue)}
\]

% \_host is unified with \'arrakis\'.

**See Also**

None.
disjoint

Compare elements in two lists for uncommon values.

Synopsis
disjoint(_list1, _list2)

Description
Succeeds if _list1 and _list2 do not have any elements in common.

Arguments
_list1 Any list.
_list2 Any list.

Examples
The following example shows a successful comparison of two lists that do not have any elements in common:

_unixHosts = ['arrakis', 'rooster','dune', 'perro'],
_ntHosts = ['charon', 'scary', 'mach5', 'callisto'],
% Assign values.

disjoint(_unixHosts, _ntHosts)
% 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.

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**
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.

```pro
action: write_predicates: (
  fopen(_fp, 'maint_mode.pro',w),
  flisting(_fp, 'unix_hosts')
  fclose(_fp)
),
```

**See Also**
fclose, fopen
fopen

Open a file.

**Synopsis**

```plaintext
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
int_to_hex(_integer)

int_to_hex(_integer, _width)

Description
Converts the integer value instantiated in _integer to an atom representation of the hexadecimal value. The _width argument specifies the field width of the result, in which case the value of _atom is left-padded with zeros if necessary.

Arguments
_integer
An integer, either instantiated or free.

_width An integer. Specifies the field width for _atom.

Examples
The following example shows various conversions from integer to hexadecimal:
% Converts an integer to hexadecimal.
_hexstring1 = ? int_to_hex(100)
% _hexstring1 is unified with '64'.

% Converts an integer to hexadecimal with a specific field width.
_hexstring2 = ? int_to_hex(100, 8)
% _hexstring2 is unified with '00000064'.

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:

```
_integer = 200,
% Assign value.
integer(_integer),
% Succeeds.
_integer = 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)

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:

```prolog
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:

\[
\text{_unixHosts = ['arrakis', 'rooster','dune', 'perro'],}
\]
\[
\% Assign value.
\]
\[
\text{intersect(_unixHosts, ['perro', 'dune', 'odin'], _intersection)}
\]
\[
\% _intersection is unified with ['dune','perro'].
\]

**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
% _integer is unified with 10.

_result is _integer + 90
% _result is unified with 100.

See Also

=?
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:
_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:

\_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**

\[
\text{member}(_\text{element}, _\text{list})
\]

**Description**

Standard membership predicate. Succeeds if \_element is in \_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 will be 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:

```prolog
_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](389)
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 will be 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
% [49,46,48,51,50,48,48,48,101,43,48,49].

name(_realNumber, _realList),
% Instantiates _realNumber to
% 1.03200000000000e+01.

name(prolog, _list),
% Instantiates _list to
% [112,114,111,108,111,103].

name(_atom, _list)
% Instantiates _atom to 'prolog'.

See Also
None.
**nmember**

Check a list for a single value at an index.

**Synopsis**

\[ \text{nmember}(\_element, \_list, \_index) \]

**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

\texttt{nmembers(_indexes, _list, _listOfElements)}

Description

This predicate is similar to \texttt{nmember}, except it uses lists to specify multiple index and element argument values. Both \texttt{_indexes} and \texttt{_listOfElements} must not be free; that is, one must be instantiated.

Arguments

\texttt{_indexes}

A list, either instantiated or free.

\texttt{_list}

A list, must be instantiated.

\texttt{_listOfElements}

A list, either instantiated or free.

Examples

The following example shows various uses of the predicate:

\texttt{unixHosts = ['arrakis', 'rooster', 'dune', 'perro'],}
% Assign value.

\texttt{nmembers(_indexes, unixHosts, ['arrakis', 'dune']),}
% _indexes is unified with the list [1,3].

\texttt{nmembers([2, 4], unixHosts, _listOfElements)}
% _listOfElements is unified with the list
% [rooster, perro].

See Also

\texttt{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),
½ Succeeds.

integer(1.234)
½ Fails because this is a real number, not an integer.

not integer(1.234),
½ Succeeds, because integer(1.234) would fail.

See Also
None.
number

Tests whether a variable is of type number.

Synopsis

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:

_integer = 200,
_real = 2.345,
% Assign values.

number(_integer),
% Succeeds.

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:

```plaintext
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:

```prolog
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:

```tcl
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 (2^24-1) 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)
)  

Appendix. Using Prolog in Rules  399
See Also
None.
read

Read a Prolog term from an open Prolog source file.

**Synopsis**

read(_file, _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 403 for reading arbitrary text data.

**Arguments**

_file   The file pointer obtained from opening the file with the fopen predicate.
_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_rnt_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:
See Also

readln
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:

```
loghost: open_hosts: (fopen(_fp, '/etc/hosts', 'r'),
    readln(_fp, _hostsEntry)
    \% _hostsEntry is unified with \% '127.0.0.1\tlocalhost\n',
),
action: close_hosts: (fclose(_fp)
)
```

**See Also**

`read`
real (convert integer)

Converts 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 1.000000000000000e+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 '2.3450000000000000e+00'.

grealtoatom(_real1, '1.234')
% Succeeds. _real1 is unified with 1.2340000000000000e+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:

```prolog
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:

```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:

```prolog
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),
% _int1 is unified with 12.
_int2 is round(_float2),
% _int2 is unified with -12.
_int3 is round(_float3),
% _int3 is unified with 13.
_int4 is round(_float4)
% _int4 is unified with -13.

See Also
None.
rremove

Remove the first element from a list.

Synopsis

\[ \text{rremove}(\_element, \_list, \_tail) \]

Description

If \_element is free, instantiates \_element to the first element of the list \_list and unifies the rest of the list with \_tail.

If \_element is instantiated, \_element is removed from the list and the modified list is unified with \_tail.

Arguments

\_element
   If free, the first element of the list. If instantiated, the element to remove from the list. Any term.

\_list
   The original list to remove an element from.

\_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 _unixHosts and assign the modified list to the new list _tail:

\begin{verbatim}
_unixHosts = ['arrakis', 'rooster','dune', 'perro'],
% Assign value.
\end{verbatim}

\begin{verbatim}
rremove(_firstElement, _unixHosts, _tail)
% Unifies _tail with [rooster,dune,perro].
\end{verbatim}

\begin{verbatim}
rremove('rooster', _unixHosts, _tail2)
% Unifies _tail with [arrakis,dune,perro].
\end{verbatim}

See Also

None.
sort

Sort the elements of a list alphabetically.

Synopsis

sort(_list, _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:

_ntHosts = ['charon','scary','mach5','callisto'],
% Assign value.

sort(_ntHosts, _sorted)
% _sorted is unified with ['callisto','charon','mach5','scary'].

See Also

None.
_Printf

Print formatted data to an atom.

