Cfengine Reference Manual

Auto generated, self-healing knowledge
for version 3.0.1a7

cfengine.com

1 Cfengine 3.0.1a7 – Getting started

Cfengine is a suite of programs for integrated autonomic management of either individual or networked computers. It has existed as as software suite since 1993 and is published under the GNU Public License. This document represents cfengine version 3.0.0 of cfengine, which is a radical departure from earlier versions.

Cfengine 3 has been changed to be both a more powerful tool and a much simpler tool. Cfengine 3 is not backwards compatible with the cfengine 2 configuration language, but it interoperates with cfengine 2 so that it is "run-time compatible". This means that you can change over to version 3 slowly, with low risk and at your own speed.

With cfengine 3 you can install, configure and maintain computers using powerful hands-free tools. You can also integrate knowledge management and diagnosis into the processes.

Cfengine differs from most management systems in being

- Open software (GPL).
- Lightweight and generic.
- Non-reliant on a working network to function correctly.
- Capable of making each and every host autonomous

Cfengine 3 consists of a number of components:

`cf-agent`	Active agent
`cf-execd`	Scheduler
`cf-graph`	Graph data extractor
`cf-know`	Knowledge modelling agent
`cf-monitord`	Passive monitoring agent
`cf-promises`	Promise validator
`cf-runagent`	Remote run agent
`cf-serverd`	Server agent
`cf-report`	Self-knowledge extractor

The starred components are new. The daemon formally called `cfenvd` in previous versions of cfengine is now called `cf-monitord`.

Unlike previous versions of cfengine, which had no consistent model for its features, you can recognize *everything* in cfengine 3 from just a few concepts.

Promise	A statement about the state we desire to maintain.
Promise bundles	A collection of promises.

Promise bodies	A part of a promise which details and constrains its nature.
Data types	An interpretation of a scalar value: string, integer or real number.
Variables	An association of the form "LVALUE *represents* RVALUE", where rval may be a scalar value or a list of scalar values.
Functions	Built-in parameterized rvalues.
Classes	Cfengine's boolean classifiers that describe context.

If you have used cfengine before then the most visible part of cfengine 3 will be its new language interface. Although it has been clear for a long time that the organically grown language used in cfengine 1 and 2 developed many problems, it was not immediately clear exactly what would be better. It has taken years of research to simplify the successful features of cfengine to a single overarching model. To understand the new cfengine, it is best to set aside any preconceptions about what cfengine is today. Cfengine 3 is a genuine "next generation" effort, which is will be a springboard into the future of system management.

1.1 A renewed cfengine

Cfengine 3 is a significant rewrite of underlying cfengine technology which preserves the core principles and methodology of cfengine's tried and tested approach. It comes with a new, improved language, with a consistent syntax and powerful pattern expression features that display the intent behind cfengine code more clearly. The main goal in changing the language is to simplify and improve the robustness and functionality without sacrificing the basic freedoms and self-repairing concepts.

Cfengine 3's new language is a direct implementation of a model developed at Oslo University College over the past four years, known colloquially as "Promise Theory". Promises were originally introduced by Mark Burgess as a way to talk about cfengine's model of autonomy and have since become a powerful way of modelling cooperative systems – not just computers, but humans too.

> *"The biggest challenge of implementing cfengine in our organization*
> *was not technical but political – getting everyone to agree.*
> *Promise theory was a big help in understand this."*

Cfengine 3 is a generic implementation of the language of promises that allows all of the aspects of configuration and change management to be unified under a single umbrella.

Why talk about promises instead of simply talking about changes? After all, the trend in business and IT management today is to talk about Change Management (with capital letters), e.g. in the IT Infrastructure Library (ITIL) terminology. This comes from a long history of process management thinking. But we are not really interested in change – we are interested in avoiding it, i.e. being in a state where we don't need to make any changes. In other words we want to be able to promise that the system is correct, verify this and only make changes if our promises are not kept. If you want to think ITIL, think of this as a service that cfengine provides.

To put it another way, cfengine is not really a *change management* system, it is a *maintenance system*. Maintenance is the process of making small changes or corrections to a model. A 'model' is just another word for a template or a specification of how we want the system to work. Cfengine's model is based on the idea of promises, which means that it focuses on what is stable and lasting about a system – not about what is changing.

This is an important philosophical shift. It means we are focused mainly on what is right and not on what is wrong. By saying what "right" is (the ideal state of our system) we are focused on the actual behaviour. If we focus too much on the changes, i.e. the differences between now and the future, we might forget to verify that what we assume is working now in fact works.

Models that talk about change management tend to forget that after every change there is a litany of *incidents* during which it is necessary to repair the system or return it to its intended state. But if we know what we have promised, it is easy to verify whether the promise is kept. This means that it is the *promises* about how the system should be that are most important, not the actual changes that are made in order to keep them.

1.2 Installation

In order to install cfengine, you should first ensure that the following packages are installed.

OpenSSL Open source Secure Sockets Layer for encryption.
URL: `http://www.openssl.org`

BerkeleyDB (version 3.2 or later)
Light-weight flat-file database system.
URL: `http://www.oracle.com/technology/products/berkeley-db/index.html`

In addition...

It is strongly recommended to make the Perl Compatible Regular Expression (PCRE) library available as this is a significant improvement over the buggy POSIX libraries. This documentation assumes the use of PCRE

On Windows machines, you need to install the basic Cygwin DLL from `http://www.cygwin.com` in order to run cfengine.

Unless you have purchased ready-to-run binaries, or are using a package distribution, you will need to compile cfengine. For this you will also need a build environment tools: `gcc`, `flex`, `bison`.

The preferred method of installation is then

```
tar zxf cfengine-x.x.x.tar.gz
cd cfengine-x.x.x
./configure
make
make install
```

This results in binaries being installed in '`/usr/local/sbin`'. Since this is not necessarily a local file system on all hosts, users are encouraged to keep local copies of the binaries on each host, inside the cfengine trusted work directory.

1.3 The work directory

In order to achieve the desired simplifications, it was decided to reserve a private work area for the cfengine tool-set. In cfengine 1.x, the administrator could choose the locations of configuration files, locks, and logging data independently. In cfengine 2.x, this diversity has been simplified to a single directory which defaults to '`/var/cfengine`' (similar to '`/var/cron`'), and in cfengine 3.x this is preserved.

```
/var/cfengine
/var/cfengine/bin
/var/cfengine/inputs
/var/cfengine/outputs
```

The installation location '/usr/local/sbin' is not necessarily a local file system, and cannot therefore be trusted to a) be present, and b) be authentic on an arbitrary system.

Similarly, a trusted cache of the input files must now be maintained in the 'inputs' subdirectory. When cfengine is invoked by the scheduler, it reads only from this directory. It is up to the user to keep this cache updated, on each host. This simplifies and consolidates the cfengine resources in a single place. The environment variable CFINPUTS still overrides this default location, as before, but in its absence or when called from the scheduler, this becomes the location of trusted files.

Unlike cfengine 2, cfengine 3 does not recognize the CFINPUTS environment variable.

The 'outputs' directory is now a record of spooled run-reports. These are often mailed to the administrator by cf-execd, or can be copied to another central location and viewed in an alternative browser.

1.4 Cfengine hard classes

Cfengine runs on every computer individually and each time it wakes up the underlying generic agent platform discovers and classifies properties of the environment or context in which it runs. This information is cached and may be used to make decisions about configuration[1].

