

Skulker Version 2 - Rule Configuration Guide

Version: 1.8
Date: September, 2009.

Version History

Version	Date	Author	Changes
1.0	12 th December, 2006.	Simon Edwards	Original.
1.1	8 th June, 2007.	Simon Edwards	New DIR_* functions support. Information on limit support and maximum file counts.
1.2	10 th August, 2007.	Simon Edwards	Minor update covering %ext and %barename macros and additional compress "target" option.
1.3	20 th November, 2007.	Simon Edwards	Updates covering multiple directory matching and flexible directory level pattern matching.
1.4	15 th April, 2008.	Simon Edwards	Use of MAGIC including the new contains() function.
1.5	22 nd September, 2008.	Simon Edwards	Include information on "generators".
1.6	January, 2009.	Simon Edwards	Improved documentation covering protected directory handling. Documentation of time and size suffixes. New date_from_name function.
1.7	May, 2009.	Simon Edwards	Implementation of "move" function and delete file scrubbing. Information on unsafe option and improved directory macro.
1.8	September, 2009.	Simon Edwards	Document the newly added "append" function.

Contents

1	INTRODUCTION	4
1.1	WHAT IS SKULKER?.....	4
1.2	PURPOSE OF RULE CONFIGURATION GUIDE.....	4
2	RULE CONFIGURATION FILES - AN OVERVIEW	5
2.1	STRUCTURE OF XML FILES.....	5
2.2	INCLUSION OF OTHER FILES	5
3	SKULKER FILE MATCHING.....	7
3.1	BASIC FILE NAME MATCHING	7
3.1.1	<i>Specifying Multiple Directories</i>	8
3.1.2	<i>Directory Level Pattern Matching</i>	9
3.1.3	<i>Matching Files and Directories with spaces</i>	10
3.2	LIMITING RULE PATTERN MATCHING.....	10
3.3	OPPOSITES MATCHING.....	10
3.4	LIMIT FILE PROCESSING	12
3.5	LIMITING RECURSIVE SEARCHING	13
3.6	MORE ON PROTECTED DIRECTORY SUPPORT.....	13
3.7	CUSTOMISED FILE/DIRECTORY MATCHING	14
3.8	SUMMARY OF FILE PROCESSING	16
4	RULE TYPES	17
4.1	DELETE RULES	17
4.1.1	<i>Use of “usermasq” Facility</i>	18
4.1.2	<i>Understanding File Scrubbing Support</i>	18
4.2	COMPRESS RULES.....	22
4.2.1	<i>Using “ifexists” Option</i>	23
4.2.2	<i>Using “target” Option</i>	23
4.2.3	<i>Using the “usermasq” Option</i>	24
4.3	TAIL RULES	25
4.3.1	<i>Use of the “maxsize” Option</i>	26
4.3.2	<i>Use of the “usermasq” Option</i>	26
4.4	ROTATE RULES	27
4.5	DELETEDIR RULES	28
4.6	MOVE RULES	29
4.7	COMMAND RULES	30
4.8	MAILFILE RULES	31
4.9	ORGANISE RULES	32
4.10	ARCHIVE RULES	33
4.10.1	<i>Limitations of “ar” archive format</i>	33
4.11	SASWORK RULES	33
4.12	SPDSWORK RULES.....	33
5	CONDITIONAL EXECUTION	34
5.1	CONDITIONAL INCLUDES	34
5.2	IGNORING RULES ON FAILURE CONDITIONS	34
6	ADDITIONAL RULE FILE SYNTAX.....	35
6.1	FILE NAME MANGLING MACROS.....	35
6.1.1	<i>Examples of Using Partial Directory Matching</i>	36
6.2	INTERVAL DEFINITIONS	37
6.3	PATTERN MATCHING FUNCTIONALITY	38
6.4	FILE CRITERION MATCHING FUNCTIONALITY.....	40
6.4.1	<i>File Attribute Macros</i>	40
6.4.2	<i>Directory Attribute Macros</i>	41
6.4.3	<i>Relational Operators</i>	41

6.4.4	<i>Functions</i>	42
6.4.5	<i>Unimplemented File Attribute Macros</i>	43
6.4.6	<i>Literal Values</i>	43
6.4.7	<i>Example Criteria</i>	45
7	EMAIL INTEGRATION	46
7.1	GENERATING TEXT IN EMAILS	46
8	USE OF FILE TYPE DETECTION [“MAGIC”]	47
8.1	WHAT IS “MAGIC”?	47
8.2	LIMITATIONS OF “MAGIC” HANDLING	47
8.3	TESTING WHETHER “MAGIC” WORKS ON NECESSARY FILES	47
8.4	USING “MAGIC” ON SKULKER RULES	48
9	WRITING A GENERATOR	50
9.1	WHY WRITE A GENERATOR?	50
9.2	WHAT A GENERATOR SHOULD INCLUDE?	50
9.3	AN EXAMPLE GENERATOR	50
9.3.1	<i>The “new” routine</i>	51
9.3.2	<i>The “check” subroutine</i>	53
9.3.3	<i>The “generate” subroutine</i>	54
9.4	BASIC SYNTAX CHECK	54
9.5	CHANGES NECESSARY TO PICK UP GENERATOR	54

1 Introduction

1.1 *What is Skulker?*

“Skulker” is a tool that has been designed to automatically manage many files that either grow in size or number over a period of time whilst the Operating System runs and applies a series of actions against these files to ensure the disk space they occupy is automatically managed.

Skulker works by applying a series of rules that define the files to apply a particular rule again, the frequency that a rule can be run and optional logic that must be matched for a particular file is considered a candidate for application by the rule.

Skulker is entirely written in Perl and has been written to work across any number of UNIX variants, including HP-UX, AIX, Solaris and Linux. It supports multiple levels of logging, accurately evaluates (and reports on) space saved, and offers a simply syntax based on XML files.

Skulker also provides a “preview” mode allowing the administrator peace of mind regarding what files will be impacted by the default rules. The environment is extensible - the administrator can change the existing rules or add further rules if they wish.

1.2 *Purpose of Rule Configuration Guide*

This document describes in detail the standard functions that Skulker offers by default and the syntax of rule configuration files. It makes use of entries from the standard rules offered for illustrations where possible rather than just providing reference material.

It also describes the process of how the standard configuration uses the rules files. Rules files are quite straightforward XML format, however they offer a reasonably powerful syntax including;

- Time Definitions
- Size Definitions
- Rule Frequency definitions
- File Name Pattern Matching
- File attribute expression evaluations

All these are also explained in detail, along with suitable real-world examples when possible.

2 Rule Configuration Files - An Overview

This section provides a short introduction to Skulker Rule Configuration files - it is useful for those who do not intend to make large changes, instead are happier to “tweak” or copy existing rules and make small modifications to them.

2.1 Structure of XML files

If Skulker is to perform any action against a series of files or directories those actions must be contained in a configuration file - no command line options to specify this level of detail are available.

All configuration files are XML files, since this has several benefits;

- It is a commonly used format
- It is well structured and can be parsed easily by machines
- It is human readable and easy to edit

A Skulker configuration file has the following format;

```
<?xml version="1.0" standalone="yes"?>
<skulker_rules>
  <defaults
    ... default settings here
  />

  <rule
... rule details
  />
  <rule
... rule details
  />
</skulker_rules>
```

The number of “rule” elements is not limited, though each must have a different number. The “defaults” setting is useful since it allows several common settings to be defined and in this case they do not need to be re-specified for each rule - except when they differ for that rule of course.

2.2 Inclusion of Other Files

Rules can also be spread over separate files if you wish. In this case the same number can be used in different files - it only needs to be unique in each file in question.

Since rules can be spread across separate files there is a mechanism for one rules file to “include” rules from a different file. In this case the rules files can contain lines of the following format:

```
<include [iftrue="Condition]" file="file.xml"/>
```

Notice that the “iftrue” attribute is optional and does not need to be specified. A rules includes can use the “include” element anywhere the “rule” elements are specified. There are no limits to the number of include elements are present.

Please note that a rules file that is included can itself include further rules files - though Skulker keeps a track of which have already been included and will silently ignore any attempt to include the same rules file more than once [to prevent any circular or recursive references].

The only limitation with include files is that they can “only” be nested to a depth of 30 files - which is something that is never likely to be an issue.

This mechanism is used in the “MAIN.xml” file which is the default rules files that is configured:

```
<?xml version="1.0" standalone="yes"?>
<skulker_rules>
  <include iftrue="UNAME eq 'HP-UX'" file="hpux.xml"/>
  <include iftrue="UNAME eq 'Linux'" file="linux.xml"/>
  <include iftrue="UNAME eq 'AIX'" file="aix.xml"/>
  <include iftrue="UNAME eq 'SunOS'" file="solaris.xml"/>
  <include file="hostonly-%H.xml"/>
  <include file="skulker2.xml"/>
</skulker_rules>
```

3 Skulker File Matching

Skulker provides several directives in an effort to be powerful and flexible when wishing to specify which files should be considered for processing for a particular rule. This section describes the series of directives that can be used with any rule type to describe files to consider processing.

3.1 Basic File Name Matching

The most basic option for indicating the files to process [which must always be present], is the “match_pattern” attribute. A basic setting might be:

```
match_pattern="/var/log/*.log"
```

Notice that it contains a directory portion and a filename portion. The directory portion must be static and can not contain any pattern matching syntax. The filename portion can include simply shell meta-characters or Perl-based patterns.

When necessary Skulker will convert simple shell pattern matches to the equivalent Perl patterns when running to ensure the matches work as expected. Thus to match all files in a directory the following might be used:

```
match_pattern="/var/log/*"
```

Skulker also includes a mechanism for matching files from a set of directories via recursive matching. A recursive search is performed when the last directory element is in either of the following two forms:

```
match_pattern="/directory/.../pattern"
```

```
match_pattern="/directory/...[min,max]/pattern"
```

Hence to scan all directories under /var for files ending in “.log”, the following pattern could be used:

```
match_pattern="/var/.../*.log$"
```

Notice the final “\$”? Skulker would have added this by default - but it is always useful to specify it! The “min” and “max” settings in square brackets are used to limit the recursion and describe the depth below the current directory to scan. “0” is the current directory and “1” is the first level of sub-directories.