Synopsis
_Printf(_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 will be 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:

% Assign values.
_Printf(_string1, '%s', _string),
% _string1 is unified with 'Hello, World'.
_Printf(_string2, '%20s', _string),
% _string2 is unified with ' Hello, World'.

Appendix. Using Prolog in Rules 413
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.
Remove characters from an atom.

**Synopsis**

\[ \text{strip}(\_\text{atom}, \_\text{position}) \]

æORæ

\[ \text{strip}(\_\text{atom}, \_\text{position}, \_\text{chars}) \]

**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 ' atom with spaces '. 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 'atom with spaces'. 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 quotes in the msg attribute. The value of this attribute is stored in the _msg variable. This example removes the double quotes 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 quotes 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:

_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:

\_ntHosts = ['charon', 'scary', 'mach5', 'callisto'],
% Assign value.

subtract(_nthosts, ['charon', 'scary'], ['mach5', 'callisto']),
% Succeeds.

subtract(_nthosts, ['mach5', 'callisto'], _difference)
% Instantiates _difference with ['charon', 'scary'].

See Also

None.
term_type

Gets the type of a variable.

Synopsis

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:

_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.

```prolog
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)
        )
    ),
    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" on page 425 to use this method.
- Using the compile, consult/reconsult predicates in a Tivoli Enterprise console rule. See "compile" on page 368, "consult" on page 370, and "reconsult" on page 407 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.
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 360.

rule: boot_string_match:
  (event: _start of_class within ['TEC_Start']
   where [],
   reception_action: action0: {
     assert((my_string_match(_string, _substring, _left, _right) :-
       atomlength(_substring, _sublen),
       atompart(_string, _substring, _startsub, _len),
       _len is _startsub - 1,
       atompart(_string, _left, 1, _len1),
       _startr is _len1 + _sublen + 1,
       atompart(_string, _right, _startr, _lenr),
       !,
       true))
   })
).
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 the following command 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.
   ```bash
   export BIM_PROLOG_DIR=..
   ```
3. Start the interpreter with the BIMprolog.exe 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
```
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.

© Copyright IBM Corp. 2002
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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>IBMLink</td>
<td>Tivoli Logo</td>
</tr>
<tr>
<td>IBM</td>
<td>NetView</td>
<td>Tivoli Enterprise Console</td>
</tr>
<tr>
<td>IBM Logo</td>
<td>Tivoli</td>
<td>TME</td>
</tr>
</tbody>
</table>

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.

Other company, product, and service names may be trademarks or service marks of others.
Glossary

This glossary includes Tivoli product terminology as well as selected terms and definitions from:

- The American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

- The ANSI/EIA Standard--440-A, Fiber Optic Terminology. Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.

- The Information Technology Vocabulary developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.


- Internet Request for Comments: 1208, Glossary of Networking Terms

- Internet Request for Comments: 1392, Internet Users’ Glossary


The following cross-references are used in this glossary:

**Contrast with:**
This refers the reader to a term that has an opposed or substantively different meaning.

**See:**
This refers the reader to (a) a related term, (b) a term that is the expanded form of an abbreviation or acronym, or (c) a synonym or more preferred term.

**Obsolete term for:**
This indicates that the term should not be used and refers the reader to the preferred term.

A

**action body.** In the Tivoli Enterprise Console, the part of a rule that contains actions to take if the rule evaluates to true.

**adapter.** (1) A part that electrically or physically connects a device to a computer or to another device. (2) Software that enables different software components or products to interact with one another. (3) See [event adapter]

**administrator.** See [Tivoli administrator]

**API.** See [application programming interface]
application programming interface (API).  A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

attribute.  A characteristic that identifies and describes a managed object. The characteristic can be determined, and possibly changed, through operations on the managed object.

attribute condition.  For Tivoli Enterprise Console rules, an expression that specifies that if certain attribute values are present in an event, the rule evaluates to true.

attribute facet.  In the Tivoli Enterprise Console, the type of data that an event can contain.

authorization role.  In a Tivoli environment, a role assigned to Tivoli administrators to enable them to perform their assigned systems management tasks. A role may be granted over the entire Tivoli Management Region or over a specific set of resources, such as those contained in a policy region. Examples of authorization roles include: super, senior, admin, and user.

B

BAROC.  See Basic Recorder of Objects in C

Basic Recorder of Objects in C (BAROC).  In the event server of the Tivoli Enterprise Console, the internal representation of the defined event classes.

C

class.  (1) In object-oriented design or programming, a model or template that can be instantiated to create objects with a common definition and therefore, common properties, operations, and behavior. An object is an instance of a class. (2) In the AIX operating system, pertaining to the I/O characteristics of a device. System devices are classified as block or character devices.

CLI.  See command line interface

command line interface (CLI).  A type of computer interface in which the input command is a string of text characters. Contrast with graphical user interface

control primitive.  In the Tivoli Enterprise Console, a primitive that provides the capability to exit a set of rule actions, a rule set, or a rule base.

D

desktop.  A graphical user interface (GUI) that enables a user to interact with and perform operations on a computer system.

E

EIF.  See Tivoli Event Integration Facility

epoch.  The time and date corresponding to 0 in an operating system’s clock and time-stamp values. For most versions of the UNIX operating system, the epoch is 00:00:00 GMT, 01 January 1970. System time is measured as the number of seconds past the epoch.

event.  (1) An occurrence of significance to a task (such as the opening of a window or the completion of an asynchronous operation). (2) In the Tivoli environment, any significant change in the state of a system resource, network resource, or network application. An event can be generated for a problem, for the resolution of a problem, or for the successful completion of a task. Examples of events are: the normal starting and stopping of a process, the abnormal termination of a process, and the malfunctioning of a server.
**event adapter.** In a Tivoli environment, software that converts events into a format that the Tivoli Enterprise Console can use and forwards the events to the event server. Using the Tivoli Event Integration Facility, an organization can develop its own event adapters, tailored to its network environment and specific needs.

**event class.** In the Tivoli Enterprise Console, a classification for an event that indicates the type of information that the event adapter will send to the event server.

**event console.** In the Tivoli Enterprise Console, a graphical user interface (GUI) that enables system administrators to view and respond to dispatched events from the event server. The Tivoli Event Integration Facility does not directly use or affect event consoles.

**event correlation.** In the Tivoli Enterprise Console, the process of correlating separate events to a common cause. For example, the Tivoli Enterprise Console may receive several NFS server not responding events from several different applications, as well as a host down event for the NFS server. The Tivoli Enterprise Console can then correlate the various NFS server not responding events to their common cause, which is: the NFS server is down. See [rule](#).

**event filter.** (1) In a Tivoli environment, software that determines which events are forwarded to a specified destination. Filtering events helps to reduce network traffic. Tivoli administrators configure the event filters. (2) In Tivoli NetView, a logical expression of criteria that determine which events are forwarded to the application program that registers the event filter with the event sieve agent. A filter is referred to as "simple" or "compound" depending on how it is handled by the filter editor.