Classes fall into hard (discovered) and soft (defined) types. A single class can be one of several things:

- The name of an operating system architecture e.g. ultrix, sun4, etc.
- The unqualified name of a particular host. If your system returns a fully qualified domain name for your host, cfengine truncates it at the first dot.
- The name of a user-defined group of hosts.
- A day of the week (in the form Monday, Tuesday, Wednesday, ..).
- An hour of the day (in the form Hr00, Hr01 ... Hr23).
- Minutes in the hour (in the form Min00, Min17 ... Min45).
- A five minute interval in the hour (in the form Min00_05, Min05_10 ... Min55_00)
- A day of the month (in the form Day1, Day2, ... Day31).
- A month (in the form January, February, ... December).
- A year (in the form Yr1997, Yr2004).
- An arbitrary user-defined string.
- The IP address octets of any active interface (in the form ipv4_192_0_0_1, ipv4_192_0_0, ipv4_192_0, ipv4_192).

To see all of the classes define on a particular host, run

```
host# cf-promises -v
```

as a privileged user. Note that some of the classes are set only if a trusted link can be established with cfenvd, i.e. if both are running with privilege, and the '/var/cfengine/state/env_data' file is secure. More information about classes can be found in connection with allclasses.

[1] There are no if-then-else statements in cfengine; all decisions are made with classes.

1.5 Global and local classes

Classes are defined in bundles. Bundles of type `common` yield classes that are global in scope, whereas in all other bundle types classes are local. Classes are evaluated when the bundle is evaluated. Consider the following example.

```
body common control
{
bundlesequence => { "g","tryclasses_1", "tryclasses_2" };
}

#################################

bundle common g
{
classes:

  "one" expression => "any";

}

#################################

bundle agent tryclasses_1
{
classes:

  "two" expression => "any";
}

#################################

bundle agent tryclasses_2
{
classes:

  "three" expression => "any";

reports:

  one.three.!two::

    "Success";
}
```

Here we see that class '`one`' is local while classes '`two`' and '`three`' are local. The report 'Success' result is therefore true because only '`one`' and '`three`' are in scope.

1.6 Filenames and paths

Filenames in Unix-like operating systems use for their directory separator the forward slash '/' character. All references to file locations must be absolute pathnames in cfengine, i.e. they must begin with a complete specification of which directory they are in. For example:

```
/etc/passwd
/usr/local/masterfiles/distfile
```

The only place where it makes sense to refer to a file without a complete directory specification is when searching through directories for different kinds of file, e.g. in pattern matching

```
leaf_name => { "tmp_.*", "output_file", "core" };
```

Here, one can write '`core`' without a path, because one is looking for any file of that name in a number of directories.

The Windows operating systems traditionally use a different filename convention. The following are all valid absolute file names under Windows:

```
c:\winnt
c:/winnt
/var/cfengine/inputs
//fileserver/share2/dir
```

The 'drive' name "C:" in Windows refers to a partition or device. Unlike Unix, Windows does not integrate these seamlessly into a single file-tree. This is not a valid absolute filename:

```
\var\cfengine\inputs
```

Paths beginning with a backslash are assumed to be win32 paths. They must begin with a drive letter or double-slash server name.

Note in recent versions of Cygwin you can decide to use the `/cygdrive` to specify a path to windows file E.g '`/cygdrive/c/myfile`' means '`c:\myfile`' or you can do it straight away in cfengine as `c:\myfile`.

1.7 Upgrading from cfengine 2

Cfengine 3 has a completely new syntax, designed to solve the issues brought up from 15 years of experience with configuration management. Rather than clutter cfengine 3 with buggy backward-compatability issues, it was decided to make no compromises with cfengine 3 and instead allow cfengine 2 and cfengine 3 to coincide in a cooperative fashion for the foreseeable future. This means that users can upgrade at their own pace, in the classic cfengine incremental fashion. We expect that cfengine 2 installations will be around for years to come so this upgrade path seems the most defensible.

The daemons and support services are fully interoperable between cfengine 2 and cfengine 3, so it does not matter whether you run `cfservd` (cf2) together with `cf-agent` (cf3) or `cf-serverd` (cf3) together with `cfagent` (cf2). You can change the servers at your own pace.

Cfengine 3's `cf-execd` replaces cfengine 2's `cfexecd` and it is designed to work optimally with `cf-agent` (cf3). Running this daemon has no consequences for access control, only for scheduling `cf-agent`. You can (indeed should) replace `cfexecd` with `cf-execd` immediately. You will want to alter your '`crontab`' file to run the new component instead of the old. The sample cfengine 3 input files asks `cf-agent` to do this automatically, simply replacing the string.

The sample '`inputs`' files supplied with cfengine 3 contain promises to integrate cfengine 2 as described. What can you do to upgrade? Here is a simple recipe that assumes you have a standardized cfengine 2 setup, running `cfexecd` in '`crontabs`' and possibly running `cfservd` and `cfenvd` as daemons.

1. Install the cfengine 3 software on a host.
2. Go to the '`inputs/`' directory in the source and copy these files to your master update repository, i.e. where you will publish policies for distribution.
3. Remove any self-healing rules to reinstall cfengine 2, especially rules to add `cfexecd` or `cfagent` to '`crontabs`' etc. Cfengine 3 will handle this from now on and encapsulate old cfengine 2 scripts.
4. Move to this inputs directory: `cd` *`your-path`*`/inputs`.
5. Set the location of this master update directory in the '`update.cf`' file to the location of the master directory.
6. Set the email options for the executor in '`promises.cf`'.
7. Run `cf-agent --bootstrap` as the root or privileged user. This will install cfengine 3 in place of cfengine 2, integrate your old cfengine 2 configuration, and warn you about any rules that need to be removed from your old cfengine configuration.
8. You should now be running cfengine 3. You can now add new rules to the files in your own time, or convert the old cfengine 2 rules and gradually comment them out of the cfengine 2 files.
9. Make sure there are no rules in your old cfengine 2 configuration to activate cfengine 2 components, i.e. rules that will fight against cfengine 3. Then, when you are ready, convert '`cfservd.conf`' into a server bundle e.g. in '`promises.cf`' and remove all rules to run `cfservd` and replace them with rules to run `cf-serverd` at your own pace.

1.8 Testing as a non-privilieged user

One of the practical advantages of cfengine is that you can test it without the need for root or administrator privileges. This is recommended for all new users of cfengine 3.

Cfengine operates with the notion of a work-directory. The default work directory for the `root` user is '`/var/cfengine`' (except on Debian Linux and various derivatives which prefer '`/var/lib/cfengine`'). For any other user, the work directory lies in the user's home directory, named '`~/.cfagent`'. Cfengine prefers you to keep certain files here. You should not resist this too strongly or you will make unnecessary trouble for yourself. The decision to have this 'known directory' was made to simplify a lot of configuration.

To test cfengine as an ordinary user, do the following:

- Compile and make the software.
- Copy the binaries into the work directory:

```
host$ mkdir -p ~/.cfagent/inputs
host$ mkdir -p ~/.cfagent/bin
host$ cd src
host$ cp cf-* ~/.cfagent/bin
host$ cd ../inputs
host$ cp *.cf ~/.cfagent/inputs
```

You can test the software and play with configuration files by editing the basic get-started files directly in the '`~/.cfagent/inputs`' directory. For example, try the following:

```
host$ ~/.cfagent/bin/cf-promises
host$ ~/.cfagent/bin/cf-promises --verbose
```

This is always the way to start checking a configuration in cfengine 3. If a configuration does not pass this check/test, you will not be allowed to use it, and '`cf-agent`' will look for the file '`failsafe.cf`'.