Min can be ignored and if so is set to “0”. The default for “max” if not set is “999”. Hence the following three pattern matches are identical:

```
match_pattern="/var/*.log$"
```

```
match_pattern="/var/...[,0]/*.log$"
```

```
match_pattern="/var/...[0,0]/*.log$"
```

Hence to scan just the current directory and two levels down the following match might be used:

```
match_pattern="/var/...[,2]/.*\.log$"
```

The other useful tool when matching files is the use of the optional attribute “pattern_match_ignore”. This is applied to any file that matches the pattern via the “match_pattern” attribute. If it matches the “pattern_match_ignore” name format then it is removed from the list of files to process.

For example consider the following two settings:

```
match_pattern="/var/tmp/.../*"  
match_pattern_ignore="\.(keep|data)$"
```

Notice that the match_pattern_ignore is compared to file names - no directory component should be present [it will never match if you do include one!]. The above will match all files under “/var/tmp” - but will then remove all files that end in a “.data” or “.keep” extensions.

3.1.1 Specifying Multiple Directories

All the above examples show how a single directory and a pattern can be used to match a series of files. If a series of different directories are to be processed then they should be white-space separated - though no white space should occur at the start or the end of the item. For example consider the following rule:

```
<rule  
  n="1020"  
  interval="6H"  
  type="delete"  
  match_type="file"  
  match_pattern="/tmp/dir1/* /home/dir2/* /var/tmp/dir3/*"  
  match_by="MTIME > 1D"  
>
```

Multiple white space can be used - the number does not matter - they can even be put on separate lines, for example:

```
<rule  
  n="1020"  
  interval="6H"  
  type="delete"  
  match_type="file"  
  match_pattern="/tmp/dir1/*  
/home/dir2/*  
/var/tmp/dir3/*"  
  match_by="MTIME > 1D"  
>
```

Please note - currently having pattern matches in the directory part is not yet supported.

3.1.2 Directory Level Pattern Matching

Skulker 2 also supports pattern matching at the directory level as well as the file name file. In some cases this can be particularly useful! The functionality is best illustrated with an example:

Consider the following files and directories

```
/files/dir1/tmp/aa  
/files/dir1/keep/bb  
/files/dir2/tmp/cc
```

Now consider the requirement to remove files in the “tmp” sub-directories. The “pattern_match” attribute could be written as:

```
pattern_match="/files/dir1/tmp/* /files/dir2/tmp/*"
```

This is fine, but if a further directory in the same format as “dir3/tmp” is added the pattern_match” would then need to be changed:

```
pattern_match="/files/dir1/tmp/* /files/dir2/tmp/* /files/dir3/tmp/*"
```

Directory level pattern matching allows that to be overcome. To match all “dirN” temporary directories the better approach would be to use:

```
pattern_match="/files/dir[0-9]+$/tmp/*"
```

What about the situation where the directory to scan files for under “dirN” was not “tmp” but everything ending in “.tmp”, the following patterns can be used:

```
pattern_match="/files/dir[0-9]+$/.*\.tmp$/*"
```

Any component in the path can be given as a pattern rather than a directory. It can even be combined with the recursive “...” operator, for example:

```
pattern_match="/files/dir[0-9]+$/.../.*\.tmp$"
```

In this case it would match all files under any “dirN” directories at whatever level that name a name ending in “.tmp”.

3.1.3 Matching Files and Directories with spaces

Allowing multiple directories to be included in the same rule is very useful - but it leads to the question of how directories and file names with spaces [quite common these days] can be handled correctly.

The key is the use of the special sequence “\s” - this will match a single space. Hence to delete some files from a users’ “My Documents” directory a rule similar to the following might be used:

```
<rule
  n="1020"
  interval="6H"
  type="delete"
  match_type="file"
  match_pattern="/home/user1/My\sDocuments/*\s.tmp"
  match_by="MTIME > 1D"
/>
```

Of course the limitation of not being able to put “/home/*/My\sDocuments” is a large one which should be resolved in later editions.

3.2 Limiting Rule Pattern Matching

Sometimes it is advantageous to be able to ensure that a particular rule does not impact more than a particular number of files. For example you might have a directory which contains log files and rather than deleting all files older than a certain age, perhaps instead you wish to delete the oldest 30 files, or keep only the newest 20 files and delete the rest.

This type of functionality is made available by two optional attributes that all rule types support:

```
match_pattern_limit="count"
match_pattern_limit_by="age|name|reverse_age|reverse_name"
```

The “match_pattern_limit_by” is only used if “match_pattern_limit” is configured for a rule, and if not set will default to “age”.

When “match_pattern_limit” is set only that number of files matching the pattern will be processed. Of course this list is then passed by any expression defined in the “match_by” logical expression which may further restrict the list of files that are processed.

It is also possible to give a negative value as well - in this case all files apart from the number specified are processed. For example if a pattern matched 30 files, and the “match_pattern_limit” was “-10” - then 20 files would be processed.

3.3 Opposites Matching

When there are lots of different things to match it is sometimes easier to indicate the files you do **not** want to match! For reason there is a directive called “match_pattern_opposites”, and it might be used as:

```
match_pattern="/tmp/*\s.save"
match_pattern_opposites="yes"
```

When a rule with that configuration is processed all files that do **not** match the “*.save” will be matched! The value to give to “match_pattern_opposites” are Boolean values, such as “true”, “yes” or “1” or “false”, “no” or “0”.

Please note that this opposite matching refers to the filename portion of the match, so any directory pattern matching higher up works as expected.

By default the list that is processed are the oldest files - the newest ones are ignored. This equates to setting “match_pattern_limit_by” to “age”. The available settings for this attribute are:

Setting	Meaning
age	The list of files generated is sorted by modification time - and the newest files are excluded from the list. This is the default since it makes the most sense to process older files in almost all cases.
reverse_age	The reverse of the above; the oldest files are excluded from the list and just the newer ones are processed.
name	The files are sorted in name order - the files with last letters of the alphabet are dealt are removed from the list.
reverse_name	The reverse of the above.

When using “name” or “reverse_name” it is worth remembering that the **complete path** of each entry is used as part of the sorting process - So if a recursive search is being performed then the list might include files from sub-directories before files in the top-level directory depending on the names of the directories themselves.

3.4 Limit File Processing

The “pattern_match_limit” process ensures that the number of files matching a particular pattern is limited. There is another related attribute that can be set that applies a limit of processed files, rather than matched files.

The difference might sound subtle; but the process of how Skulker processes files should be understood first. It does the following;

- The specified directory [recursive or otherwise] is scanned and a list of files matching the pattern is built up.
- The number of elements in this list is reduced to a size of “pattern_match_limit” files if this is set. This implicitly sorts the data whether or not a “pattern_match_limit_by” is explicitly set or not.
- The “pattern_match_by” expression is applied to each file and any not meeting the criteria are discarded.
- If “max_files_count” is set the list of matched files is reduced down to this value, discarding the entries at the end of the list.

So setting “pattern_match_limit” will reduce the list of files to process based on the name, but the actual number of files processed might be far less - depending on what expression the files then have to match.

Setting “max_files_count” only will ensure a maximum number of files to process, no matter the number matched [or any limit by which this is restricted to].

Both enforce maximums at different points and so both will be useful in different circumstances. Both can be used at once - indeed using them together is often the most useful because it will ensure the list of files is sorted rather than “random”.

3.5 Limiting Recursive Searching

The ability to perform recursive searching is a very powerful mechanism, and the previous two sections describe mechanisms for limiting the amount of matching and processing that might be performed when a pattern matches a large set of files.

Of course the “protected_dir” settings can be used to ensure the recursive mechanism does not traverse into particular directories - but there is a more flexible way of limiting traversal which can be used on a per-rule basis - “pattern match truncation”.

The “match_pattern_truncate” attribute can ensure that when performing recursive directory searches any sub-directory matching the specified pattern is dropped from the list of files/directories to consider. This action is recursive - if a particular directory matches the truncation pattern all files and sub-directories are ignored.

Notice that the item attempts to match directory names rather than file names. Hence the attribute only takes affect when recursive pattern matching is taking place. For example consider the following two attributes for matching files:

```
match_pattern="/data/build/.../*"  
match_pattern_truncate="archive"
```

The above recursively matches all files under the “/data/build” directory. However if the search finds any sub-directories called “archive” these will be ignore from whatever the rule intends to do.

Of course because this is a pattern the usual rules for Perl pattern matching can be used, for example:

```
match_pattern="/data/build/.../*"  
match_pattern_truncate="^(archive|safe|keep)$"
```

The above truncates directories from the recursive matching if they are called “archive”, “safe” or “keep”.

3.6 More on Protected Directory Support

Protected directory support works on two levels; if a particular rule specifies matching only files from a particular directory then just that directory is checked. However if a rule makes sure of recursive matching it checks on a per-directory basis too.

When recursive directory pattern matching is in use then when each directory is entered it is compared to the list of directories that are considered protected. If a directory is found, either explicitly or mentioned via a recursive protected directory then **the specified directory tree is ignored**.

The above means that when using recursive pattern matching if “/usr/keep” is specified as a non-recursive protected directory, scanning “/usr” recursively will ignore sub-directories of “/usr/keep” too - even though a sub-directory explicitly mentioned in a rule will be allowed!

3.7 Customised File/Directory Matching

So far much information has been given on how files and directories can be matched using patterns and other directives to modify what happens to the list of matched files. This approach is very powerful, but Skulker has another way of matching files too - “generators”.

Generators are small Perl modules that take parameters and return a list of files based on those parameters. Currently there are four generators included, though a section at the end of this document describes the process of how further customised generators can be written.

- `olderthanboot(dir, ...)` - Scan the specified directories and return all files or directories that are older than the time of the last machine boot.
- `olderthanboot_recursive(dir, ...)` - Same as the above, but recursive scan and include results from any sub-directories too.
- `invalid_user_recursive(dir, ...)` - Scans the specified directories and returns those objects contained which do not belong to a valid user on the system. Useful for ensuring files from removed users are caught and processed appropriately.
- `empty_dirs(dir)` - Recursively checks the specified directory for empty sub-directories and return them as a deepest-first list. Typically used with the “deletedir” functionality. Never returns the name of “dir” - only applicable sub-directories.