**event group.** In the Tivoli Enterprise Console, a set of events that meet certain criteria. Each event group is represented by an icon on the event console. Tivoli administrators can monitor event groups that are relevant to their specific areas of responsibility.

**event repository.** In a Tivoli environment, a RIM repository that contains information that is collected or generated by the Tivoli Enterprise Console. For example, in the event repository, Tivoli Enterprise Console stores information regarding events.

**event server.** In the Tivoli Enterprise Console, a central server that processes events. The event server creates an entry for each incoming event and evaluates the event against a rule base to determine whether it can respond to or modify the event automatically. The event server also updates the event consoles with the current event information. If the primary event server is not available, events can be sent to a secondary event server.

**event specifier.** In the Tivoli Enterprise Console, a primitive that is used in searching for events in the event cache. For example, the event specifier can search for duplicate events, an event that matches a user-specified attribute, or an event that occurs within a certain time period.

**G**

**graphical user interface (GUI).** A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device. Contrast with [command line interface](#).

GUI. See [graphical user interface](#).

**H**

**host.** (1) A computer that is connected to a network (such as the Internet or an SNA network) and provides an access point to that network. Also, depending on the environment, the host may provide centralized control of the network. The host can be a client, a server, or both a client and a server simultaneously. (2) In a Tivoli environment, a computer that serves as a managed node for a profile distribution.

**I**

**instance.** In object-oriented programming, an object created by instantiating a class.
managed node. (1) In Internet communications, a workstation, server, or router that contains a network management agent. In the Internet Protocol (IP), the managed node usually contains a Simple Network Management Protocol (SNMP) agent. (2) In a Tivoli environment, any managed resource on which the Tivoli Management Framework is installed.


oserv. The name of the object request broker used by the Tivoli environment. Oserv runs on the TMR server and each TMR client.

policy region. In a Tivoli environment, a group of managed resources that share one or more common policies. Tivoli administrators use policy regions to model the management and organizational structure of a network computing environment. The administrators can group similar resources, define access to and control the resources, and associate rules for governing the resources. The policy region contains resource types and the list of resources to be managed. A policy region is represented on the Tivoli desktop by an icon that resembles a capitol building (dome icon). When a Tivoli Management Region (TMR) is created, a policy region with the same name is also created. In this case, the TMR has only one policy region. However, in most cases, a Tivoli administrator creates other policy regions and subregions to represent the organization of the TMR. A TMR addresses the physical connectivity of resources whereas a policy region addresses the logical organization of resources.

primitive. In the Tivoli Enterprise Console, a rules-language program construct that comprises an action. A control primitive and an event specifier are examples of primitives.

profile. In a Tivoli environment, a container for application-specific information about a particular type of resource. A Tivoli application specifies the template for its profiles; the template includes information about the resources that can be managed by that Tivoli application.

A profile is created in the context of a profile manager; the profile manager links a profile to the Tivoli resource (for example, a managed node) that uses the information contained in the profile. A profile does not have any direct subscribers.

RDBMS. See relational database management system.

relational database management system (RDBMS). A collection of hardware and software that organizes and provides access to a relational database.

resource. (1) Any facility of a computing system or operating system required by a job or task, including main storage, input/output devices, the processing unit, data sets, and control or processing programs. (2) In Tivoli NetView for OS/390, any hardware or software that provides function to the network.

rule. In the Tivoli Enterprise Console product and Tivoli Event Services Manager, one or more logical statements that enable the event server to recognize relationships among events (event correlation) and to execute automated responses accordingly. See also event correlation, rule base, and rule set.

rule base. In the Tivoli Enterprise Console, one or more rule sets and the event class definitions for which the rules are written. The Tivoli Enterprise Console uses the rule base in managing events. An organization can create many rule bases, with each rule base fulfilling a different set of needs for network computing management.

rule set. In the Tivoli Enterprise Console, a file that contains one or more rules. Also see rule base.
severity level. In the Tivoli Enterprise Console, a classification for an event that indicates its degree of severity. Severity levels can be modified by a user or a Tivoli Enterprise Console rule. The predefined severity levels, in order of descending severity, include: fatal, critical, warning, minor, harmless, and unknown.

Simple Network Management Protocol (SNMP). In the Internet suite of protocols, a network management protocol that is used to monitor routers and attached networks. SNMP is an application layer protocol. Information on devices managed is defined and stored in the application’s Management Information Base (MIB).


source. In the Tivoli Enterprise Console, a resource, such as a host, that is being monitored by an event adapter.

T

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. (1) (A) (2) In a Tivoli environment, the definition of an action that must be routinely performed on various managed nodes throughout the network. A task defines the executables to be run when the task is executed, the authorization role required to execute the task, and the user or group name under which the task will execute.

TEC. See Tivoli Enterprise Console.

Tivoli Enterprise software. The integrated suite of Tivoli products for systems management in a large organization. These products enable system administrators to manage their network computing enterprise according to the disciplines of availability management, deployment management, operations and administration, security management, and service-level management. This suite includes Tivoli Business Systems Manager, Tivoli NetView for OS/390, and Tivoli Decision Support.

Tivoli Event Integration Facility. A Tivoli toolkit that provides a simple application programming interface to enable customers and Tivoli Partners to develop new event adapters that can forward events to the Tivoli Enterprise Console product.

Tivoli Management Region (TMR). In a Tivoli environment, a Tivoli server and the set of clients that it serves. An organization can have more than one region. A Tivoli management region addresses the physical connectivity of resources whereas a policy region addresses the logical organization of resources.


Tivoli NetView for OS/390. A Tivoli product that enables centralized systems and network management from an OS/390 environment. Through its MultiSystem Manager component, Tivoli NetView for OS/390 enables management of distributed resources, such as Internet Protocol (IP) resources, NetWare resources, asynchronous transfer mode (ATM) resources, and others. Contrast with Tivoli NetView.

TME 10. See Tivoli Enterprise software.

Tivoli administrator. In a Tivoli environment, a system administrator who has been authorized to perform systems management tasks and manage policy regions in one or more networks. Each Tivoli administrator is represented by an icon on the Tivoli desktop.

Tivoli Enterprise Console (TEC). A Tivoli product that collects, processes, and automatically initiates corrective actions for system, application, network, and database events. The Tivoli Enterprise Console product is the central control point for events from all sources. It provides a centralized, global view of the network computing environment, and it uses distributed event monitors to collect information, a central event server to process information, and distributed event consoles to present information to system administrators.