Notice that the cfengine 3 binaries have slightly different names than the cfengine 2 binaries. They all start with the '`cf-`' prefix.

```
host$ ~/.cfagent/bin/cf-agent
```

1.9 The 'bear' necessities of a cfengine 3

Here is the simplest 'Hello world' program in cfengine 3:

```
body common control
{
bundlesequence  => { "test" };
}

bundle agent test
{
reports:

 Yr2009::
    "Hello world";
}
```

If you try to process this using the `cf-promises` command, you will see something like this:

```
atlas$ ~/LapTop/Cfengine3/trunk/src/cf-promises -r -f ./unit_null_config.cf
Summarizing promises as text to ./unit_null_config.cf.txt
Summarizing promises as html to ./unit_null_config.cf.html
```

The '`-r`' option produces a report. Examine the files produced:

```
cat ./unit_null_config.cf.txt
firefox ./unit_null_config.cf.html
```

You will see a summary of how cfengine interprets the files, either in HTML or text. By default, the cfengine components also dump a debugging file, e.g. '`promise_output_agent.html`', '`promise_output_agent.txt`' with an expanded view.

1.10 Familiarizing yourself

To familiarize yourself with cfengine 3, type or paste in the following example text:

```
########################################################
#
# Simple test execution
#
########################################################

body common control

{
bundlesequence  => { "testbundle"  };
}

########################################################

bundle agent testbundle

{
```

```
vars:

  "size" int => "46k";
  "rand" int => randomint("33","$(size)");

commands:

  "/bin/echo"
     args => "Hello world - $(size)/$(rand)",
     contain => standard,
     classes => cdefine("followup","alert");

  followup::

     "/bin/ls"
       contain => standard;

reports:

  alert::

     "What happened?";

}

######################################################################

body contain standard

{
exec_owner => "mark";
useshell => "true";
}

######################################################################

body classes cdefine(class,alert)

{
promise_repaired => { "$(class)" };
repair_failed => { "$(alert)" };
}
```

If you are familiar with cfengine's history, this will look quite strange to you, but fear not.

This example shows all of the main features of cfengine: bundles, bodies, control, variables, and promises. To the casual eye it might look complex, but that is because it is explicit about all of the details. Fortunately it is easy to hide many of these details to make the example simpler without sacrificing any functionality.

The first thing to try with this example is to verify it – did we make any mistakes? Are there any inconsistencies? To do this we use the new cfengine program `cf-promises`. Let's assume that you typed this into a file called '`test.cf`' in the current directory.

```
cf-promises -f ./test.cf
```

If all is well, typing this command shows no output. Try now running the command with verbose output.

```
cf-promises -f ./test.cf -v
```

Now you see a lot of information

```
Reference time set to Sat Aug  2 11:26:06 2008

cf3 Cfengine - 3.0.0
Free Software Foundation 1994-
Donated by Mark Burgess, Oslo University College, Norway
cf3 ----------------------------------------------------------------------
cf3 Host name is: atlas
cf3 Operating System Type is linux
cf3 Operating System Release is 2.6.22.18-0.2-default
cf3 Architecture = x86_64
cf3 Using internal soft-class linux for host linux
cf3 The time is now Sat Aug  2 11:26:06 2008
cf3 ----------------------------------------------------------------------
cf3 Additional hard class defined as: 64_bit
cf3 Additional hard class defined as: linux_2_6_22_18_0_2_default
cf3 Additional hard class defined as: linux_x86_64
cf3 Additional hard class defined as: linux_x86_64_2_6_22_18_0_2_default
cf3 GNU autoconf class from compile time: compiled_on_linux_gnu
cf3 Interface 1: lo
cf3 Trying to locate my IPv6 address
cf3 Looking for environment from cfenvd...
cf3 Unable to detect environment from cfMonitord
-----------------------------------------------------------------------
Loading persistent classes
-----------------------------------------------------------------------

-----------------------------------------------------------------------
Loaded persistent memory
-----------------------------------------------------------------------
cf3   > Parsing file ./test.cf
-----------------------------------------------------------------------
Agent's basic classified context
-----------------------------------------------------------------------



Defined Classes = ( any Saturday Hr11 Min26 Min25_30 Q2 Hr11_Q2 Day2
August Yr2008 linux atlas 64_bit linux_2_6_22_18_0_2_default x86_64
linux_x86_64 linux_x86_64_2_6_22_18_0_2_default
linux_x86_64_2_6_22_18_0_2_default__1_SMP_2008_06_09_13_53_20__0200
compiled_on_linux_gnu net_iface_lo )

Negated Classes = ( )

Installable classes = ( )
cf3 Wrote expansion summary to promise_output_common.html
cf3 Inputs are valid
```

The last two lines of this are of interest. Each time a component of cfengine 3 parses a number of promises, it summarizes the information in an HTML file called generically `promise_output_`*`component-type`*`.html`. In this case the `cf-promises` command represents all possible promises, by the type "common". You can view this output file in a suitable web browser to see exactly what cfengine has understood by the configuration. The non-verbose output of the script when run in the cfengine 3 directory looks something like this:

```
host$ ./cf-agent -f ../tests/units/unit_exec_in_sequence.cf
Q ".../bin/echo Hello": Hello world - 46k/219
 -> Last 1 QUOTEed lines were generated by "/bin/echo Hello world - 46k/219"
Q ".../bin/ls": agent.c
Q ".../bin/ls": agentdiagnostic.c
Q ".../bin/ls": agentdiagnostic.o
Q ".../bin/ls": agent.o
Q ".../bin/ls": args.c
Q ".../bin/ls": args.lo
Q ".../bin/ls": args.o
...
Q ".../bin/ls": verify_reports.o
Q ".../bin/ls": verify_storage.c
Q ".../bin/ls": verify_storage.o
 -> Last 288 QUOTEed lines were generated by "/bin/ls"
atlas$
```

1.11 Remote access troubleshooting

1.11.1 Server connection

When setting up `cf-serverd`, you might see the error message

```
  Unspecified server refusal
```

This means that `cf-serverd` is unable or is unwilling to authenticate the connection from your client machine. The message is generic: it is deliberately non-specific so that anyone attempting to attack or exploit the service will not be given information which might be useful to them. There is a simple checklist for curing this problem:

1. Make sure that the domain variable is set in the configuration files read by both client and server; alternatively use `skipidentify` and `skipverify` to decouple DNS from the the authentication.
2. Make sure that you have granted access to your client in the server body

```
body server control
{
allowconnects         => { "127.0.0.1" , "::1" ...etc };
allowallconnects      => { "127.0.0.1" , "::1" ...etc };
trustkeysfrom         => { "127.0.0.1" , "::1" ...etc };
}
```

3. Make sure you have created valid keys for the hosts using `cf-key`.
4. If you are using secure copy, make sure that you have created a key file and that you have distributed and installed it to all participating hosts in your cluster.

Always remember that you can run cfengine in verbose or debugging modes to see how the authentication takes place:

```
cf-agent -v
cf-serverd -v
```

`cf-agent` reports that access is denied regardless of the nature of the error, to avoid giving away information which might be used by an attacker. To find out the real reason for a denial, use verbose '`-v`' or even debugging mode '`-d2`'.

1.11.2 Key exchange

The key exchange model used by cfengine is based on that used by OpenSSH. It is a peer to peer exchange model, not a central certificate authority model. This means that there are no scalability bottlenecks (at least by design, though you might introduce your own if you go for an overly centralized architecture).

The problem of key distribution is the conundrum of every public key infrastructure. Key exchange is handled automatically by cfengine and all you need to do is to decide which keys to trust.

When public keys are offered to a server, they could be accepted automatically on trust because no one is available to make a decision about them. This would lead to a race to be the first to submit a key claiming identity.

Even with DNS checks for correct name/IP address correlation (turned off with `skipverify`), it might be possible to submit a false key to a server.

The server `cf-serverd` blocks the acceptance of unknown keys by default. In order to accept such a new key, the IP address of the presumed client must be listed in the `trustkeysfrom` stanza. Once a key has been accepted, it will never be replaced with a new key, thus no more trust is offered or required.