The generators can take note of the current type of object the rule is matching if they wish, so by default they might return files, or directories if the `match_type` is set to "directory". This needs to be set - for example using "empty_dirs" without setting the `match_type` to "directory" will always return an empty list!

Two example rules that make use of this feature can be found in the "AIX" and "HP-UX" rules:

```
<rule
  n="1040"
  type="delete"
  match_pattern="@olderthanboot_recursive(/tmp) "
  ignore_on_file="1041"
/>
<rule
  n="1041"
  type="deletedir"
  match_type="directory"
  match_pattern="@olderthanboot_recursive(/tmp) "
/>
```

The first rule above scans all files in "/tmp" and returns a list of any older than the time of last reboot, whilst the second one handles directories.

A few points regarding generators:

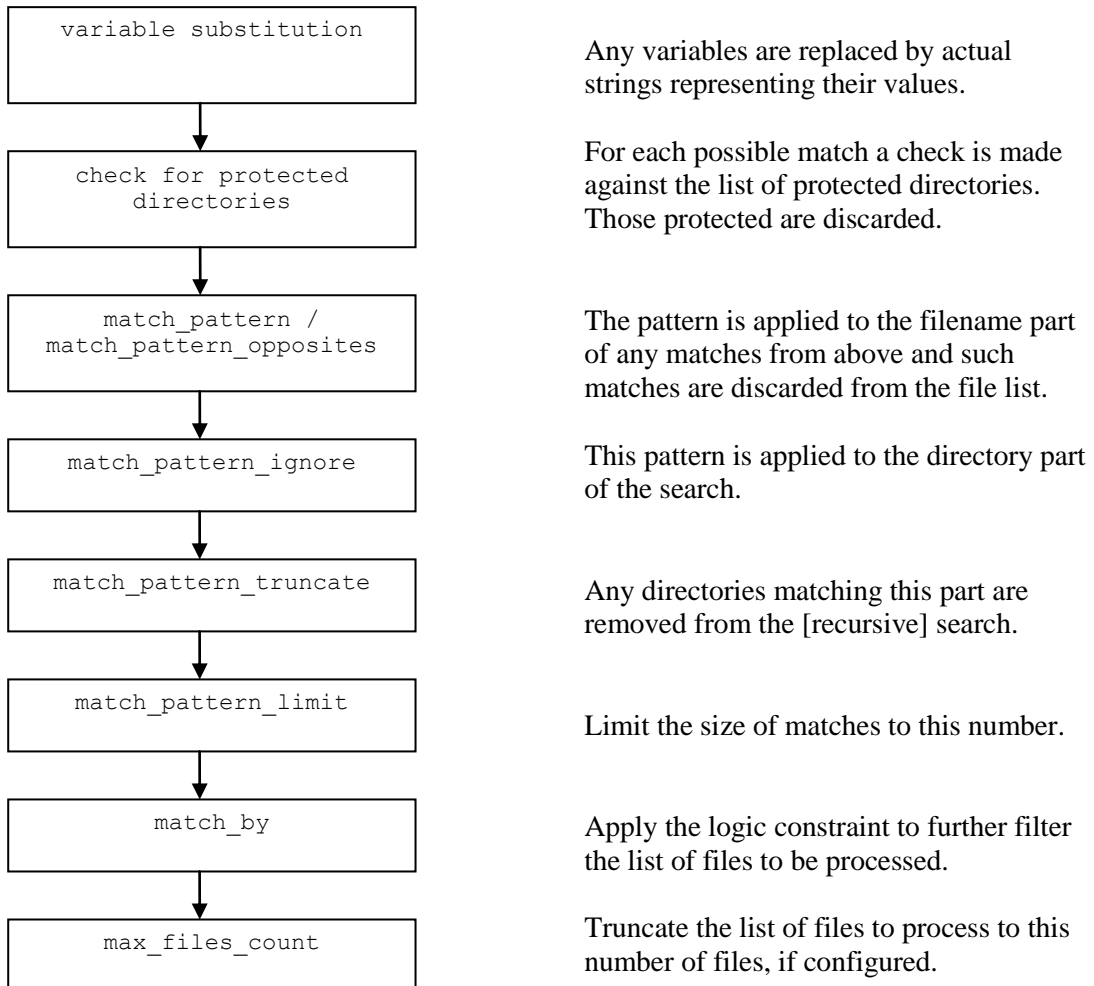
- Notice that the generator is preceded by a "@" symbol and must include a "(.)" following it - even if no parameters are needed.
- The arguments specified are passed in without change - so patterns are not expanded into lists prior to processing.
- If a generator does not exist the processing of the Skulker ruleset will be aborted when that particular rule is reached. For example:

```
2008/09/22 12:04:40 LOG      Rule hpux-tmplogs:1040  Rule action      : delete
2008/09/22 12:04:40 LOG                        Match type      : file
2008/09/22 12:04:40 LOG                        Match pattern   :
@bad_generator(/tmp)
2008/09/22 12:04:40 LOG                        Match by       : ALWAYS
2008/09/22 12:04:40 LOG                        Last run      : Never
2008/09/22 12:04:40 LOG                        Run interval   : 1D
2008/09/22 12:04:40 LOG
[Files: 0, Duration: 00:00:00, Space: 0 Kb]
2008/09/22 12:04:40 ERROR  Generator 'bad_generator' not found - aborting.
```

- Generators will ignore the other pattern match attributes apart from variable substitution.

3.8 Summary of File Processing

The various attributes discussed can be summarized via the following flow diagram:



4 Rule Types

The rules available with Skulker will change over time, but currently include 10 different ones - though this does include two sample application specific rules. Each of the available rules will now be described. Following the rule definitions the other aspects of the rule syntax, including expressions, pattern matching and interval definition will also be shown with suitable examples.

It is also possible for administrators to define their own rules - though this does require knowledge of Perl. An additional document describing this process is also available - "Skulker version 2 - Rule Function Creation Guide".

4.1 Delete Rules

These rules are probably the most common - and as the name suggests any files that match the criteria for the rule are removed. Such rules in a rule file have the following definition [optional details shown in italics].

```
<rule
  n="1000"
  interval="1D"
  type="delete arguments"
  match_type="file"
  match_pattern="/var/opt/perf/datafiles/archive/.*\.Z$"
  match_pattern_truncate="pattern"
  match_pattern_ignore="pattern"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

If not specified Skulker will match files and directories that match the specified pattern. Hence it is typical to include a `match_type` setting to ensure just files or just directories are matched.

This is optional and if not present, but the rule file contains a `match_type` in a defaults section that will be used. If either is not present the delete function will default to "file".

The `match_by` is an expression that determines any criteria that must apply to the file names that match the pattern specified by the `match_pattern` setting.

The `interval` again is optional and if not present a value is expected in a defaults section in the rule file. If that is not present an error will be given and the Skulker process abort. A section describing allowed `interval` settings can be found in section 6.2 on page 37.

The other item that might be present are arguments, which are white-space separated and can include the following for “delete” rules:

truncate	Rather than deleting the file it will be truncated to 0 bytes. This is useful since some utilities will only log if the specified file exists. It also means that if the file is open in “append” mode by a running process it will not “disappear” and yet continue to consume disk space.
unsafe	Over rides the protected directory settings if possible for the files to attempt to match.
touch	An alias for the above option.
usermasq	Switch to specified user to delete a matched file [see below]
handleopen	An alias for the above option.
reportonly	Never make only changes – just record what would happen.
emailto	Send the copy of any changes [or report content] to the specified users.
scrubbing	Defaults to “false” but if set to true it will overwrite the file before removing it.

Information on the “emailto” argument can be found in section 7 on page 46.

4.1.1 Use of “usermasq” Facility

When Skulker is run as “root” and comes across an NFS directory structure which the local machine does not have root access to deleting files can occasionally file with errors like the following.

```
2007/11/21 09:34:16 WARN Unable to delete file /xfertmp/simon/file1.gz!
2007/11/21 09:34:16 WARN Unable to delete file /xfertmp/simon/file2.gz!
```

To overcome such problems the additional “usermasq” [which can also be referred to as “nonroot”] should be used in the options. When this action takes affect it will check each file before attempting to delete it will attempt to switch to using the user ID in question as the effective user - allowing the files to be deleted:

```
2007/11/21 09:36:47 LOG Delete: Removed - /xfertmp/simon/file1.gz
[simon:sys]
2007/11/21 09:36:47 LOG Delete: Removed - /xfertmp/simon/file2.gz
[simon:sys]
```

There is no performance lost using this option though it is recommended it should only be added when absolutely necessary.

4.1.2 Understanding File Scrubbing Support

File scrubbing is an entirely optional feature that will cause the complete contents of the file being removed to be over-written by NULL values prior to the deletion taking place. There are two possible reasons for turning on this facility:

- **Security** – by default many UNIX file systems do not remove the contents of deleted files when the file is “unlinked”. It may be possible to retrieve the data via a block-level read for example.
- **Disk Space Reclamation** – some virtualised storage sub-systems are able to reclaim storage to their pooling mechanisms when large contiguous regions of the file system consist only of NULL values.

If logging level of “3” is used then each scrubbing file will be reported on, for you will see entries similar to the following for each file being deleted:

```
2009/04/29 09:54:31 LOG      Delete: Scrubbed file /tmp/skulkerv2-tests-  
25770/files/scrubit/testfile.  
2009/04/29 09:54:31 LOG      Delete: Removed - /tmp/skulkerv2-tests-  
25770/files/scrubit/testfile [simon:users]
```

Obviously there are some performance over-heads from using this facility – especially if the files being removed are large. Hence the recommendation is to only use if when deemed necessary rather than turning it on globally via the configuration file.

4.2 Append Rules

This is a very basic rule; it simply takes the contents of one file and appends them to another. Typically this rule is combined with two others; for example:

- Append “fileA” onto “fileB”
- Tail “fileB” to keep a particular line count
- Delete “fileA”

Typically such a series of actions suit a situation where daily log files are generated and need to be appended to keep just a single file containing the log entries in question.

Since this rule actually appends to a file (creating it if it does not exist), it actually increases disk space usage, though of course subsequent rules (in the above example scenario), reverse the overall picture.

Please remember that due to the transparent use of compressed files the target file to append to can be a compressed file. Further the source file can also be compressed to.

A sample “append” rule might appear as:

```
<rule
    n="1010"
    type="append arguments"
    interval="1D"
    match_type="file"
    match_pattern="dirname/pattern"
    match_pattern_ignore="expression"
    match_pattern_truncate="expression"
    match_pattern_limit="count"
    match_pattern_limit_by="age|name|reverse_age|reverse_name"
    max_files_count="N"
    match_pattern_opposites="no"
    match_by="expression"
/>
```

The list of supported arguments for this rule type are:

unsafe	Over rides the protected directory settings if possible for the files to attempt to match.
handleopen	In this case the action is a “no-op”, since Skulker does nothing to the source file in question. It is provided simply to ensure the syntax of the rule does not differ compared to other functions.
usermasq	Run the “append” function as the user that owns each source file that is queried or modified.
target	The name of the file to append to. This defaults to “%d/%f.log”, so appending typically appends to a file with the same name. Of course lots of matching files can be appended to the same log if necessary.

The target value can include any of the macros as defined on page 35 -so an example might be:

```
target=%d/%basename.Z
```

In the above case each matched file will generate a file in the same directory as the matched file with an extension of “.Z”. So “/tmp/fred.log.bloggs” would attempt to create or append to a file called “/tmp/fred.Z”.

4.2.1 Use of the “usermasq” Option

This option can be useful when attempting to use the “append” function on files which are on NFS shares. In such circumstances it is possible that even running as “root” problems such as the following occur:

```
2007/11/21 11:13:28 WARN  Unable to open file /xfertmp/simon/file1:
2007/11/21 11:13:28 WARN  Unable to open file /xfertmp/simon/file1:
2007/11/21 11:13:28 WARN  Permission denied
2007/11/21 11:13:28 WARN  Unable to open file /xfertmp/simon/file2:
2007/11/21 11:13:28 WARN  Unable to open file /xfertmp/simon/file2:
2007/11/21 11:13:28 WARN  Permission denied
```

Once the “usermasq” has turned on the next attempt at tailing the files should work as expected.

4.3 Compress Rules

As expected this function type will compress any file system object that it processes. Again it expects a `match_pattern` attribute to determine the files to process. A typically rule of this type will appear as:

```
<rule
  n="1010"
  interval="1D"
  type="compress arguments"
  match_type="file"
  match_pattern="/tmp/.../*"
  match_pattern_ignore="pattern"
  match_pattern_truncate="pattern"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

Please note that specifying `match_type` will not cause problems, but the default of “file” is only option that is likely to work (depending on file system type).

The following arguments are supported:

<code>type=TYPE</code>	Indicates the compression algorithm to use. This can be any of those registered in the Skulker configuration file. If this argument is not present the default compression type specified in Skulker configuration file is used.
<code>reportonly</code>	Do not make any changes – simply report on the changes that would be made.
<code>unsafe</code>	Over rides the protected directory settings if possible for the files to attempt to match.
<code>emailto</code>	Send a report of any activity [either in “reportonly” or normal modes] to the specified list of users.
<code>usermasq</code>	Run the changes as the source file owner [also referred to as “nonroot”]
<code>target=name</code>	Indicates an alternative name to call the created [or appended] compressed file. See section 4.3.1 below for details.
<code>ifexists=action</code>	What to do if the target file exists - can be set to “overwrite”, “ignore” or “append”. See section 4.3.2 below for details.

It should be noted that if this rule is passed a file that it already recognises as being compressed [by any of the registered types] it will be silently ignored.

The `match_type`, `interval`, `match_pattern_ignore`, `match_pattern_truncate` and `match_by` settings are described in the “Delete Rules” section above.

4.3.1 Using “ifexists” Option

This is a useful option which causes the “compress” rule to intelligently handle the situation when a target filename to compress to already exists. The default setting if the file exists is defined in the referenced configuration file and is distributed as “overwrite”. This of course can be over-written in the defaults section for the rules file itself, or can be changed in the configuration file. If a default does not exist a compress rule will fail [to ensure data integrity].

The options available are fairly obvious in their actions:

Action	Purpose
overwrite	If the target file already exists it is removed and overwritten with the newly compressed file.
ignore	If the target file exists then do not attempt the compression - the source file and the pre-existing compressed file will be left unchanged.
append	Append the contents of the compressed file to the existing file. This obviously is performed in a sensible manner - the result is a single compressed file rather than two compressed files appended on to each other. Useful for compressing text log files.

4.3.2 Using “target” Option

The target option allows the user to override the default name that is assigned to a file when it is compressed. The default name is simply the name of the file with a standard extension appended - the extension being deemed determined by the configuration of the compression type.

The option is very useful not only for giving the files a different extension to their names, consider the following examples:

```
target=%d/%d.compressed
```

The above will override the default extension given to the compressed file to call it “*filename.compressed*”.

```
target=/tmp/%f.Z
```

Creates the compressed version of the file in “/tmp” rather than in the directory where the file was originally found. It appends a “.Z” extension as well.

```
target=%d/%bname-%D.gz
```

The above creates a filename in the current directory with the “bname” of the file followed by the current date and finally a “.gz” extension. For example on August 10th, 2007 a file called “detailed.summary.log” would have a target name of “detailed-20070810.gz”.

4.3.3 Using the “usermasq” Option

As previously described for the “delete” function when Skulker encounters an NFS directory structure, even when running as “root” sometimes this fails if the host does not have NFS “root” permissions. Such errors are seen similar to the following:

```
2007/11/21 09:24:48 WARN    Unable to open /xfertmp/simon/file1 for reading:
2007/11/21 09:24:48 WARN    Unable to open file /xfertmp/simon/file1:
2007/11/21 09:24:48 WARN    Permission denied
2007/11/21 09:24:48 WARN    Unable to open /xfertmp/simon/file2 for reading:
2007/11/21 09:24:48 WARN    Unable to open file /xfertmp/simon/file2:
2007/11/21 09:24:48 WARN    Permission denied
2007/11/21 09:24:48 LOG     Processed 0 files in 0 directories.
```

In this case simply add the “usermasq” to the options on the compress command and the results of the next run should work as expected.

```
2007/11/21 09:30:23 LOG     Compress: /xfertmp/simon/file1 ->
/xfertmp/simon/file1.gz [18.20%].
2007/11/21 09:30:23 LOG     Compress: /xfertmp/simon/file2 ->
/xfertmp/simon/file2.gz [18.16%].
2007/11/21 09:30:23 LOG     Processed 2 files in 1 directory.
```

Please remember that when using “usermasq” the compress functionality runs as the owner of the “source” file. If a destination file exists and **is not** owned by the user same then the compression functionality will fail.

4.4 Tail Rules

This is one of the most useful rules since it manipulates log files to change their contents but leaving the “most recent” entries in them. Tail works by estimating the count of lines in the file and removing any to take the file down to a certain size. However it removes the lines at the start of the file, not at the end, and so logically the “most recent” entries are those left in the file.

Of course since it estimates the number of lines in the file it will only work with text files - though it transparently works with compressed [text] files that are compressed with the compression routines registered with Skulker.

A sample “tail” rule might appear as:

```
<rule
  n="1010"
  type="tail arguments"
  interval="1D"
  match_type="file"
  match_pattern="/var/adm/syslog/.*\.log"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

The list of supported arguments for this rule type are:

keep	Indicates the number of lines to keep in the file. If the file currently has no more than this number no changes will occur to it. If this is not specified the main configuration file value for keep will instead be used.
maxsize	This can be used instead of the “keep” argument just described. It indicates the maximum amount of data to keep in the original file as a maximum size rather than number of lines.
unsafe	Over rides the protected directory settings if possible for the files to attempt to match.
handleopen	This will attempt to process the file in such a way that if it is open for writing new log entries will correctly write to the end of file once Skulker finishes with it. This requires Skulker to perform more work, but is safer.
olderto	Indicates the name of the file to write any discarded data to. See the paragraph below for details.
usermasq	Run the “tail” function as the user that owns each source file that is queried or modified.
target	Alias for the above option

The `olderto` setting has a default typically of “NULL” in the Skulker configuration file - meaning that older lines from the file processed are simply discarded. However it is also possible to specify the name of a file - and in this case the data will be appended to the end of the file.

Notice that the `olderto` value can also be a compressed file - and in this case Skulker will intelligently uncompress the target, append any new data and then recompress the file automatically.

The name can also make use of the following macros - useful if the rule might process many files:

%F	The full path and filename of the currently matched file
%d	The directory portion of the currently matched file
%f	The filename portion of the currently matched file

4.4.1 Use of the “maxsize” Option

Some people find it more useful to indicate the size of the file they wish to maintain rather than a number of lines [which can result in differing file sizes depending on the lengths of the lines in question]. In such cases the “maxsize” option is recommended. It takes the following form:

```
maxsize="NNN[kmg]"
```

The optional letter on the value indicates that the value should be taken in as:

- k - Kilobytes
- m - Megabytes
- g - Gigabytes

Without a letter it means that the value is in bytes.

It should be remembered that “maxsize” value is the maximum size of the file to keep - it is likely to be slightly less since it is based on keeping text lines intact rather than working on a byte-by-byte basis.

4.4.2 Use of the “usermasq” Option

This option can be useful when attempting to use the “tail” function on files which are on NFS shares. In such circumstances it is possible that even running as “root” problems such as the following occur:

```
2007/11/21 11:13:28 WARN Unable to open file /xfertmp/simon/file1:
2007/11/21 11:13:28 WARN Unable to open file /xfertmp/simon/file1:
2007/11/21 11:13:28 WARN Permission denied
2007/11/21 11:13:28 WARN Unable to open file /xfertmp/simon/file2:
2007/11/21 11:13:28 WARN Unable to open file /xfertmp/simon/file2:
2007/11/21 11:13:28 WARN Permission denied
```

Once the “usermasq” has turned on the next attempt at tailing the files should work as expected:

```
2007/11/21 11:16:28 LOG /xfertmp/simon/file1 [nbkbb5h:users] - 373
lines removed.
2007/11/21 11:16:28 LOG /xfertmp/simon/file2 [nbkbb5h:users] - 371
lines removed.
```

It should be noted that if the “olderto” option is used in this case the target directory must be writable by the user of the source file. Of course if that file already exists it also needs to be writable by the source file owner too.

4.5 Rotate Rules

The rotate function takes a file and then renames it to include a numeric extension. If a file of that name already exists the extension value it has is increased. Once a file will be given an extension larger than that configured in the rule it will simply be deleted rather than renamed.

For example if the intention is to keep 3 copies, a file might be remained in following stages:

Original: filename
Run 1: filename.1
Run 2: filename.2
Run 3: filename.3
Run 4: *File removed*

Of course if “filename” was recreated after run 1 then it would also start to be renamed as well and eventually be removed. Log rotation is a common feature offered by many applications or log managers. A typical rule of this type follows the familiar syntax:

```
<rule
  n="1000"
  type="rotate arguments"
  interval="1D"
  match_type="file"
  match_pattern="/opt/hpws/tomcat/logs/catalina\.out$"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

This rule type supports the following arguments:

handleopen	Treat the log file as potentially being open by a file as a log file and handle accordingly. In this case the filename in question will exist following the action, but unlike “keep” more care is taken to ensure the file writing to the processed file is not affected by this operation.
unsafe	Over rides the protected directory settings if possible for the files to attempt to match.
copies=N	Keep this many copies of the rotated file. Typical value is “5”.
keep=0 1	Whether the original file should cease to exist after the rotate or exist as an empty file. If not specified the default in the configuration file is typically “0” - indicating the original file should not be used.
format=Format	The format of the filename of the rotated file. If not specified the value from the Skulker configuration file is used - which is typically “%d/%f.%N” - that is the original filename with a decimal numeric extension.

4.6 Deletedir Rules

All of the previous rule types allow files to be deleted. As the name of this rule type suggests this instead works on removal of *empty* directories. A typical rule might look like:

```
<rule
  n="1010"
  type="deletedir arguments"
  match_type="directory"
  match_pattern="/var/adm/crash/.*"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

When a directory passed to the routine is seen as empty (it contains no files), then it will be removed. The utility checks the specified directory recursively and runs from the bottom upwards. Thus if a specified directory has a large number of directories and sub-directories it will all be removed if no actual files are found anywhere in the directory structure.

It is possible to remove all empty sub-directories by matching all directory names and passing them to this function. In this case the top-level directory should be specified recursively as part of the rule, for example:

```
match_pattern="/var/adm/crash/.../.*"
```

At present only the following arguments are supported:

reportonly	Do not make any changes – simply report on the changes that would be made.
emailto	Send a report of any activity [either in “reportonly” or normal modes] to the specified list of users.

4.7 Move Rules

This implements a very straightforward facility – it will simply take the matching files and move them to a different directory. This works across file systems without issue [performing a copy/delete when necessary].

The typical format of such a rule might be:

```
<rule
  n="2000"
  type="move arguments"
  match_pattern="/var/adm/cron/log"
  commands="[1] %INST/utls/stopcron"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

For any file moved the owner, group, permissions and modification times from the original [even if copied] are kept intact. At present the tool supports the following arguments:

reportonly	Do not make any changes – simply report on the changes that would be made.
emailto	Send a report of any activity [either in “reportonly” or normal modes] to the specified list of users.
destination	The name of the directory to move the files too.
crdirs	If set to true will create any missing directories in the path for the destination that are currently missing.
unsafe	Attempt to ignore any protected directory settings when possible [if when override=”yes” for directories being scanned].

4.8 Command Rules

This is a very straightforward rule - it simply runs the specified series of commands. This is useful if Skulker should shut down a process before dealing with a log file and then starting it up again - useful for “cron” and “at” daemons for example.

The typical format of such a rule might be:

```
<rule
    n="2000"
    type="command_arguments"
    match_pattern="/var/adm/cron/log"
    commands="[1] %INST/Utils/stopcron"
    match_pattern_ignore="expression"
    match_pattern_truncate="expression"
    match_pattern_limit="count"
    match_pattern_limit_by="age|name|reverse_age|reverse_name"
    max_files_count="N"
    match_pattern_opposites="no"
    match_by="expression"
/>
```

It is often useful to ignore certain rules if this specified commands return with an unsuccessful return code - see the “Ignoring Rules on Failure Conditions” later for more information.

Notice that although it does not actually directly manipulate any file a “match_pattern” is expected and the commands called will subject macro evaluation as defined later. Hence if 10 files are matched the rules will be executed 10 times.

The commands themselves are expected to exist on separate lines, prefixed by a number, for example:

```
commands="[1] /usr/bin/true"
```

or

```
commands="    [1] echo Stopping daemons
             [2] /application/daemon stop"
```

The numbers are simply used as labels - they are executed in the order given, not any kind of numerical sorted order.

4.9 Mailfile Rules

As the name suggest this is a specialist rule type that can handle mailbox format files. Please note that the current version *does not handle maildir folders*. Maildir folders might be managed by an additional rule or modifications to this rule, in future releases.

A typical rule file entry for this rule might be:

```
<rule
  n="1010"
  type="mailfile arguments"
  match_pattern="/var/mail/*.*"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

The arguments are used to define what actions to take on the file. Please note that an effort is made to ensure that the file is indeed the correct format before it is processed.

The available arguments, which are defaulted from the main configuration file if not present are;

keep	Determine the maximum number of messages that should exist in the file.
maxsize	Determine the maximum size of the mailbox. The format of the number is a decimal number followed by any of “b”, “k”, “m”, “g” or “t” suffixes to indicate the maximum in bytes, kilobytes, megabytes, gigabytes or terabytes [that is a lot of Spam!]
unsafe	Over rides the protected directory settings if possible for the files to attempt to match.

Notice that if both arguments are present older messages are deleted until both conditions become true.

4.10 Organise Rules

Originally it was thought that this rule would probably not be that useful; however the opposite was found to be true. When `organise` is called it will use the modification time of any matched file to ascertain a subdirectory based on the match filename to move it to. The format of this directory defaults to `logs-%year%mon%day` but can be changed via arguments or even changing the options in the main Skulker configuration file.

This rule type is particularly useful for environments where large number of log files are generated in a directory on a frequent basis. Using this tool to migrate files into sub-directories makes directory searches more efficient, and allows the log files to be managed to more easily by other UNIX tools.

A typical `organise` entry might be:

```
<rule
  n="20"
  type="organise arguments"
  match_pattern="/logdir/.*\.log"
  match_pattern_ignore="expression"
  match_pattern_truncate="expression"
  match_pattern_limit="count"
  match_pattern_limit_by="age|name|reverse_age|reverse_name"
  max_files_count="N"
  match_pattern_opposites="no"
  match_by="expression"
/>
```

In the above example it would move any `.log` files in the specified directory to the directory defined by the supplied argument `format`, or the `format` defined for `organise` rules in the Skulker configuration file.

At present `organise` only supports one optional argument:

<code>format</code>	Defines the format of the directories to move the file into. If not specified the default from the Skulker configuration file is used, which is typically set to <code>“%d/%year-%mon-%day”</code> .
---------------------	--

Typically `organise` rules are followed by other rules that work on the directory the files were moved to - to compress the files, for example.

4.11 Archive Rules

It should be noted that this is not actually a way of saving disk space - as with the “organise” rule the aim here is to manage files rather than reduce disk space. For this rule type the files that match will be copied into an archive file.

A typically rule might appear similar to:

```
<rule
    n="20"
    type="archive archive=%d/%H.archive"
    match_pattern="directory/pattern"
    match_pattern_ignore="expression"
    match_pattern_truncate="expression"
    match_pattern_limit="count"
    match_pattern_limit_by="age|name|reverse_age|reverse_name"
    max_files_count="N"
    match_pattern_opposites="no"
    match_by="expression"
/>
```

The “archive” argument is necessary - it gives the name of the archive to move any matching files into. It can be in another directory, and the name is subject to the usual macro expansion.

Once the file has been copied into the archive it is removed from the file system. The following optional arguments are supported:

handleopen	Both are synonymous - rather than deleting the file it is truncated down to zero bytes. This is typically used if the file is likely to be open at the time it needs to be moved to the archive.
truncate	
type	Indicates the type of the archive to create. At present only one is available [and is of course the default!] - “ar”.
unsafe	Attempts to override patterns matching protected directories if possible.

4.11.1 Limitations of “ar” archive format

Ar is a very basic archive which has the following limitations:

No pathname information is kept - if two files of the same name in different directories are added to the archive only the last one will exist in it!

The function always works in “replace” mode - adding the same file twice to the archive will overwrite an existing copy of that file in the archive. [Hint: If you wish to keep multiple versions use an “organise” rule first to rename the file].

No compression occurs - hence no disk space is saved. This can be circumvented by matching the files using a compress function first, of course.

4.12 Saswork Rules

4.13 SPDSwork Rules

5 Conditional Execution

As the complexity of any series of rules increases it is sometimes useful to be able to automatically run or exclude a series of rules. At present, [apart from rule frequency, or rule specification on the command line], there are two ways to perform conditional execution;

- Only include a rules file based on a specified condition
- Ignore specified rules if a currently executing rule fails

5.1 Conditional Includes

There are two different ways in which excludes can be considered conditional, firstly through the use of standard macro processing. For example consider;

```
<include file="hostonly-%H.xml"/>
```

Based on the macros (described below), that name of the file to include will be “hostonly-*hostname.xml*”. If such a file does not exist then the include statement will be silently ignored.

The alternative approach is to include an expression that must be true for a rule file to be included. For example:

```
<include iftrue="UNAME eq 'HP-UX'" file="hpux.xml"/>
```

The above would only attempt to read the rules files “hpux.xml” if the operating system name (UNAME) was actually set to HP-UX. Details on expression evaluation can be found in a section below.

5.2 Ignoring Rules on Failure Conditions

By default when running against a specified rules file, Skulker will typically run all rules where the specified minimum time interval for a rule has been exceeded. This is even the case when those rules are explicitly specified via command line arguments.

There are occasions when the user might wish to ignore certain rules if other associated rules fail when attempting to be processed. This is often typical if the rule type is a command which must stop a process prior to the log files it generates being managed. For example;

```
<rule
  n="2000"
  type="command onfail=abort"
  match_pattern="/var/adm/cron/log"
  commands="[1] %INST/utis/stopcron"
  ignore_on_fail="hpux-crontab:2001,hpux-crontab:2002,hpux-
crontab:2003"
/>
```

In the above case the rules 2001, 2002 and 2003 are ignored if the command files. Please note currently there is no ability to “shortcut” the names of the ignored rules to something like “2001-2003” [that may come later].

6 Additional Rule File Syntax

6.1 File Name Mangling Macros

If a filename as a “target” needs to be specified - such as defined for “rotate” and “tail”, the name is subject to macro expansion. At present the following macros are available:

Macro	Purpose	Example
%f	The filename being manipulated without the pathname component.	filename
%d	Only the pathname component of the current file.	/a/b
%d[n]	Takes the first ‘n’ parts of the pathname of the current file. For example, if a file was “/a/b/c/d/file”, then “%d[2]” would be “/a/b”.	/a/b
%d[-n]	Takes the first ‘n’ parts of the pathname of the current file. For example, if a file was “/a/b/c/d/file”, then “%d[-2]” would be “/c/d”.	/c/d
%ext	The last extension of the current file including the preceding period - for example “.gz” for a file “file.tar.gz”. If the file does not have an extension it will return an empty string.	.Z
%barename	The file name with all extensions removed.	file
%F	The complete filename of the current file - including the path.	a/b/filename
%T	The current time in the format “HHMMSS”	124423
%D	The current date in the format “YYYYMMDD”	20060516
%N	The current file rotation number [only useful for rotate files]	4
%year	The year part of the modification time of the filename being processed, (including century).	2005
%mon	The month part of the modification time of the file being processed, in the range of “01” to “12”.	08
%day	The day part of the modification time of the file being processed, in the range of “01” to “31”.	02
%hour	The hour part of the modification time of the file being processed, in the range of “00” to “23”.	22
%min	The minute part of the modification time of the file being processed, in the range of “00” to “59”.	05
%sec	The second part of the modification time of the file being processed, in the range of “00” to “59”.	42
%H	The hostname of the current machine.	bluenut
%U	The current Uname of the machine	HP-UX
%INST	The installation directory of the currently running Skulker binary.	/opt/skulker2
%V:name	The value of the named variable “name”. The variable must be predefined in the Skulker configuration file, or specified on the command line using the “--var name=value” option.	value

The values above can be used any number of times in the same expression [if for any reason that is needed].

6.1.1 Examples of Using Partial Directory Matching

The “%d[n]” and “%d[-n]” constructs can actually be very useful. For example consider the following directories:

```
/usr/opensv/volmgr/debug  
/usr/opensv/volmgr/debug/ltid  
/usr/opensv/volmgr/debug/daemon  
/usr/opensv/volmgr/debug/reqlib  
/usr/opensv/volmgr/debug/acssi
```

If you wished to remove the “debug” logs from under the various “debug” directories it might be possible to write different rules for each sub-directory, whereas a better approach might be to use a rule such as:

```
<rule  
  n="2710"  
  interval="0D"  
  type="move unsafe crdirs=true destination=/savedir/volmgr/%d[-2]/%f"  
  match_type="file"  
  match_by="MTIME OLDER_THAN 2D"  
  match_pattern="/usr/opensv/volmgr/.../log\.\d+"  
>
```

Notice that the “destination” setting for the move action contains “%d[-2]”, then consider the following file:

```
/usr/opensv/volmgr/debug/daemon/log.043009
```

When this is applied to the destination it will be:

```
/savedir/volmgr/debug/daemon/log.043009
```

The value above in bold equates to “%d[-2]” for this particular example. This allows the moves to keep to the existing directory structure, creating any missing directories when required.

6.2 Interval Definitions

The interval defines the minimum amount of time that must pass before a specified rule can be run again. In the simplest form this might be;

```
5760m
96h
4d
```

Each of the above is equivalent - 5760 minutes, 96 hours or 4 days. If a rule can be run every time Skulker is called simply specify the interval as “0h”, “0m” or “0d”.

Sometimes it is useful to ensure that a rule is only run on a specified day of the week, and in this case the following are equivalent;

```
Sun
Sunday
```

If a rule can be scheduled to run on either a Saturday or a Monday the following are all equivalent:

```
Sat, Mon
Saturday, Monday
```

It is also possible to run a rule on a specified day of the month, for example for following are equivalent;

```
4h:28
4h:28DOM
```

These indicate that the rule can only be run on the 28th day of the month. Since the “28” is prefixed by “4h” it means that it could be potentially run every 4 hours on the 28th day of the month. To run at most once a day on days 5,10,15 and 20th of the month the following might be used:

```
1d:5,10,15,20
1d:5DOM,10DOM,15DOM,20DOM
```

Finally note that the interval may include spaces and if so each part must be true for the rule to be considered, for example:

```
4h:1,2,3,4,5,6,7 Monday
```

This would only allow the rule to be run on the first Monday of the month, up to 6 times due to a frequency of 4 hours!

6.3 Pattern Matching Functionality

The pattern matching facility is powerful - since native Perl patterns are allowed. However to cater for the fact that many administrators tend to be more comfortable with Shell-based pattern matching the following basic shell constructs are supported and converted when necessary to Perl patterns;

<code>fred.*</code>	This gets converted to “ <code>^fred\.*\$</code> ” - that is it will match all files starting with “fred.” as the administrator might expect.
<code>*.log</code>	This gets converted to “ <code>^.*.log\$</code> ” - this is <i>almost</i> equivalent - it will match all files ending in “.log” - but it will additionally match any file ending in “blog”, “flog” and “slog” for example.
If correct Perl equivalent would be “ <code>.*\.log\$</code> ”.	

In almost all cases using native Perl pattern matching provides the administrator significantly more power. Of course regular expression matching in Perl literally fills whole books, but here are some common examples that might be useful.

<code>^(fred mike pete)\.log\$</code>	Matches the files “fred.log”, “mike.log” or “pete.log”.
<code>core</code>	Matches just the file called “core”.
<code>^core.*</code>	Matches any file name beginning with “core”.
<code>^.*\.(gz bz2 Z)\$</code>	Matches any filename which ends in a known compression type “Z”, “.gz” or “.bz2”.
<code>^[a-z]+</code>	Matches any file that begins with any character “a” through “z” (in lower case).

All the above pattern matching assumes that the filename to pattern match is preceded by a directory to scan, for example:

```
/tmp/.*\.log$
```

Of course it is sometimes useful to look for the specified pattern in a series of sub-directories. This functionality is provided by “recursive directory pattern matching”. This feature is activated by separating the top level directory and the pattern to match with any of the following;

<code>/dir/.../pattern</code>	Match the specified pattern in the specified directory or any subdirectory.
<code>/dir/...[n]/pattern</code>	Match the specified pattern in this directory and any directory up to “n” levels down the directory tree.
<code>/dir/...[,n]/pattern</code>	Same as the above recursive expression.
<code>/dir/...[m,n]/pattern</code>	Matches the specified pattern in directories “m” to “n” levels down the directory structure.

<code>/dir/.../[n,]/pattern</code>	Matches the specified pattern, starting at “n” sub-directories below the search directory.
------------------------------------	--

In the above cases “m” and “n” are set to “0” for the directory itself and “1” for the first level of sub-directories.

This for example to search “/tmp” and all sub-directories for files that are Gzip compressed the following might be used:

<code>/tmp/.../**.gz\$</code>

Further remember that the `match_pattern_truncate` attribute can be used to ignore sub-directories [and any directories beneath them from the recursive matching, and `match_pattern_ignore` further filter out unwanted matches.

6.4 File Criterion Matching Functionality

Apart from matching the pattern for filenames it is often important that a rule is only applied to a file matching a specified criterion - such as not being modified for a certain number of days, owned by a certain user, or larger than a certain size for example. This is where the “match_by” attribute is used in rule specification.

6.4.1 File Attribute Macros

The value given is an expression that is used against each and every file and must be true for the file to be processed by the rule in question. A key part of the expression are a series of values that refer to attributes of the current file being processed. At present the following values are available;

SIZE	The size of the current file (in bytes).
MTIME	The UNIX modification time of the file.
TIME	An alias for “MTIME”.
ATIME	The UNIX time when the file was last accessed.
CTIME	The UNIX time when the inode of the file was last changed.
MAGIC	The type of the file based on the file contents. More information on use of this feature is described in section 8, on page 47.
OWNER	The user name (not ID) of the owner of the file. If this user is not known to the system the value is “?”.
GROUP	The group name (not ID) for the group of the file. If this group is not known to the system the value is “?”.
TYPE	The type of file system object that the current “file” is - will be set to any of the following: f file d directory c Character device b Block device s UNIX domain socket l Symbolic Link p Named pipe

6.4.2 Directory Attribute Macros

There are currently four other values which can be used in expressions which describe the status of the directory that is currently being monitored. These are:

DIR_FILECOUNT	This is the number of files in the directory which contains the current file being processed.
DIR_FILECOUNT_R	This is the same as above, but includes a count of files from all sub-directories as well.
DIR_SIZE	This contains the total size of all the files in the current directory being processed. Value is in bytes.
DIR_SIZE_R	As above, but the size of all files from the directory and any sub-directories of the directory of the current file.

It should be noted that the values for the above are calculated once for each directory/rule combination. Hence if a pattern matches 10 files in a directory the value used will be calculated for the first file and will not change.

It should also be noted that the DIR_SIZE and DIR_SIZE_R do not simply sum of the total of the file sizes in question. Instead the utility will attempt to work out the disk allocated used. For example a 100 byte file will be taken as using 8Kb if the file system 'preferred IO size' is 8Kb. On the same file system a 9Kb file will assume to have taken up 16Kb.

6.4.3 Relational Operators

The following relational operators are available for comparing the above attributes to values [all standard Perl are available - below is just a common sub-set]:

>	Numerical greater than
gt	Text comparison "greater than"
<	Numeric less than
lt	Text comparison "less than"
!=	Numeric not equal to
ne	Text comparison "not equal to".
OLDER_THAN	Used for comparison of file create/access or modification times.
NEWER_THAN	Used for comparison of file create/access or modification times.

The following logical operators are available to link multiple comparisons if desired.

Operator	Purpose
&&	Logical "and"
AND	Same as the above.
	Logical "or"
OR	Same as the above.
!	Logical "not"
NOT	Same as the above.
IS	String comparison check [true if two values are the same]
IS_NOT	String comparison check [true if two values are not the same]

6.4.4 Functions

Only a very small number of functions are currently implemented. These typically act on the current file attempting to be processed like the macro variables described earlier.

Function	Purpose
<code>contains(string1, string2)</code>	Returns true if “string2” occurs once or more as a literal string in “string1”.
<code>date_from_name(format, order)</code>	This takes a format, and returns substrings that are meant to represent a date that are part of the file name. See the examples below this table.

6.4.4.1 Examples of “date_from_name” Function

This function can be useful in very particular circumstances. It allows a portion of a filename to be converted to a date - which is then typically compared to the current date using the “OLDER_THAN” or “NEWER_THAN” logical operators.

For example consider the following series of filenames:

<code>-rw-rw----</code>	<code>1</code>	<code>user1</code>	<code>users</code>	<code>0</code>	<code>Jan</code>	<code>8</code>	<code>13:50</code>	<code>/tmp/test_2007_03.xml</code>
<code>-rw-rw----</code>	<code>1</code>	<code>user1</code>	<code>users</code>	<code>0</code>	<code>Jan</code>	<code>8</code>	<code>14:40</code>	<code>/tmp/test_2008_01.xml</code>
<code>-rw-rw----</code>	<code>1</code>	<code>user1</code>	<code>users</code>	<code>0</code>	<code>Jan</code>	<code>8</code>	<code>14:40</code>	<code>/tmp/test_2008_02.xml</code>
<code>-rw-rw----</code>	<code>1</code>	<code>user1</code>	<code>users</code>	<code>0</code>	<code>Jan</code>	<code>8</code>	<code>13:50</code>	<code>/tmp/test_2008_03.xml</code>
<code>-rw-rw----</code>	<code>1</code>	<code>user1</code>	<code>users</code>	<code>0</code>	<code>Jan</code>	<code>8</code>	<code>13:49</code>	<code>/tmp/test_2008_04.xml</code>

Notice that all the files appear to have the same modification date - but logically contain “YYYY_MM” [year and month] to indicate when the data is considered relevant for. Hence to extract this part of a date and see if older than a year a rule such as the following could be used:

```
<rule
  n="40"
  match_pattern="/tmp/test_\d{4}_\d{2}\.xml"
  match_by="date_from_name('.*_(\d{4})_(\d{2})\..*', 'year,month') OLDER_THAN 1Y"
/>
```

Notice to make use of this feature it is necessary to use regular expressions - in the above example “\d{4}” matches 4 digits.

The “order” is a comma-separated string of any of the following values [no spaces should be included]:

- year - indicates the year - such as 2008. If matching against a two-digit representation of the year (“such as 08”), then 2008 is added to it - for example “01” becomes “2001”.
- month - The month of the year - range of 1 to 12. Optionally can include a leading zero.
- day - The day of the month - range of 1 to 31. Optionally can include a leading zero.

6.4.5 Unimplemented File Attribute Macros

The following “helper functions” will become available at some point in the future but are not currently implemented:

FILE_CLOSED	Not yet implemented
FILE_OPEN	Not yet implemented.

6.4.6 Literal Values

When specifying constants values to compare against the following are available;

'string'	Any string value.
"string"	Same as the above.
2323	Any number

When attempting to match the size of files the number specified is the number of bytes, but various suffix strings are available to help make the rules easier to write and then later read. The following are examples of numeric suffixes supported:

2b 1byte 22bytes	The size to compare against in bytes. The strings “byte” and “bytes” can be upload/lower case, though the “2b” example must be in lower case.
1k 42Kb 1Kbyte 1Kilobyte 3Kbytes 4Kilobytes	Specified the size in kilobytes. Here the number before the suffix is multiplied by 1024 to work out the size for comparison.
2m 1Mb 1Mbyte 1megabyte 32Mbytes 13Megabytes	Specified the size in megabytes. Here the number before the suffix is multiplied by 1,048,576 to work out the size for comparison.
33g 1Gb 1Gbyte 1Gigabyte 232Gbytes 11Gigabytes	Specified the size in gigabytes. Here the number before the suffix is multiplied by 1,073,741,824 to work out the size for comparison.
12t 1Tb 1Tbyte 1Terabyte 3Tbytes 2Terabytes	Specified the size in terabytes. Here the number before the suffix is multiplied by 1,099,511,627,776 to work out the size for comparison.

When looking at the age of files various value to indicate a time **relative to the current time** are available, and take the form of suffixes similar to the size values just shown. For example “1s” means 1 second before the current time, so these relative times are converted to UNIX times for comparison with the ages of files. The following suffix strings are supported:

2S 1Second 44Seconds	<p>Indicates the current UNIX time with this number subtracted from it. The suffix indicates multipliers of 1 [since just a number of seconds].</p> <p>The longer “second” or “seconds” suffixes are used they can be upper/lower case - but must be upper case is just using “S”.</p> <p>, 60, 360 and 86400 (for seconds, minutes, hours or days).</p> <p>The “W” takes away the specified number of months from the current time/date - hence the amount of time older it is than the current time/date depends on which month it is currently. If taking away a month day leaves the day of the month beyond the previous month the last day in the previous month is taken. For example on 31st March, the value “1W” means 28th February.</p> <p>Y is a number of years.</p>
2M 1Minute 33Minutes	<p>Indicates the current UNIX time with this number of minutes subtracted from it.</p> <p>The longer “minute” or “minutes” suffixes are used they can be upper/lower case - but must be upper case is just using “M”.</p>
1H 1Hour 13Hours	<p>Indicates the current UNIX time with this number of minutes subtracted from it. The case must be as described in the previous options.</p>
1D 1Day 3Days	<p>Indicates the current UNIX time with this number of days subtracted from it. The case must be as described in the previous options.</p>
1W 1Month 15Months	<p>Notice that we can’t use “1M” since this is used for minutes! This differs from previous settings since the amount of time it subtracts depends on the month. For example, if the date is 30th July and it will reduce the date to 30th June, [at the same time - this is remove 31 days].</p> <p>However if the preceding month does not have that date the last day of that month is used. For example on 31st July, “1Month” will actually mean 30th June.</p>
1Y 1Year 3Years	<p>Reduces the date by the specified number of years [using the same time as currently]. If the new year is not a leap year but the current date is 29th February then the date will be changed to 28th February on that year.</p>

6.4.7 Example Criteria

To make this meaningful consider the following examples.

```
MTIME OLDER_THAN 10D
```

Specifies that the file should only be processed if the modification time of the file is more than 10 days ago.

```
OWNER eq "root"
```

Only process files that are owned by root.

```
GROUP ne "sys"
```

Only process files that do *not* have a group of "sys".

```
MTIME OLDER_THAN 2D AND SIZE > 100k
```

Only process files that have not been modified for at least 2 days and are larger than 100 Kilobytes in size.

```
OWNER eq "bin" OR OWNER eq "root"
```

Process files if they are owned by "bin" or "root".

```
(OWNER eq "root" OR OWNER eq "bin") AND MTIME OLDER_THAN 6H
```

Process files if they are owned by "root" or "bin" - but only if the modification time is more than 6 hours old.

```
date_from_name('.*_(\d{4})_(\d{2})\..*', 'year,month') OLDER_THAN 1Y
```

The above looks for a pattern "YYYY_MM." from the files matched and checks to see if that is older than 1 year. The "date_from_name" is a powerful function - just remember that if it does not match the the date it generates is 1st January, 2030 - ensuring no "OLDER_THAN" rules match files unexpectedly.

7 Email Integration

Some of the Skulker rule types now support the concept of email reporting - both either reporting only [and making no file system changes], or by making the changes as normal, but also sending out an email once the rule has completed.

The email functionality is made available whenever a rule action [as described previously] supports a “mailto” argument. The emails are generated for each rule - so if 10 rules use the “mailto” argument, then 10 different emails will be generated - even if the recipients are the same.

The “mailto” argument is given a comma-separated list of recipients to send the information to - the content of the message and the subject is not driven by any arguments. In fact the title for the message is set and will always appear in the following format:

```
Skulker v2 Rule rulesfile:rulenumber Results
```

Since it is set the title can be very easily used to filter/organise mail as necessary. The majority of the contents of the file [which is simple text] are obviously defined by the rule type itself. However it is possible to provide some text to precede the rule output.

7.1 Generating Text in Emails

To provide some text in an email [to describe to the recipients what the rule is actually doing], files need to be generated in the “email prefix directory”. This is defined by the “emailprefixdir” variable in the configuration file, which typically is defined as:

```
emailprefixdir="%INST/email"
```

Thus a directory may need to be created, for example:

```
# mkdir -p /opt/customer/skulker2/email
```

In this directory each rule that wishes to prefix the rule output with some text should create a separate file. The filename format is as follows:

```
rulesfile.rulenum.prefix
```

For example for the rulesfile “solaris-wtmp.xml” rule number 1020, the filename to use would be:

```
solaris-wtmp.1020.prefix
```

Note: The “.xml” extension is not included in the file name.

8 Use of File Type Detection [“Magic”]

8.1 What is “Magic”?

One useful feature that is provided as part of UNIX installations is a tool called “file”. It uses a file containing a series of rules and attempts to ascertain the type of a file based on applied those rules to the contents of the file until a match is found.

The name of the rules file is typically called “magic” - since it contains rules to recognise the “magic numbers” used by files to ensure applications can recognise them correctly. The number of rules [and hence the number of different file types that can be recognised] varies by Operating System, and for Linux, distribution.

Skulker supplies a copy of a “magic” file used on Ubuntu - since this recognises a larger number of files than most other UNIX variants - though use of the one made available as part of the standard Operating System installation is possible via changes to the Skulker configuration file.

8.2 Limitations of “Magic” Handling

The approach taken by Skulker in attempting to make use of magic file types is to include a module for directly scanning the rules file and the file contents. This approach was taken for performance reasons - running an external command to check for magic information would be extremely inefficient - especially if many thousands of files need to be tested.

However, although the rules file might look simple, they can be quite complex and thus generating a Perl module to efficiently handle those rules is a challenge. As it stands only a small [but useful] sub-section of rules can be understood. Later versions will improve on this to ensure a larger range of file types can be successfully detected.

8.3 Testing whether “Magic” works on Necessary Files

Because Skulker uses a custom “magic” file, and only recognises a small subset of files a small utility is provided to allow testing of files to see if they are recognised. Assuming that Skulker is installed in “/opt/skulker2” then the command that can be run is:

```
/opt/skulker2/bin/skulker_magic_test
```

If installed in a vendor directory the path instead will be:

```
/opt/vendor/skulker2/bin/skulker_magic_test
```

It is past the name of the “magic” file to use and file types that should be attempted to be recognised. For example:

```
/opt/skulker2/bin/skulker_magic_test \  
-m /opt/skulker2/cfg/extended_magic xxx.Z a  
xxx.Z: compress'd data  
a: POSIX tar archive
```

If a file type is not recognised the output will appear as:

```
xxx: UNKNOWN
```

8.4 Using “Magic” on Skulker Rules

Using the information gained from the output of the test utility it is then possible to make use of the magic information to either recognise file types, or more commonly, act as a “backup” - for example consider the following rule:

```
<rule
  n="20"
  type="delete"
  match_pattern="/directory/path/*"
  match_by="MAGIC eq 'PA-RISC2.1'"
/>
```

The above would compare the magic value of any compared file with “PA-RISC2.1” and only if they match exactly will the file be considered for deletion. However as stated usually it is using in conjunction with other functionality:

```
<rule
  n="20"
  type="delete"
  match_pattern="/a/directory/*.tar$"
  match_by="MAGIC eq 'POSIX tar archive'"
/>
```

The above ensures rather than assuming that all “.tar” files actually tar archives the contents also indicate the file type is correct.

Another thing to consider is that output can differ depending on how programs are created, for example consider the following output:

```
/opt/skulker2/bin/skulker_magic_test \
-m /opt/skulker2/cfg/extended_magic /tmp/sss /tmp/xxx
/tmp/sss: GNU tar archive
/tmp/xxx: POSIX tar archive
```

The usual way of dealing with these two types might be to have a rule such as:

```
<rule
  n="20"
  type="delete"
  match_pattern="/a/directory/*.tar$"
  match_by="MAGIC eq 'POSIX tar archive' OR MAGIC eq 'GNU tar archive'"
/>
```

However there is also a useful function that has been added to allow sub-string patterns to be matched instead. Hence a better way might be to look for all magic types that contain the string “tar archive”:

```
<rule
  n="20"
  type="delete"
  match_pattern="/a/directory/*.tar$"
  match_by="contains(MAGIC,'tar archive')"
/>
```

The “contains” function checks to see if the string from the first argument contains the string given in the second argument.

Finally remember that this function can be used with other logic that the “match_by” element can use, for example to negative the check above:

```
<rule
  n="20"
  type="delete"
  match_pattern="/tmp/cp"
  match_by="NOT contains(MAGIC,'tar archive')"
```

9 Writing a Generator

9.1 Why Write a Generator?

A generator provides a mechanism for matching files that simply directory/pattern matching can not hope to perform. Typically a generator might be used to match files that describe a particular characteristics of the operating environment or are particular to an application.

For example the default generators allow the administrator to match files older than the last machine reboot.

9.2 What a Generator should include?

A generator is a Perl script that must include the following routines:

Subroutine	Purpose
<code>new (LOGGER=>x, STATTER=>y, COMPRESSION=>z)</code>	Create a object and return it. The arguments specified are just the Skulker key objects. Unlikely you will need them, though LOGGER is useful.
<code>check (...)</code>	This takes the parameters for a run of the generator and checks that they are valid. Return values are: (-1,msg) - fatal error (0,undef) - check was successful.
<code>generate (type=>"file directory")</code>	Actually generate a list of matches using the parameters that were last sent to the "check" routine. Return a reference to a list of matching file system object names].

The example in the next section should clarify much of the above.

9.3 An example Generator

For an example generate one will be created called "invalid_user_recursive" and it will take one or more directories and recursively scan them for files that are not owned by valid users on the system [for example when a user account has been deleted].

9.3.1 The “new” routine

The top section defines the name of the package and loads any necessary modules. The name of the package is the name of the generator with “_lister” added to the end:

```
package invalid_user_recursive_lister;

#####
# An example generator to return list of files in one or more      #
# directories that are not owned by valid users.                  #
#####

use File::Find;
use File::Basename;
```

The “new” routine is usually very straightforward. In this instance it scans in the “/etc/passwd” file details since the local “getpruid()” library call does not work will when groups with large number of secondary users exist.

```
sub new {
my $proto=shift;
my $class=ref($proto) || $proto;
my %ARGS=@_;

    my $self={};
    my %users=();
    my ($fd,@F);
    open($fd,"/etc/passwd");
    while(<$fd> {
        @F=split(/:/,$_);
        $users{$F[2]}=$users{$F[0]};
    }
    close($fd);
    $self->{users}=\%users;

    if(exists($args{LOGGER})) {
        $self->{LOGGER}=$args{LOGGER};
    }
    if(exists($args{STATTER})) {
        $self->{STATTER}=$args{STATTER};
    }
    if(exists($args{COMPRESSION})) {
        $self->{COMPRESSION}=$args{COMPRESSION};
    }
    bless($self,$class);
    return($self);
}
```

The handling of the LOGGER, STATTER and COMPRESSION arguments is standard and should be repeated for each generator. Most of the above code is then simply loading in all the user details from “/etc/passwd”.

If your generator wishes to emit log, warning or error messages, then typically the following lines are also present in the generator listing:

```
sub _log ($$$) {
my $self=shift;

    $self->{LOGGER}->logmsg(@_) if exists($self->{LOGGER});
}

sub _warn ($$) {
my $self=shift;

    $self->{LOGGER}->logwarn(@_) if exists($self->{LOGGER});
}

sub _error ($$$) {
my $self=shift;

    $self->{LOGGER}->logerror(@_) if exists($self->{LOGGER});
}
```

9.3.2 The “check” subroutine

The check routine is called for every rule that uses the generator. The purpose of the routine is to check each of the supplied parameters, issuing warnings or errors or log messages as appropriate.

Note: *This routine should record all the “valid” parameters [typically directories] - since they are not passed to the “generate” routine which will be called shortly after the check routine.*

If the generator takes a list of directories the code in the “check” is likely to be very similar to the following:

```
sub check {
my $self=shift;
my @parms=@_;

    if(! @parms) {
        return(-1,"At least one directory must be specified.");
        # -1 return code = abort processing
    }

#####
# At least one directory must exist or a -2 is indicated, #
# which gives a warning and fails the rule - causing the #
# ignore_on_fail processing for the rule to occur. #
#####

my @ok=();
for my $carg (@parms) {
    if(! -e $carg) {
        _warn($self,"Path '$carg' does not exist -
ignoring.");
        next;
    }
    if(! -d $carg) {
        _warn($self,"Path '$carg' is not a directory -
ignoring.");
        next;
    }
    if(! -r $carg) {
        _warn($self,"Path '$carg' is not readable -
ignoring.");
        next;
    }
    push @ok,$carg;
}
if(! @ok) {
    _warn($self,"Generated matched no directories.");
}
$self->{dirs}=@ok;
return(0,undef);
}
```

The return codes for the “check” routine are important - if it returns (-1,msg) then Skulker will abort with an error message of “msg”. Otherwise return (0,undef) to continue - even if warnings are issued.

As can be seen from the code above a list of directories is built up and then kept in “\$self->{dirs}” for processing via the “generate” routine.

9.3.3 The “generate” subroutine

The “generate” routine takes any information saved from the previous call to the “check” routine and returns a list of matches. In this case since it is a “recursive” type of generator we use the “File::Find” module to scan through the list of directories, taking any file (or directory) that matches.

```
sub generate {
my $self=shift;
my %args=@_ ;

    if(! @{$self->{dirs}}) {
        return [];
    }
my @matches=();
my $mtype="file";
if(exists($args{type})) {
    $mtype=$args{type};
}
for $cdire (@{$self->{dirs}}) {
    find( sub {return if($_ eq "." || $_ eq "..");
        if( ($mtype eq "file" && -f $File::Find::name) ||
            ($mtype eq "directory" && -d
$File::Find::name) ) {
                my @S=stat(_);
                if(! exists($self->{users}->{$S[4]})) {
                    push @matches,$File::Find::name;
                } },$cdire);
    }
    return [@matches];
}
```

The return value of the routine is a reference to a list of matches. Notice the first section - if the list of valid directories from the “check” routine is empty an empty list is returned.

9.4 Basic Syntax Check

Once the code has been written is it quite easy to check for a syntax error. Simply go to the directory of the “generators” directory and issue something like:

```
# perl -w -e 'do "invalid_user_recursive.lister"; print "$@\n";'
```

If no error message is printed then the compile-time syntax of the code has passed successfully.

9.5 Changes necessary to pick up Generator

Once the generator has been written the “generators” section of the configuration file must be altered to include the new generator. For example in this case:

```
<generators directory="%INST/generators"
    list="olderthanboot olderthanboot_recursive
        invalid_user_recursive"/>
```