Tivoli Management Region (TMR). In a Tivoli environment, a Tivoli server and the set of clients that it serves. An organization can have more than one TMR. A TMR addresses the physical connectivity of resources whereas a policy region addresses the logical organization of resources.
Index

Special characters

_(Prolog anonymous variable)_ 71
_ class specification 71
_ class class specification 71
_ class outside class specification 73
_ class variable 71
_ event variable 71
- (pointer offset subtraction) Prolog predicate 332
- (pointer subtraction) Prolog predicate 334
- (sign reversal) Prolog predicate 336
- (subtraction) Prolog predicate 337
/ Prolog predicate 339
// Prolog predicate 340
/* */ (comment delimiters) 59, 322
.conf file 212
.rbtargets directory 12
.rls file 53
.rpk file 53
* Prolog predicate 338
\ 348
\ (line continuation character) xiii
\= Prolog predicate 345
@ \ 341
@ Prolog predicate 343
@>= Prolog predicate 344
@= \ 342
% (comment delimiter) 59, 322
> Prolog predicate 345
# (comment delimiter) 59
+= Prolog predicate 346
= Prolog predicate 349
== Prolog predicate 350
?= Prolog predicate 351
= \ 353
=\= Prolog predicate 352
== Prolog predicate 354

A
abandon Prolog predicate 357
ACK status 7
acl attribute 6
action body, rule 78
actions, rules
See rules, actions 78
active rule base 35
adapters, event 3
add_to_repeat_count rule language predicate 89
administrator attribute
changing the value 117
described 6
setting the value 226
administrator, changing 117, 226
administrator, setting 226
all_clear_targets rule language predicate 90
all_duplicates rule language predicate 91
all_instances rule language predicate 93
anonymous variable, Prolog 71
any_clear_target rule language predicate 94
any_clearing_event rule language predicate 95
append Prolog predicate 358
ascii Prolog predicate 359
assert Prolog predicate 360, 425
atom manipulation Prolog predicates 327
atom Prolog predicate 362
atom, in Prolog 319
atomconcat Prolog predicate 366
atomic Prolog predicate 363
atmlen Prolog predicate 364
atompart Prolog predicate 365
atmtimeout Prolog predicate 367
attr_condition rule language predicate 96
attr_exception rule language predicate 98
attr_sequence rule language predicate 99
attribute change filter 75
See also change rule filters, attribute change filter 75
attribute conditions
See event filters, attribute conditions or events, attribute
conditions or event sequence, attribute conditions 73
attribute filter
See event filters, attribute filter 10
attribute value test 74
attributes, event
See event attributes 107
automated response 7

B
back slashes, converting 218
BAROC files
/** */ (comment delimiters) 59
attribute definitions, retrieving 103
comments 59
data types, mapping to Prolog 323
enumeration definitions, described 47
evaluation definitions, retrieving 105
importing modified files 48
location in rule base 18
naming conventions 53
order 17, 45
parser grammar 47
root.baroc 17, 47
See also event classes 39
syntax 40, 47
tec.baroc 17, 47
UTF-8 support 48
BAROC manipulation rule language predicates 83
base event 17
bo_add_at_slotval_begin rule language predicate 100
bo_add_at_slotval_end rule language predicate 101
bo_get_class_of rule language predicate 102
bo_get_class_slots rule language predicate 103
bo_get_enum_options rule language predicate 105
bo_get_slotval rule language predicate 106
bo_is_defined_for_class rule language predicate 107
bo_is_direct_super_of rule language predicate 108
bo_is_super_of rule language predicate 109
bo_remove_from_slotval rule language predicate 110

COPYRIGHT IBM Corp. 2002

439
bo_reset_default_slotval rule language predicate 112
bo_set_slotval rule language predicate 113

C

cache, event
See event cache 7
cancel_all_timers rule language predicate 114
cancel_timer rule language predicate 115
causal relationships between events
creating a compound rule, rule builder procedure 295
recording the relationships 237
special cases 238
using relationship information 239
cause events
described 239
linking effect events 194
processing guidelines 241
searching for most causal 166
unlinking effect events 235
cause_date_reception attribute 6
cause_event_handle attribute 6
change request
described 8
drop 151
external 50
place 199
processing by rule engine 50
senders 74
change rule filters
attribute change filter 75
described 74
sender filter 74
senders of change requests 74
change rules
collection, placing 199
See also change request 8
described 8
processing by rule engine 50
structure 63
change_event_administrator rule language predicate 117
change_event_severity rule language predicate 118
change_event_status rule language predicate 119
check_all_thresholds rule language predicate 120
check_and_increment_count rule language predicate 121
check_event_criteria rule language predicate 123
check_threshold rule language predicate 125
classes
See event classes 39
clear_closed_events rule language predicate 127
clearing closed events 127
clearing events
checking for existence 193
defining a clearing event 138
described 240
processing guidelines 242
searching for 95
target events, finding 90, 94
clears rule language predicate 128
CLOSED status 7
commands
postemsg 5, 253
See also the Tivoli Enterprise Console Reference Manual x
upgrade_gui.sh 279
wpostemsg 5, 253
wrb
rule base targets, creating 13

commands (continued)
wrbi (continued)
rule packs, creating 16
usage recommendation 11
wrbuspgride 11
wtdumprl 3
comments
BAROC files 59
Prolog 322
rule language 59
commit_action rule language predicate 130
commit_rule rule language predicate 131
commit_set rule language predicate 132
compile Prolog predicate 368
compiling a rule base, rule builder procedure 33, 301
compiling a rule base, wrb command
See the Tivoli Enterprise Console Reference Manual x
compiling older rule bases 11
compound rules 295
See also correlation rules 9
compound rules, creating, rule builder procedure 295
conditions, simple rule 67
cnf file 212
constants, rule language 59
consult Prolog predicate 370
convert data type Prolog predicates 326
convert_ascii_time rule language predicate 133
convert_gm_time rule language predicate 134
convert_local_time rule language predicate 135
copying a rule base, rule builder procedure 27
copying a rule base, wrb command
See the Tivoli Enterprise Console Reference Manual x
copying rule bases 14
correlating events 237
correlating events, rule builder 297
correlation examples 237
correlation rule language predicates 80
correlation rules
described 9
elements 237
See also compound rules 295
structure 69
counters in rule actions 121, 188
create_cache_search_criteria rule language predicate 136
create_clearing_event rule language predicate 138
create_event_criteria rule language predicate 141
create_event_sequence rule language predicate 144
create_threshold rule language predicate 149
creating a compound rule, rule builder procedure 295
creating a new rule base, rule builder procedure 22
creating a simple rule, rule builder procedure 285
credibility attribute 6
criteria
checking against event 123
creating for event 141
creating for event cache search 136
evaluation of event 243
current rule base 35
Customer Support xi