Once you have arranged for the right to connect to the server, you must decide which hosts will have access to which files. This is done with `access` rules.

```
bundle server access_rules()

{
access:

  "/path/file"

    admit   => { "127.0.0.1", "127.0.0.2", "127.0.0.3" },
    deny    => { "192.*" };
}
```

On the client side, i.e. `cf-runagent` and `cf-agent`, there are three issues:

1. Choosing which server to connect to.
2. Trusting the identity of any previously unknown servers, i.e. trusting the server's public key to be its and no one else's. (The issues here are the same as for the server.)
3. Choosing whether data transfers should be encrypted (with `encrypt`).

Because there are two clients for connecting to `cf-serverd` (`cf-agent` and `cf-runagent`), there are also two ways on managing trust of server keys by a client. One is an automated option, setting the option `trustkey` in a `copy_from` stanza, e.g.

```
body copy_from example
     {
     # .. other settings ..
     trustkey => "true";
     }
```

Another way is to run `cf-runagent` in interactive mode. When you run `cf-runagent`, unknown server keys are offered to you interactively (as with `ssh`) for you to accept or deny manually:

```
WARNING - You do not have a public key from host ubik.iu.hio.no = 128.39.74.25
          Do you want to accept one on trust? (yes/no)
-->
```

1.11.3 Time windows (races)

Once public keys have been exchanged from client to server and from server to client, the issue of trust is solved according to public key authentication schemes. You only need to worry about trust when one side of a connection has never seen the other side before.

Often you will have a central server and many client satellites. Then the best way to transfer all the keys is to set the `trustkey` flags on server and clients sides to coincide with a time at which you know that `cf-agent` will be run, and when a spoofer is unlikely to be able to interfere.

This is a once-only task, and the chance of an attacker being able to spoof a key-transfer is small. It would require skill and inside-information about the exchange procedure, which would tend to imply that the trust model was already broken.

Another approach would be to run `cf-runagent` against all the hosts in the group from the central server and accept the keys one by one, by hand, though there is little to be gained from this.

Trusting a host for key exchange is unavoidable. There is no clever way to avoid it. Even transferring the files manually by diskette, and examining every serial number of the computers you have, the host has to trust the information you are giving it. It is all based on assertion. You can make it almost impossible for keys to be faked or attacked, but you cannot make it absolutely impossible. Security is about managing reasonable levels of risk, not about magic.

All security is based on a moment of trust at some point in time. Cryptographic key methods only remove the need for a repeat of the trust decision. After the first exchange, trust is no longer needed, because they keys allow identity to be actually verified.

Even if you leave the trust options switched on, you are not blindly trusting the hosts you know about. The only potential insecurity lies in any new keys that you have not thought about. If you use wildcards or IP prefixes in the trust rules, then other hosts might be able to spoof their way in on trust because you have left open a hole for them to exploit. That is why it is recommended to return the system to the default state of zero trust immediately after key transfer, by commenting out the trust options.

It is possible, though somewhat laborious to transfer the keys out of band, by copying '`/var/cfengine/ppkeys/localhost.pub`' to `/var/cfengine/ppkeys/user-aaa.bbb.ccc.mmm` (assuming IPv4) on another host. e.g.

```
     localhost.pub -> root-128.39.74.71.pub
```

This would be a silly way to transfer keys between nearby hosts that you control yourself, but if transferring to long distance, remote hosts it might be an easier way to manage trust.

1.11.4 Other users than root

Cfengine normally runs as user "root" (except on Windows which does not normally have a root user), i.e. a privileged administrator. If other users are to be granted access to the system, they must also generate a key and go through the same process. In addition, the users must be added to the server configuration file.

1.11.5 Encryption

Cfengine provides encryption for keeping file contents private during transfer. It is assumed that users will use this judiciously. There is nothing to be gained by encrypting the transfer of public files – overt use of encryption just contributes to global warming, burning unnecessary CPU cycles without offering any security.

The main role for encryption in configuration management is for authentication. Cfengine always uses encrypted for authentication, so none of the encryption settings affect the security of authentication.

2 A simple crash course in concepts

2.1 Rules are promises

Everything in cfengine 3 can be interpreted as a promise. Promises can be made about all kinds of different subjects, from file attributes, to the execution of commands, to access control decisions and knowledge relationships.

This simple but powerful idea allows a very practical uniformity in cfengine syntax. There is only one grammatical form for statements in the language that you need to know and it looks generically like this:

```
 type:

   classes::

    "promiser" -> { "promisee1", "promisee2", ... }

       attribute_1 => value_1,
       attribute_2 => value_2,
       ...
       attribute_n => value_n;
```

We speak of a promiser (the abstract object making the promise), the promisee is the abstract object to whom the promise is made, and them there is a list of associations that we call the 'body' of the promise, which together with the promiser-type tells us what it is all about.

Not all of these elements are necessary every time. Some promises contain a lot of implicit behaviour. In other cases we might want to be much more explicit. For example, the simplest promise looks like this:

```
commands:

  "/bin/echo hello world";
```

This promise has default attributes for everything except the 'promiser', i.e. the command string that promises to execute. A more complex promise contains many attributes:

```
files:

  "/home/mark/tmp/test_plain" -> "system blue team",

       comment => "This comment follows the rule for knowledge integration",
       perms   => users("@(usernames)"),
       create  => "true";
```

The list of promisees is not used by cfengine except for documentation, just as the comment attribute (which can be added to any promise) has no actual function other than to provide more information to the user in error tracing and auditing.

You see several kinds of object in this example. All literal strings (e.g. `"true"`) in cfengine 3 must be quoted. This provides absolute consistency and makes type-checking easy and error-correction powerful. All function-like objects (e.g. `users("..")`) are either builtin special functions or parameterized templates which contain the 'meat' of the right hand side.

2.2 Containers

Cfengine allows you to group multiple promise statements into containers called bundles.

```
bundle agent identifier

{
commands:

  "/bin/echo These commands are a silly way to use cfengine";
  "/bin/ls -l";
  "/bin/echo But they illustrate a point";

}
```

Bundles serve two purposes: they allow us to collect related promises under a single heading, like a subroutine, and they allow us to mix configuration for different parts of cfengine in the same file. The type of a bundle is the name of the component of cfengine for which it is intended.

For instance, we can make a self-contained example agent-server configuration by labelling the bundles:

```
#
# Not a complete example
#

bundle agent testbundle

{
files:

  "/home/mark/tmp/testcopy"

    copy_from    => mycopy("/home/mark/LapTop/words","127.0.0.1"),
    perms        => system,
    depth_search => recurse("inf");

}

#

bundle server access_rules

{
access:

  "/home/mark/LapTop"

    admit   => { "127.0.0.1" };
}
```

Another type of container in cfengine 3 is a 'body' part. Body parts exist to hide complex parameter information in reusable containers. The right hand side of some attribute assignments use body containers to reduce the amount of in-line information and preserve readability. You cannot choose where to use bodies: either they are used or they are not used for a particular kind of attribute. What you can choose, however, is the name and number of parameters for the body; and you can make as many of them as you like: For example:

```
body copy_from mycopy(from,server)

{
source      => "$(from)";
servers     => { "$(server)" };
copy_backup => "true";

special_class::

  purge       => "true";
}
```

Notice also that classes can be used in bodies as well as parameters so that you can hide environmental adaptations in these bodies also. The classes used here are effectively ANDed with the classes under which the calling promise is defined.