D
data types, mapping BAROC to Prolog 323
date_reception attribute 6, 7
DBCS support
See UTF-8 support 37
debug
detailed debugging for rules 225
rules
logging error messages 197
profile 261
trace 256
tracing user-defined Prolog predicates 234
debuge rule language predicates 88
decrement_slot rule language predicate 150
decrementing attribute values 150
default facet 43
default rule base 13, 18
default rule set
See also rule sets, default rule set 271
delete Prolog predicate 372
deleting a rule base, rule builder procedure 28
developing event classes 39
developing rule bases 1
directives
fire_on_non_leaf 44, 79
profile 80
See also rules, directives 79
trace 80
directory paths, converting back slashes 218
directory paths, rule base 53
directory permissions for UNIX rule bases 36
discard events 152
disjoint Prolog predicate 373
distributed rule bases 11
distributing files with rule bases 13
domain-significant events 2
double-byte character support (DBCS)
See UTF-8 support 37
drop_change_request rule language predicate 151
drop_received_event rule language predicate 152
dropping change requests 151
dropping received events 152
dup_detect facet 43
duplicate events
counting duplicates 89
described 82
dup_detect facet 43
searching the event cache for all duplicates 91
searching the event cache for the first duplicate 167
duration attribute 6
event actions for simple rules 67
event activity reporting
capturing information 236
defining reporting criteria 190
resetting counts 219
writing the report 203
event activity rule language predicates 84
event adapters 3
event analysis x, 1, 39
event attributes
acl 6
adapter, supplied by 3
adding a value to a list 100, 101
administrator
changing the value 117
described 6
setting the value 226
cause_date_reception 6
cause_event_handle 6
conditions 243
credibility 6
date_reception 6, 7
decrementing values 150
default value, setting 112
definition in BAROC file 41
definitions, checking 107
duration 6
event server, supplied by 6
event_handle 6, 7
facets 43
incrementing values 187
mapping to Prolog data types 323
modified, mark as 208
msg 227
num_actions 6
retrieving values 177
server_handle 6, 7
server_path 7
server_path, parsing 213
severity 118, 229
simple_type 42
status
changing the value 119
described 7
setting the value 230
test for values 74
value, deleting from a list 110
value, modifying 113
event cache
all duplicates search 91
all instances search 93
cause event search 166
described 7
duplicate events, searching for all duplicates 91
duplicate events, searching for the first duplicate 167
effect event search 169
first duplicate search 167
first instance search 170
forced cleaning 49
related event search 172, 244
reverse lookup 241
search criteria, creating 136
searching based on event criteria 224
writing to file 200
event cache search rule language predicates 81
EVENT class  45
event class definition files  
   See BAROC files  39
event classes  
   attribute definitions  41, 103
   attribute definitions, checking  107
   attribute values, retrieving  106
   attributes, retrieving information about  103
   base event  45
   class names, retrieving  102
   definition  40
   design recommendation  47
   designing event classes  39
   distributed rule base  11
   enumerations, compared to  47
   EVENT class  45
   fire_on_non_leaf directive  79
   hierarchy  41
   hierarchy, writing to file  202
   importing modified BAROC files  48
   inheritance  44, 46
   leaf  44
   leaf-node  44
   naming conventions  53
   non-leaf classes, rule evaluation  79
   order  45
   parser grammar  47
   root event  45
   See also BAROC files  39
   subclass, described  44
   superclass checking  108, 109
   superclass, described  44
   synchronizing among distributed rule bases  17
   syntax  47
   triggering rules with non-leaf classes  79
   event conditions for simple rules  67
   event correlation rule language predicates  80, 242
   event criteria rule language predicates  84
   event criteria, checking  123
   event criteria, creating  141
   event filters  
      attribute conditions  73
      attribute filter  10, 73
      class filter  10, 71
      class specification  
         _  71
         _class  71
         _class outside  73
         _class within  72
      described  71
      described  10, 71
      multiple solutions, tracing  261
      of_class operator  71
      operators  74
      structure  71
      syntax  71
      variable instantiation  74
      variables  71
      where clause  73
   event ID  7
   event sequence  
      attribute conditions  243
      attribute conditions, defining  96
      attribute exceptions, defining  98
      attribute sequence  99
      attribute values, changing  99
      cause event search  166

   event sequence (continued)  
      clearing events, defining  128, 138
      creating an event sequence  144, 237
      definition  237
      described  237
      effect event search  169
      matching different attributes in other events  98
      related event search  172
   event server  
      .conf file  212
      configuration file  178, 212
      forwarding events  174, 212
      path parsing  213
      sending events  174, 212
      server_path attribute parsing  213
      start  304
      stop  304
   event_handle attribute  6, 7
   events  
      activity report  
         capturing information  236
         defining criteria  190
         resetting counts  219
         writing the report  203
      administrator, changing  117, 226
      administrator, setting  226
      attribute conditions  73, 243
      cache searching  136, 224
      cache, described  7
      causal relationships  
         creating a compound rule, rule builder procedure  295
         recording relationships  237
         using relationship information  239
      cause events  
         described  239
         linking effect events  194
         processing guidelines  241
         searching for  166
         unlinking effect events  235
      change request  
         described  8
         external  50
         place  199
         processing by rule engine  50
      clearing events  
         checking for existence  193
         defining a clearing event  128, 138
         described  240
         finding target events for clearing  90, 94
         processing guidelines  242
         searching for  95
         closed events, clearing  127
         correlation examples  237
      counting  121, 188
      criteria  
         check  123
         create  141
         evaluation of event  243
      described  2
      discarding events  152
      domain-significant  2
      dropping events  152
      duplicate events  
         described  82
         dup_detect facet  43
         searching the event cache for all duplicates  91
         searching the event cache for the first duplicate  167
Index 443

events (continued)
effect events
described 240
linking cause events 194
processing guidelines 241
searching for 169
unlinking cause events 235
enterprise-significant 1
filtering 5
first instance search 170
forwarding to another server 174, 212
generating an internal event 176
ID 7
instances, searching the cache 93
internal events, processing 50
internal events, sending 176
message, changing 227
message, setting 227
NetView component, received from 265
path, event server 213
reanalysis 8, 50
reception log, viewing events 3
redo request 8, 50
related event search 172, 244
relationships, defining 243
reporting activity
See events, activity report 203
sending from command line 5, 253
sending to another server 174, 212
sending with SendEvents program 254
sending with tec_agent_demo program 253
severity, changing 118, 229
severity, setting 229
status, changing 119, 230
status, setting 230
testing rules with events 253
threshold checking 120, 125
threshold creating 149
timer, setting on event 9
examples
correlation 237
event activity report 191
event sequence definition 145
global variables 57
profile report, rule execution 263
Prolog interpreter 427
Prolog, using in a rule 324
rule base 18
rule builder 305
rules
correlation 237
provided on event server x
trace, rule 259
exec_program rule language predicate 154
exec_program_local rule language predicate 156
exec_task rule language predicate 159
exec_task_local rule language predicate 162
executing a program from rules 154, 156
executing a task from rules 159, 162

F

facets, attribute 43
facts, Prolog 320
fclose Prolog predicate 375
feedback about publications xi

g

general purpose rule language predicates 87
generate_event rule language predicate 176
generating internal events 176
get_attributes rule language predicate 177
get_config_param rule language predicate 178
globalGRP rule language predicate 180
global_var rule language predicate 181
globals rule language predicate 182
global_variable rule language predicates 86
global variables
deleting 153
described 55
element 57
existence checking 186
information, retrieving for all 182
loading from file 196
manipulating 56
predicates, rule language 86
values
resetting for group 220
files
.conf 212
BAROC
attribute definitions 41
described 39
See also BAROC files 39
configuration, event server 212
event cache 200
root.baroc 17, 47
rule packs 53
rule sets 53
tec.baroc 17, 47
user_predicates 424
files, distributing with rule base 13
filtering events 5
filters
change rule
See change rule filters 74
described 71
event
See event filters 71
timer rule
See timer rule filters 76
fire_on_non_leaf directive 44, 79
first_causal_event rule language predicate 166
first_duplicate rule language predicate 167
first_effect_event rule language predicate 169
first_instance rule language predicate 170
first_related_event rule language predicate 172
flisting Prolog predicate 376
flow control in rules
controlling execution within a rule set 131
controlling execution within actions of rule 130
controlling execution within rule base 132
flow control rule language predicates 83
fopen Prolog predicate 377
forced cleaning of the event cache 49
forward slashes, converting from back slashes 218
forward_event rule language predicate 174
forwarding events 174, 212
free, described 324
global variables (continued)
values (continued)
retrieving for group 180
retrieving single 181
setting 231
writing group to file 223
global_exists rule language predicate 186
grammar for BAROC file parser 47
grammar for rule parser 60
ground Prolog predicate 378
ground, described 324
GUI rule builder
See rule builder 279

H
heartbeat monitoring 274

I
1/O Prolog predicates 330
importing a rule set into a rule base, rule builder procedure 29
importing BAROC files into a rule base, rule builder procedure 31
importing modified BAROC files 48
importing objects into a rule base 12
importing predicates 17
importing rule packs 14
importing rule sets 14
increment_slot rule language predicate 187
incrementing attribute values 187
inheritance, event class 44, 46
init_count rule language predicate 188
init_event_activity rule language predicate 190
instantiated, described 324
instantiation and unification Prolog predicates 325
int_to_hex Prolog predicate 379
integer Prolog predicate 380
Integrated TCP/IP Services Component
see rule sets, NetView 265
internal events, generating 176
internal events, processing 50
internationalization support 37, 48, 279
interpreter, Prolog 426
intersect Prolog predicate 381
inttoatom Prolog predicate 383
IP address, checking for unreachable 192
ip_node_unreachable rule language predicate 192
is Prolog predicate 384
is_clearing_event rule language predicate 192
is_list Prolog predicate 385

K
knowledge base Prolog predicates 330
knowledge base, described 321

L
launching a program from rules 154, 156
launching a task from rules 159, 162
leaf class 44
leaf-node class 44
length Prolog predicate 386

M
mathematical expression Prolog predicates 328
MBCS support
See UTF-8 support 37
member Prolog predicate 388
message, changing for an event 227
message, setting for an event 227
modifying rule sets 17
monitoring, heartbeat 274
msg attribute 227
multiple-byte character support (MBCS)
See UTF-8 support 37

N
ame Prolog predicate 390
naming conventions, rule language 53
NetView component
see rule sets, NetView 265
Netview Integrated TCP/IP Services Component
see rule sets, NetView 265
netview.rls
see rule sets, NetView 265
nmember Prolog predicate 391
nmembers Prolog predicate 392
non-editable rule sets 279
not Prolog predicate 393
num_actions attribute 6
number Prolog predicate 394

O
object file, Prolog 368
of_class operator 71
online publications xi
OPEN status 7

P
parse facet 43
parser grammar for BAROC files 47
parser grammar for rules 60
paths, converting back slashes 218
performance tips, rule engine 51
place_change_request rule language predicate 199
plain rules, described 8
plain rules, structure 61
pointer Prolog predicate 395
pointeroffset Prolog predicate 396
pointertoatom Prolog predicate 397
pointertoint Prolog predicate 398
postemsg command 5, 253

predicates
- described 80, 319
- erase 272
- import, automatic 17
- loading user-defined Prolog predicates 424
- online information 80, 319
- Prolog interpreter 426
- record 272
- recorded 272
- rerecord 272
- See also rule language predicates or Prolog predicates 80
- synchronizing among distributed rule bases 17
- tracing user-defined Prolog predicates 234
- user-defined, loading Prolog 424, 425

print_cache rule language predicate 200
print_class_tree rule language predicate 202
print_event_activity rule language predicate 203
printf Prolog predicate 399
profile directive 80
profiling rule execution 80, 261
programs, executing from rules 154, 156

Prolog
- (anonymous) variable 71
- atoms 319
- basics 319
- comments 322
- constants 319
- data types 322
- data types, mapping to BAROC 323
- example, using in a rule 324
- facts 320
- free, described 324
- ground, described 324
- instantiated, described 324
- interpreter 426
- knowledge base 321
- language, described 8
- loading user-defined predicates 424
- object file 368
- predicates, described 319
- See also Prolog predicates 319
- predicates, online information 319
- rules 320
- terms 319
- unification, described 324
- using in rules 319
- variables 319

Prolog predicates
- (pointer offset subtraction) 332
- (pointer subtraction) 334
- (sign reversal) 336
- (subtraction) 337
- / 339
- // 340
- * 338
- \ 348
- \= 345
- @@\ 341
- @> 343
- @>= 344
- @=\ 342
- > 355

Prolog predicates (continued)
- >= 356
- + (addition) 331
- + (pointer offset addition) 346
- = 349
- == 350
- ?? 351
- \ 353
- \= 352
- == 354
- abolish 357
- append 358
- ascii 359
- assert 360, 425
- atom 362
- atom manipulation predicates 327
- atomconcat 366
- atomic 363
- atomlength 364
- atompart 365
- atomtolist 367
- categories 325
- compile 368
- consult 370
- convert data type predicates 326
- delete 372
- described 319
- disjoint 373
- empty_list 374
- fclose 375
- flisting 376
- fopen 377
- ground 378
- I/O predicates 330
- instantiation and unification predicates 325
- int_to_hex 379
- integer 380
- intersect 381
- inttoatom 383
- is 384
- is_list 385
- knowledge base predicates 330
- length 386
- list manipulation predicates 328
- loading user-defined predicates 424
- lowertoupper 387
- mathematical expression predicates 328
- member 388
- name 390
- nmember 391
- nmembers 392
- not 393
- number 394
- online information 319
- pointer 395
- pointeroffset 396
- pointertoatom 397
- pointertoint 398
- printf 399
- Prolog interpreter 426
- quick reference 325
- read 401
- readln 403
- real (convert integer) 404
- real (test real type) 405
- realtime 406
- reconsult 407