2.3 When and where are promises made?

When you type a promise into a cfengine bundle, the promise will be read by every cf-agent that reads the file, each time it is called into being. For some promises this is okay, but for others you only want to verify the promise once in a while, e.g. once per day or once per hour. There are two ways to say when and where a promise applies in cfengine:

Classes Classes are the double-colon decision syntax in cfengine. They determine in what context a promise is made, i.e. when and where. Recall the basic syntax of a promise:

```
promise-type:

   class-expression::

      promiser -> promisee

         attribute => body,
              ifvarclass => other-class-expression;
```

The class expression may contain words like 'Hr12', meaning from 12:00 p.m - 13:00 p.m., or 'Hr12&Min05_10', meaning between 12:05 and 12:10. Classes may also have spatial descriptors like 'myhost' or 'solaris', which decide which hosts in the namespace, or 'ipv4_192_168_1_101' which decides the location in IPv4 address space.

If the class expression is true, the promise can be considered made for the duration of the current execution.

Cfengine 3 has a new class predicate `ifvarclass` which is ANDed with the normal class expression, and which is evaluated together with the promise. It may contain variables as long as the resulting expansion is a legal class expression.

Locks Locks determine how often a promise is verified.

Cfengine is controlled by a series of locks which prevent it from checking promises too often, and which prevent it from spending too long trying to verify promises it already verified recently. The locks work in such a way that you can start several cfengine processes simultaneously without them interfering with each other. You can control two things about each kind of action in the action sequence:

'`ifelapsed`'
: The minimum time which should have passed since the last time that promise was verified. It will not be executed again until this amount of time has elapsed. (Default time is 1 minute.)

'`expireafter`'
: The maximum amount of time cf-agent should wait for an old instantiation to finish before killing it and starting again. (Default time is 120 minutes.)

You can set these values either globally (for all actions) or for each action separately. If you set global and local values, the local values override the global ones. All times are written in units of *minutes*. Global setting is in the control body:

```
body agent control
{
ifelapsed => "60";
}
```

or locally in the transaction bodies:

```
body action example
{
ifelapsed => "60";
}
```

These locks do not prevent the whole of cf-agent from running, only atomic promise checks. Several different atoms can be run concurrently by different cf-agents. The locks ensure that atoms will never be started by two cf-agents at the same time, or too soon after a verification, causing contention and wasting CPU cycles.

2.4 Types in cfengine 3

A key difference in cfengine 3 compared to earlier versions is the presence of data types. Data types are a mechanism for associating values and checking consistency in a language. Once again, there is a simple pattern to types in cfengine.

The principle is very simple: types exist in order to match like a plug-socket relationship. In the examples above, you can see two places where types are used to match templates:

- Matching bundles to components:

  ```
  bundle TYPE name  # matches TYPE to running agent
  {
  }
  ```

- Match bodies templates to lvalues in `lvalues => rvalue` constraints:

  ```
  body TYPE name    # matches TYPE => name in promise
  ```

```
         {
         }
```

Check these by identifying the words '`agent`' and '`copy_from`' in the examples above. Types are there to make configuration more robust.

2.5 Normal ordering

Ordering of promise verification within cfengine takes a pragmatic approach. In the absence of dependencies (independent) promises can be checked in any order. However, one can encode both implicit and explicit ordering dependency.

Cfengine takes a pragmatic point of view to ordering, since certain data structures require ordering to be preserved, e.g. editing in files. The rules are as follows:

1. Cfengine executes promise bundles in the strict order defined by the `bundlesequence`.
2. Within a bundle, the promise types are executed in a round-robin fashion according to so-called 'normal ordering' (essentially deletion first followed by creation). The actual sequence continues for up to three iterations of the following:

   ```
   vars
   classes
   interfaces
   processes
   storage
   packages
   commands
   methods
   files
   reports
   ```

   Within line_editing bundles, the normal ordering is:

   ```
   vars
   classes
   delete_lines
   field_edits
   replace_patterns
   insert_lines
   reports
   ```

3. The order of promises within one of the above types follows their top-down ordering within the bundle itself.
4. The order may be overridden by making a promise depend on a class that is set by another promise.

2.6 Loops and lists in cfengine 3

There are no explicit loops in cfengine, instead there are lists. To make a loop, you simply refer to a list as a scalar and cfengine will assume a loop over all items in the list.

For example, in the excepts below the list `component` has three elements. The list as a whole may be referred to as `@(component)`, in order to pass the whole list to a promise where a list is expected.

However, if we write `$(component)`, i.e. the scalar variable, then cfengine assumes that it should substitute each scalar from the list in turn, and thus iterate over the list elements using a loop.

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };

  "new_list" slist => { "cf-know", @(component) };

processes:

  "$(component)" restart_class => canonify("start_$(component)");

commands:

   "/bin/echo /var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
}
```

If a variable is repeated, its value is tied throughout the expression; so the output of:

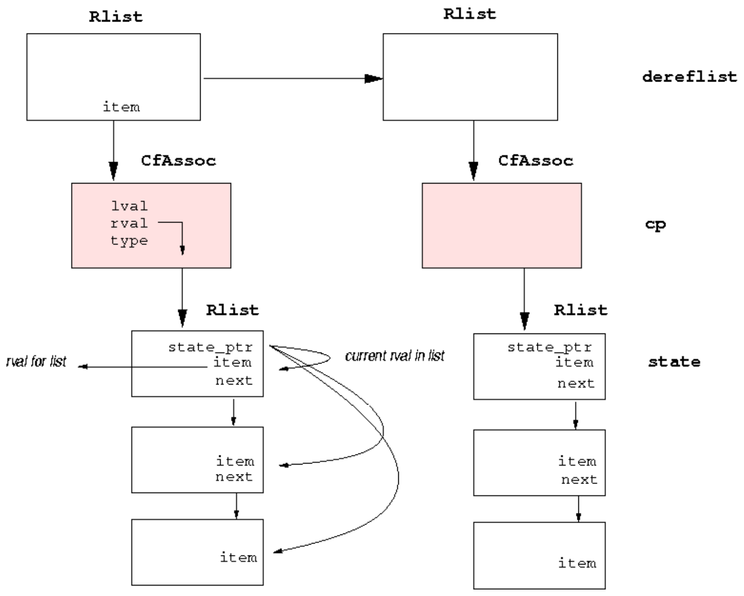
```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:
  "component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };

  "array[cf-monitord]" string => "The monitor";
  "array[cf-serverd]" string => "The server";
  "array[cf-execd]" string => "The executor, not executionist";

commands:

   "/bin/echo $(component) is"

            args => "$(array[$(component)])";
}
```

is as follows:

```
Q ".../bin/echo cf-mo": cf-monitord is The monitor
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-monitord is The monitor"
Q ".../bin/echo cf-se": cf-serverd is The server
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-serverd is The server"
Q ".../bin/echo cf-ex": cf-execd is The executor, not executionist
 -> Last 1 QUOTEed lines were generated by "/bin/echo cf-execd is The executor, not executionist"
```

2.7 Developer structures

Developers may note that the internal data-types follow a simple set of internal linked lists, as in the diagram below.



3 How to run cfengine 3 examples

The cfengine ‘`tests`’ directory contains a multitude of examples of cfengine 3 code. These instructions assume that you have all of your configuration in a single test file, such as the example in the distribution directory ‘`tests/units`’.

1. Test the file as a non-privileged user first, if you can.
2. Always verify syntax first with `cf-promises`. This requires no privileges. An `cf-agent` will not execute a configuration that has not passed this test.

   ```
   host$ cf-promises -f ./inputfile.cf
   ```

3. Run the examples like this, e.g.

   ```
   host$ src/cf-promises -f ./tests/units/unit_server_copy_localhost.cf
   host$ src/cf-serverd -f ./tests/units/unit_server_copy_localhost.cf
   host$ src/cf-agent -f ./tests/units/unit_server_copy_localhost.cf
   ```

Running `cf-agent` in verbose mode provides detailed information about the state of the systems promises.

```
Outcome of version 1.2.3: Promises observed to be kept 99%,
Promises repaired 1%, Promises not repaired 0%
```

The log-file ‘`WORKDIR/promise.log`’ contains the summary of these reports with timestamps. This is the simplest kind of high level audit record of the system.

4 A complete configuration

To illustrate a complete configuration for agents and daemons, consider the following example code, supplied in the ‘`inputs/`’ directory of the distribution. Comments indicate the thinking behind this starting point.

4.1 ‘promises.cf’

This file is the first file that `cf-agent` with no arguments will try to look for. It should contain all of the basic configuration settings, including a list of other files to include. It should have a `bundlesequence`.

This file can stay fixed, except for extending the bundlesequence. The bundlesequence acts like the ‘genetic makeup’ of the configuration. In a large configuration, you will want to have a different bundlesequence for different classes of host. Use list variables to paste in list fragments to describe the make-up of each class.

```
#######################################################
#
# promises.cf
#
#######################################################

body common control

{
# List the ‘genes’ for this system..

bundlesequence  => {
                   "update",
                   "garbage_collection",
                   "main",
                   "cfengine"
                   };


inputs          => {
                   "update.cf",
                   "site.cf",
                   "library.cf"
                   };
}

#######################################################
# Now set defaults for all components’ hard-promises
#######################################################

body agent control
{
# if default runtime is 5 mins we need this for long jobs
ifelapsed => "15";
}

#######################################################

body monitor control
{
forgetrate => "0.7";
histograms => "true";
}

#######################################################

body executor control

{
splaytime => "1";
mailto => "cfengine_mail@example.org";
smtpserver => "localhost";
mailmaxlines => "30";

# Instead of a separate update script, now do this

exec_command => "$(sys.workdir)/bin/cf-agent -f failsafe.cf && $(sys.workdir)/bin/cf-agent";
}

#######################################################

body reporter control

{
reports => { "performance", "last_seen", "monitor_history" };
build_directory => "/tmp/nerves";
report_output => "html";
}

#######################################################

body runagent control
{
hosts => {
          "127.0.0.1"
          # , "myhost.example.com:5308", ...
         };

}
```

```
#######################################################

body server control

{
allowconnects         => { "127.0.0.1" , "::1" };
allowallconnects      => { "127.0.0.1" , "::1" };
trustkeysfrom         => { "127.0.0.1" , "::1" };

# Make updates and runs happen in one

cfruncommand =>

 "$(sys.workdir)/bin/cf-agent -f failsafe.cf && $(sys.workdir)/bin/cf-agent";

allowusers   => { "root" };
}
```

4.2 'site.cf'

Use this file to add your site-specific configuration. Common bundles can be used to define global variables. Otherwise, unqualified variables are local to the bundle in which they are defined – however they can be access by writing `$(bundle.variable)`.

```
#######################################################
#
# site.cf
#
#######################################################

bundle common g
{
vars:

  SuSE::

   "crontab" string => "/var/spool/cron/tabs/root";

 !SuSE::

   "crontab" string => "/var/spool/cron/crontabs/root";
}
```

The cfengine bundle below detects whether cfengine 2 is already running on the host or not, and if so attempts to kill off old daemon processes and encapsulate the agent. It also looks for rules in the old cfengine configuration that would potentially spoil cfengine 3's control of the system: the last thing we want is for cfengine 2 and cfengine 3 to fight each other for control of the system. Cfengine

3 tries to edit an existing crontab entry to replace any references to `cfexecd` with `cf-execd`; if none are found it will add a 5 minute run schedule. You should never put `cf-agentor` `cf-agent` directly inside `cron` without the `cf-execd` wrapper.

```
#######################################################
# Start with cfengine itself
#######################################################

bundle agent cfengine

{
classes:

  "integrate_cfengine2"

      and => {
             fileexists("$(sys.workdir)/inputs/cfagent.conf"),
             fileexists("$(sys.workdir)/bin/cfagent")
             };

vars:

   "cf2bits" slist => { "cfenvd", "cfservd", "cfexecd" };

commands:

 integrate_cfengine2::

   "$(sys.workdir)/bin/cfagent"

        action => longjob;

files:

  # Warn about rules relating to cfengine 2 in inputs - could conflict

  "$(sys.workdir)/inputs/.*"

       comment     => "Check if there are still promises about cfengine 2 that need removing",
       edit_line   => DeleteLinesMatching(".*$(cf2bits).*"),
       file_select => OldCf2Files,
       action      => WarnOnly;

  # Check cf-execd and schedule is in crontab


  "$(g.crontab)"
```

```
       edit_line => upgrade_cfexecd;

processes:

  exec_fix::

    "cron" signals => { "hup" };


}

#######################################################
# General site issues can be in bundles like this one
#######################################################

bundle agent main

{
vars:

  "component" slist => { "cf-monitord", "cf-serverd" };

 # - - - - - - - - - - - - - - - - - - - - - - - - -

files:

  "$(sys.resolv)"  # test on "/tmp/resolv.conf" #

     create        => "true",
     edit_line     => resolver,
     edit_defaults => def;

 # Uncomment this to perform a change-detection scan

 #  "/usr"
 #     changes      => lay_trip_wire,
 #     depth_search => recurse("inf"),
 #     action       => measure;

processes:

  "cfenvd"              signals => { "term" };

 # Uncomment this when you are ready to upgrade the server
 #
 #  "cfservd"              signals => { "term" };
 #
 # Now make sure the new parts are running, cf-serverd will fail if
 # the old server is still running

  "$(component)" restart_class => canonify("start_$(component)");

 # - - - - - - - - - - - - - - - - - - - - - - - -

commands:

   "$(sys.workdir)/bin/$(component)"

       ifvarclass => canonify("start_$(component)");

}
```

This section takes a backup of a user home directory. This is especially useful for a single laptop or personal workstation that does not have a regular external backup. If a user deletes a file by accident, this shadow backup might contain the file even while travelling offline.

```
#######################################################
# Backup
#######################################################

bundle agent backup
{
files:

  "/home/backup"

     copy_from => cp("/home/mark"),
  depth_search => recurse("inf"),
   file_select => exclude_files,
        action => longjob;

}

#######################################################
# Garbage collection issues
#######################################################

bundle agent garbage_collection
{
files:

  "$(sys.workdir)/outputs"
```

```
    delete => tidy,
    file_select => days_old("3"),
    depth_search => recurse("inf");


}

###########################################################

body file_select OldCf2Files
{
leaf_name => {
             "promises.cf",
             "site.cf",
             "library.cf",
             "failsafe.cf",
             ".*.txt",
             ".*.html",
             ".*~",
             "#.*"
             };

file_result => "!leaf_name";
}

###########################################################

body action measure
{
measurement_class => "Detect Changes in /usr";
ifelapsed => "240";
expireafter => "240";
}
```

Some basic anomaly detection: we respond with simple warnings if resource anomalies are detected.