Index 445
Prolog predicates (continued)
  remove_dups 408
  retract 409
  round 410
  remove 411
  sort 412
  sprintf 413
  strip 415
  subset 417
  substring 418
  subtract 419
  term_type 420
  test data type and variable predicates 326
  union 421
  write 422

reports (continued)
  profile, rule execution 261
  rerecord Prolog predicate 272
  reset_event_activity rule language predicate 219
  reset_global_grp rule language predicate 220
  resolve_time rule language predicate 221
  RESPONSE status 7
  response, automated 7
  restarting the event server 304
  retract Prolog predicate 409
  reverse lookup 241
  rls file 53
  root event 45
  root.baroc file 17, 47
  round Prolog predicate 410
  rpk file 53
  remove Prolog predicate 411

rule actions
  See rules, actions 78

rule base targets
  described 11, 12
  event classes 17
   See also event classes 39
  importing rule packs 14
  importing rule sets 14
  structure 15

rule bases
  active 35
  BAROC file location 18
  class hierarchy, writing to file 202
  commands 13
  compile 11
  compile, rule builder procedure 301
  compile, wrb command
   See the Tivoli Enterprise Console Reference Manual 301
  copy 14
  default 13, 18
  described 11
  development 1, 11
  directory paths 53
  directory permissions, UNIX 36
  distributed 11
  editing an existing rule base, rule builder procedure 283
  event classes 11
   See also event classes 39
  example 18
  importing objects 12
  importing rule packs 14
  importing rule sets 14
  loaded 35
  manipulation 11
  naming conventions 53
  rule base targets 11, 12
   See also rule base targets 12
  rule builder
   See rule builder 279
  rule packs 13
   See also rule packs 13
  rule sets 13
   See also rule sets 13
  structure 12, 15
  targets 11, 12
   See also rule base targets 12
  UNIX directory permissions 36
  upgrading 11
  wrb command 11
rule builder
    compiling a rule base 33, 301
    copying a rule base 27
    correlating events 297
    creating a compound rule 295
    creating a new rule base 22
    creating a new rule set 283
    creating a simple rule 285
    deleting a rule base 28
described 279
edited a rule set 284
    editing an existing rule base 24, 283
event actions for simple rules 292
    event conditions for simple rules 289
examples 305
importing a rule set into a rule base 29
    importing BAROC files into a rule base 31
listing known rule bases 24
    listing rule sets and BAROC files in rule base 26
loading and activating a rule base 35
overview 280
procedures 280
    saving a rule set 300
start 282
text editor, using 279
upgrading rules from previous releases 22
UTF-8 support 279
Rule cache full
forced cleaning 49
rule engine
    configuration, retrieving 178
described 49
    distributed 11
event processing 49
    knowledge base 321
performance tips 51
processing change rules 50
processing timer rules 51
transaction processing 51
rule language predicates
    add_to_repeat_count 89
    all_clear_targets 90
    all_duplicates 91
    all_instances 93
    any_clear_target 94
    any_clearing_event 95
attr_condition 96
    attr_exception 98
    attr_sequence 99
BAROC manipulation predicates 83
    bo_add_at_slotval_begin 100
    bo_add_at_slotval_end 101
    bo_get_class_of 102
    bo_get_class_slots 103
    bo_get_enum_options 105
    bo_get_slotval 106
    bo_is_defined_for_class 107
    bo_is_direct_super_of 108
    bo_is_super_of 109
    bo_remove_from_slotval 110
    bo_reset_default_slotval 112
    bo_set_slotval 113
cancel_all_timers 114
cancel_timer 115
categories 80
    change_event_administrator 117
rule language predicates (continued)
    change_event_severity 118
    change_event_status 119
    check_all_thresholds 120
    check_and_increment_count 121
    check_event_criteria 123
    check_threshold 125
clear_closed_events 127
clears 128
commit_action 130
commit_rule 131
commit_set 132
cancel_all_timers 114
    convert_ascii_time 133
    convert_gm_time 134
    convert_local_time 135
create_cache_search_criteria 136
create_clearing_event 138
create_event_criteria 141
create_event_sequence 144
described 8, 80
deprecated 8, 80
drop_change_request 151
drop_received_event 152
eraseGlobals 153
event activity predicates 84
event cache search predicates 81
event correlation predicates 80, 242
event criteria predicates 84
event criteria predicates 84
event criteria predicates 84
event criteria predicates 84
event criteria predicates 84
exec_program 154
eexec_program_local 156
eexec_task 159
eexec_task_local 162
first_causal_event 166
    first_duplicate 167
    first_effect_event 169
first_instance 170
    first_related_event 172
flow control 83
    forward_event 174
general purpose predicates 87
generate_event 176
generate_event 176
generate_event 176
generate_event 176
get_attributes 177
global variable predicates 86
    global_exists 186
increment_slot 187
init_count 188
init_event_activity 190
ip_node_unreachable 192
is_clearing_event 193
link_effect_to_cause 194
load_globals 196
log_error 197
online information 80
place_change_request 199
print_cache 200
print classify tree 202
print_event_activity 203
quick reference 80
rule language predicates (continued)
re_after_match 205
re_before_match 206
re_create 207
re_mark_as_modified 208
re_match 209
re_search_string 211
re_send_event_conf 212
re_split_event_id 213
re_substitute 214
re_substitute_global 215
redo_analysis 216
regular expression predicates 85
remove_bslashes 218
reset_event_activity 219
reset_global_grp 220
resolve_time 221
save_globals 223
search_cache 224
See also Prolog predicates or predicates 330
set_detailed_debugging 225
set_event_administrator 226
set_event_message 227
set_event_severity 229
set_event_status 230
set_global_var 231
set_log_error_source 232
set_timer 233
threshold predicates 84
time manipulation predicates 86
trace_it 234
unlink_from_cause 235
update_event_activity 236
rule packs
described 13
example 16
files 53
import 14
list 14
naming conventions 53
structure 15
rule sets
creating a new rule set, rule builder procedure 283
default 13
default rule base
activating detailed debugging 272
criteria, defining and determining for events 273
event correlation 276
event forwarding 276
heartbeat monitoring 274
preventing display of nested predicates 273
redirecting Prolog messages to files 272
restoring the event cache 273
rule engine shutdown 277
rule engine startup 272
threshold, defining criteria 274
default rule baser
maintenance mode processing 275
default rule set
described 271
described 13
editing a rule set, rule builder procedure 284
files 53
flow control 131, 132
import 14
list 14
modify 17
rule sets (continued)
naming conventions 53
NetView
clearing events 267
clearing TEC_ITS_SA_STATUS events 267
correlation of network and non-network events 269
correlation rules described 268
described 265
events received 265
synchronization rules described 266
synchronization, acknowledged events 266
rule builder, created with 279
saving a rule set, rule builder procedure 300
structure 15
Tivoli Availability Intermediate Manager
event activity report, activating 273
record predicates 272
rule types
change rules 63
See also change rules 63
compound rules 295
correlation rules 69
See also correlation rules 69
plain rules 61
See also plain rules 61
simple rules 67
See also simple rules 67
timer rules 65
See also timer rules 65
rules
/* */ (comment delimiters) 59
actions
action 78
action body 78
action name 78
action type 78
change rules 78
correlation rules 69
described 78
examples 10
flow control 130
naming conventions 53
plain rules 78
processing order 78
reception_action 78
redo_action 79
simple rule 67
timer rules 78
types 78
analysis
profile 80, 261
trace 80, 256
change rule filters
See change rule filters 74
change rules 50, 63
See also change rules 50
comments 59
compound rule, creating, rule builder procedure 295
compound rules 295
constants 59
correlation rules 69
See also correlation rules 69
correlation, event sequence 144
correlation, specifying with compound rule 297
counters 121, 188
debug
logging error messages 197, 232
rules (continued)
default rules (continued)
  tracing rules 256
described 8
directives
described 79
distributed rule base 11
errors, logging 197, 232
event filters
  See event filters or change rule filters or timer rule filters 71
event sequence 144
examples
correlation with event sequences 237
  rule builder 305
exterior event 154, 156
executing a task 159, 162
flow control 130, 131, 132
free, described 324
global variables, described 55
  See also global variables 55
ground, described 324
instantiated, described 324
language 8
launching a program 154, 156
launching a task 159, 162
naming conventions 53
parser grammar 60
plain rules 61
  See also plain rules 61
predicates, described 8
  See also rule language predicates or Prolog
  rules 8
processing events 49
profiling rules 261
Prolog
language 8
  rules 320
  See also Prolog 8
  using in rules 319
regular expressions
  See regular expressions 207
running a program 154, 156
running a task 159, 162
simple rules 9
  See also simple rules 67
structure
  change rules 63
correlation rules 69
genral 10
plain rules 61
  simple rules 67
timer rules 65
syntax 60
template
  See rule language predicates 88
testing rules
  sending events with commands 253
  sending events with SendEvents program 254
  sending events with tec_agent_demo program 253
timer rule filters
  See timer rule filters 76
rules (continued)
timer rules 51, 65
  See also timer rules 51
trace example 259
tracie multiple solution event filters 261
tracie rules 256
tracie user-defined Prolog predicates 234
types
  change rules 8
  compound rules 9, 295
correlation rules 9
plain rules 8
simple rules 9
timer rules 9
unification, described 324
upgrading from previous releases 279
variables
  global 55
  instantiated 54
  naming conventions 54
  quotation marks 54
  reading from file 54
  regular 55
  scope 55
  storage 54
  uninstantiated 54
  uppercase names 54
  writing to file 54
rules cache
  See event cache 7
running a program from rules 154, 156
running a task from rules 159, 162
S
saveGlobals rule language predicate 223
saving a rule set, rule builder procedure 300
search event cache 136, 224
searchCache rule language predicate 224
search, reverse lookup 241
sender filter, change rule 74
SendEvents program 254
sending events 174, 212
server_handle attribute 6, 7
server_path attribute 7
server_path attribute parsing 213
setDetailedDebugging rule language predicate 225
setEventAdministrator rule language predicate 226
setEventMessage rule language predicate 227
setEventSeverity rule language predicate 229
setEventStatus rule language predicate 230
setGlobalVar rule language predicate 231
setLogErrorSource rule language predicate 232
setTimer rule language predicate 233
setTo operator 75
severity attribute 118, 229
severity, changing 118, 229
severity, setting 229
simple rules
  creating a simple rule, rule builder procedure 285
described 9
event actions 67, 292
event conditions 67, 289
structure 67
simple_type 42
slashes, converting 218
sort Prolog predicate 412
sprintf Prolog predicate 413
starting the event server 304
starting the rule builder 282
state correlation 5
status attribute
  changing the value 119
  described 7
  setting the value 230
status, changing 119, 230
status, setting 230
stopping the event server 304
strip Prolog predicate 415
subclass, described 44
subset Prolog predicate 417
substring Prolog predicate 418
subtract Prolog predicate 419
superclass checking 108, 109
superclass, described 44
target events, clearing 90, 94
targets, rule base
  See rule base targets 12
tasks, executing from rules 139, 162
tec_agent_demo program 253
TEC_UMASK environment variable 36
tec.baroc file 17, 47
template
  See rule language predicates or Prolog predicates or predicates 80
template rules 271
  see also rule sets, default rule base 271
term_type Prolog predicate 420
terms, Prolog 319
test data type and variable Prolog predicate 326
testng rules 253
  See also rules, test 253
text editor, use with creating rules 279
textual checking 120, 125
threshold checking 149
threshold rule language predicates 84
time
  epoch number to GMT, converting 134
time manipulation rule language predicates 86
timer rule filters
  described 76
timer_duration filter 77
timer_info filter 76
timer rules
  described 9
  processing of 51
  structure 65
timer_duration filter 77
timer_info filter 76
timers
  canceling 114, 115
  described 9
  setting a timer 233
Tivoli Availability Intermediate Manager
  rule base example 18
Tivoli Customer Support xi
Tivoli Enterprise Console Rule Builder
  See rule builder 279
trace 225, 256
trace directive 80
trace_it rule language predicate 234
transact rule execution 80, 256
tracing user-defined Prolog predicates 234
transaction processing, rule engine 51
unification, described 324
union Prolog predicate 421
UNIX directory permissions for rule bases 36
unlink_from_cause rule language predicate 235
unlinking cause and effect events 235
update_event_activity rule language predicate 236
upgrade_gui.sh command 279
upgrading rule bases 11
upgrading rules from previous releases 279
user_predicates file 424
UTF-8 support 37, 48, 279
variables
  _ (Prolog anonymous variable) 71
  _class 71
  _event 71
  environment variables, notation for xiii
  environment variables, TEC_UMASK 36
  event filters, in 71
  global variables
    See global variables 55
    Prolog variables, described 319
    regular variables 55
    rules 54
    See also rules, variables 54
    scope in rules 55
    TEC_UMASK environment variable 36
where clause, event filter 73
wpostemsg command 5, 253
wr command
  creating rule base targets 13
  creating rule bases 16
  usage recommendation 11
wrbupgrade command 11
write Prolog predicate 422
wtdumprl command 3