```
#######################################################
# Anomaly monitoring
#######################################################

bundle agent anomalies
{
reports:

rootprocs_high_dev2::
   "RootProc anomaly high 2 dev on $(sys.host) at $(sys.env_time)
    measured value $(sys.value_rootprocs) av $(sys.average_rootprocs)
    pm $(sys.stddev_rootprocs)"

      showstate => { "rootprocs" };

entropy_www_in_high&anomaly_hosts.www_in_high_anomaly::

   "HIGH ENTROPY Incoming www anomaly high anomaly dev!!
    on $(sys.host) at $(sys.env_time)
    - measured value $(sys.value_www_in)
    av $(sys.average_www_in) pm $(sys.stddev_www_in)"

      showstate => { "incoming.www" };

 entropy_www_in_low.anomaly_hosts.www_in_high_anomaly::

   "LOW ENTROPY Incoming www anomaly high anomaly dev!!
    on $(sys.host) at $(sys.env_time)
     - measured value $(svalue_www_in)
    av $(average_www_in) pm $(stddev_www_in)"

     showstate => { "incoming.www" };

entropy_tcpsyn_in_low.anomaly_hosts.tcpsyn_in_high_dev2::

   "Anomalous number of new TCP connections on $(sys.host)
    at $(sys.env_time)
    - measured value $(sys.value_tcpsyn_in)
    av $(sys.average_tcpsyn_in) pm $(sys.stddev_tcpsyn_in)"

     showstate => { "incoming.tcpsyn" };

 entropy_dns_in_low.anomaly_hosts.dns_in_high_anomaly::

   "Anomalous (3dev) incoming DNS packets on $(sys.host)
    at $(sys.env_time) - measured value $(sys.value_dns_in)
    av $(average_dns_in) pm $(sys.stddev_dns_in)"

     showstate => { "incoming.dns" };

 entropy_dns_in_low.anomaly_hosts.udp_in_high_dev2::

   "Anomalous (2dev) incoming (non-DNS) UDP traffic
    on $(sys.host) at $(sys.env_time) - measured value
    $(sys.value_udp_in) av $(sys.average_udp_in) pm $(sys.stddev_udp_in)"
```

```
     showstate => { "incoming.udp" };

 anomaly_hosts.icmp_in_high_anomaly.!entropy_icmp_in_high::

   "Anomalous low entropy (3dev) incoming ICMP traffic
    on $(sys.host) at $(sys.env_time) - measured value $(sys.value_icmp_in)
    av $(sys.average_icmp_in) pm $(sys.stddev_icmp_in)"

     showstate => { "incoming.icmp" };
}
```

Server access rules are a touchy business. In an enterprise setting you generally want every host to allow a monitoring host to be able to download data, and a backup host to be able to access important data on every host. On a laptop or personal workstation, there might not be any reason to run a server for external use; however you might configure it as below to allow localhost access for testing.

```
#######################################################
# Server configuration
#######################################################

bundle server access_rules()
{
access:

  "/home/mark/test_area"

    admit   => { "127.0.0.1" };

  # Rule for cf-runagent

  "/home/mark/.cfagent/bin/cf-agent"

    admit   => { "127.0.0.1" };

# New in cf3 - RBAC with cf-runagent

roles:

  ".*"  authorize => { "mark" };
}
```

4.3 ‘update.cf’

This file should never change. If you ever change it, or when you upgrade cfengine, make sure the old and the new cfengine can parse and execute this file. If not, you risk losing control of your system.

```
#########################################################
```

```
#
# update.cf
#
#########################################################

bundle agent update
{
vars:

 "master_location" string => "/your/master/cfengine-inputs";

files:

  # Update the configuration

  "/var/cfengine/inputs"

    perms => system("600"),
    copy_from => mycopy("$(master_location)","localhost"),
    depth_search => recurse("inf"),
    action => immediate;

  # Update the software cache

  "/var/cfengine/bin"

    perms => system("700"),
    copy_from => mycopy("/usr/local/sbin","localhost"),
    depth_search => recurse("inf"),
    action => immediate;
}

############################################

body perms system(p)

{
mode  => "$(p)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };
file_result => "leaf_name";
}

#########################################################

body copy_from mycopy(from,server)

{
source      => "$(from)";
compare     => "digest";
}

#########################################################

body action immediate
{
ifelapsed => "1";
}
```

4.4 ‘failsafe.cf’

This file should never change. If you ever change it, or when you upgrade cfengine, make sure the old and the new cfengine can parse and execute this file. If not, you risk losing control of your system.

```
#########################################################
#
# Failsafe file
#
#########################################################

body common control

{
bundlesequence => { "update" };

inputs => { "update.cf" };
}

############################################

body depth_search recurse(d)

{
depth => "$(d)";
}
```

4.5 What should a failsafe and update file contain?

The ‘`failsafe.cf`’ file is to make sure that your system can upgrade gracefully to new versions even when mistakes are made.

As a general rule:

- Upgrade the software first, then add new features to the configuration.
- Never use advanced features in the failsafe or update file.
- Avoid using library code. Paste it in explicitly using a special name that does not collide with a name in library. The update process should not have *any* dependencies.

A cfengine configuration will fail-over to the failsafe configuration if it is unable to read or parse the contents successfully. That means that any new features you try in a configuration will cause a fail-over, because the parser will not be able to interpret the new features until the software itself has been updated.

4.6 Recovery from errors in the configuration

The ‘`failsafe.cf`’ file should be able to download the latest master configuration from source always.

```
#######################################################
#
# failsafe.cf
#
#######################################################

body common control

{
bundlesequence => { "update" };
}

#########################################################

bundle agent update
{
files:

  "/var/cfengine/inputs"

    perms => system,
    copy_from => mycopy("/home/mark/cfengine-inputs","localhost"),
    depth_search => recurse("inf");

  "/var/cfengine/bin"

    perms => system,
    copy_from => mycopy("/usr/local/sbin","localhost"),
    depth_search => recurse("inf");
}

#########################################################

body perms system

{
mode  => "0700";
}

#########################################################

body depth_search recurse(d)

{
depth => "$(d)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };

file_result => "leaf_name";
}

#########################################################

body copy_from mycopy(from,server)

{
source       => "$(from)";
servers      => { "$(server)" , "failover.domain.tld" };
#copy_backup => "true";
#trustkey    => "true";
encrypt      => "true";
}
```

If the `copy_backup` option is true, cfengine will keep a single previous version of the file before copy, if the value is '`timestamp`' cfengine keeps time-stamped versions either in the location of the file, or in the file repository if one is defined. The `trustkey` option should normally be commented out so that public keys are only exchanged under controlled conditions.

4.7 Recovery from errors in the software

The update should optionally include an update of software so that a single failover from a configuration that is 'too new' for the software will still correct itself once the new software is available.

```
#######################################################
#
# update.cf
#
#######################################################

bundle agent update

{
files:

  "/var/cfengine/inputs"

    perms => system("600"),
    copy_from => mycopy("/home/mark/cfengine-inputs","localhost"),
    depth_search => recurse("inf");

  "/var/cfengine/bin"

    perms => system("700"),
    copy_from => mycopy("/usr/local/sbin","localhost"),
    file_select => cf3_files,
    depth_search => recurse("inf");

}

############################################

body perms system(p)

{
mode  => "$(p)";
}

############################################

body file_select cf3_files

{
leaf_name => { "cf-.*" };

file_result => "leaf_name";
}

#########################################################

body copy_from mycopy(from,server)

{
source      => "$(from)";
compare     => "digest";
}
```

5 Control promises

While promises to configure your system are entirley user-defined, the details of the operational behaviour of the cfengine software is of course hard-coded. You can still configure the details of this behaviour using the control promise bodies. Control behaviour is defined in bodies because the actual promises are fixed and you only change their details within sensible limits.

Note that in cfengine's previous versions, the `control` part of the configuration contained a mixture of internal control parameters and user definitions. There is now a cleaner separation in cfengine 3. User defined behaviour requires a promise, and must therefore be defined in a bundle.

Below is a list of the control parameters for the different components (Agents and Daemons[1]) of the cfengine software.

5.1 `common` control promises

```
body common control

{
inputs  => {
           "update.cf",
           "library.cf"
           };

bundlesequence  => {
                   update("policy_host.domain.tld"),
                   "main",
                   "cfengine2"
                   };

output_prefix => "cfengine>";
version => "1.2.3";
}
```

the `common` control body refers to those promises that are hard-coded into all the components of cfengine, and therefore affect the behaviour of all the components.

5.1.1 `bundlesequence`

Type: slist

Allowed input range: `.*`

Synopsis: List of promise bundles to verify in order

[1] There is no Da Vinci code in cfengine

Example:

```
body common control

{
#..
bundlesequence  => {
                   update("policy_host.domain.tld"),
                   "main",
                   "cfengine2"
                   };
}
```

Notes:

The `bundlesequence` determines whether compiled bundles will be executed and in what order they will be executed. The list refers to the names of bundles which might be parameterized function-like objects.

The `bundlesequence` is like a genetic makeup of a machine. The bundles act like characteristics of the systems. If you want different systems to have different bundlesequences, distinguish them with classes:

```
webservers::

  bundlesequence => { "main", "web" };

others::

  bundlesequence => { "main", "otherstuff" };
```

If you want to add a basic common sequence to all sequences, then use global variable lists to do this:

```
body agent control
{
webservers::

  bundlesequence => { @(g.bs), "web" };

others::

  bundlesequence => { @(g.bs), "otherstuff" };

}

bundle common g
{
```

```
vars:

  "bs" slist => { "main", "basic_stuff" };
}
```

5.1.2 `inputs`

Type: slist
Allowed input range: `.*`
Synopsis: List of filenames to parse for promises
Example:

```
body common control
{
inputs  => {
           "update.cf",
           "library.cf"
           };
}
```

Notes:

5.1.3 `version`

Type: string
Allowed input range: (arbitrary string)
Synopsis: Scalar version string for this configuration
Example:

```
body common control
{
version => "1.2.3";
}
```

Notes:

The version string is used in error messages and reports.

5.1.4 `lastseenexpireafter`

Type: int

Allowed input range: `0,99999999999`
Synopsis: Number of minutes after which last-seen entries are purged
Example:

Notes:

```
body common control
{
lastseenexpireafter => "72";
}
```

5.1.5 `output_prefix`

Type: string
Allowed input range: (arbitrary string)
Synopsis: The string prefix for standard output
Example:

```
body common control
{
output_prefix => "my_cf3";
}
```

Notes:

5.1.6 `domain`

Type: string
Allowed input range: `.*`
Synopsis: Specify the domain name for this host
Example:

```
body common control
{
domain => "example.org";
}
```

Notes:

There is no standard, universal or reliable way of determining the DNS domain name of a host, so it can be set explicitly to simplify discovery and name-lookup.

5.2 agent control promises

```
body agent control
{
123_456_789::

  domain => "mydomain.com";

123_456_789_111::

  auditing => "true";

any::

  fullencryption => "true";

}
```

Settings describing the details of the fixed behavioural promises made by `cf-agent`. For example:

5.2.1 maxconnections

Type: int
Allowed input range: `0,99999999999`
Synopsis: Maximum number of outgoing connections to cf-serverd
Example:

```
# client side

body agent control
{
maxconnections => "1000";
}

# server side

body server control
```

```
{
maxconnections => "1000";
}
```

Notes:

Watch out for kernel limitations for maximum numbers of open file descriptors which can limit this.

5.2.2 `abortclasses`

Type: slist
Allowed input range: `.*`
Synopsis: A list of classes which if defined lead to termination of cf-agent
Example:

```
 body agent control

  {
  abortclasses => { "danger", "should_not_continue" };
  }
```

Notes:

A list of classes that `cf-agent` will watch out for. If any of these classes becomes defined, it will cause the current execution of `cf-agent` to be aborted. This may be used for validation, for example.

5.2.3 `abortbundleclasses`

Type: slist
Allowed input range: `.*`
Synopsis: A list of classes which if defined lead to termination of current bundle
Example:

This example shows how to use the feature to validate input to a method bundle.

```
body common control

{
bundlesequence  => { "testbundle"  };
version => "1.2.3";
}
```

```
###########################################

body agent control

{
abortbundleclasses => { "invalid" };
}

###########################################

bundle agent testbundle
{
vars:

 "userlist" slist => { "xyz", "mark", "jeang", "jonhenrik", "thomas", "eben" };

methods:

 "any" usebundle => subtest("$(userlist)");

}

###########################################

bundle agent subtest(user)

{
classes:

  "invalid" not => regcmp("[a-z][a-z][a-z][a-z]","$(user)");

reports:

 !invalid::

  "User name $(user) is valid at 4 letters";
}
```

Notes:

A list of classes that `cf-agent` will watch out for. If any of these classes becomes defined, it will cause the current bundle to be aborted. This may be used for validation, for example.

5.2.4 `addclasses`

Type: slist

Allowed input range: `.*`
Synopsis: A list of classes to be defined always in the current context
Example:

Add classes adds global, literal classes. The only predicates available during the control section are hard-classes.

```
any::

  addclasses => { "My_Organization" }

solaris::

  addclasses => { "some_solaris_alive", "running_on_sunshine" };
```

Notes:

Another place to make global aliases for system hardclasses. Classes here are added unqeuivocally to the system. If classes are used to predicate definition, then they must be defined in terms of global hard classes.

5.2.5 `agentaccess`

Type: slist
Allowed input range: `.*`
Synopsis: A list of user names allowed to execute cf-agent
Example:

```
 agentaccess => { "mark", "root", "sudo" };
```

Notes:

A list of user names that will be allowed to attempt execution of the current configuration. This is mainly a sanity check rather than a security measure.

5.2.6 `agentfacility`

Type: (menu option)
Allowed input range:

```
               LOG_USER
               LOG_DAEMON
               LOG_LOCAL0
               LOG_LOCAL1
               LOG_LOCAL2
               LOG_LOCAL3
               LOG_LOCAL4
               LOG_LOCAL5
               LOG_LOCAL6
               LOG_LOCAL7
```

Synopsis: The syslog facility for cf-agent

Example:

```
agentfacility => "LOG_USER";
```

Notes:

Sets the agent's syslog facility level. See the manual pages for syslog.

5.2.7 auditing

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false flag to activate the cf-agent audit log

Example:

```
body agent control
{
auditing  => "true";
}
```

Notes:

If this is set, cfengine will perform auditing on promises in the current configuration. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in cfengine 2.

5.2.8 `binarypaddingchar`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Character used to pad unequal replacements in binary editing

Example:

```
body agent control
{
binarypaddingchar => "$(const.0)";
}
```

Notes:

When editing binary files, it can be dangerous to replace a text string with one that is longer or shorter as byte references and jumps would be destroyed. Cfengine will therefore not allow replacements that are larger in size than the original, but shorter strings can be padded out to the same length.

5.2.9 `bindtointerface`

Type: string

Allowed input range: `.*`

Synopsis: Use this interface for outgoing connections

Example:

```
bindtointerface => "192.168.1.1";
```

Notes:

On multi-homed hosts, the server and client can bind to a specific interface for server traffic. The IP address of the interface must be given as the argument, not the device name.

5.2.10 `hashupdates`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether stored hashes are updated when change is detected in source
Example:

```
body agent control
{
hashupdates => "true";
}
```

Notes:

If '`true`' the stored reference value is updated as soon as a warning message has been given. As most changes are benign (package updates etc) this is a common setting.

5.2.11 `childlibpath`

Type: string
Allowed input range: `.*`
Synopsis: LD_LIBRARY_PATH for child processes
Example:

```
body agent control
{
childlibpath => "/usr/lcoal/lib:/usr/local/gnu/lib";
}
```

Notes:

This string may be used to set the internal `LD_LIBRARY_PATH` environment of the agent.

5.2.12 `defaultcopytype`

Type: (menu option)
Allowed input range:

```
               mtime
               atime
```

```
                ctime
                digest
                hash
                binary
```

Synopsis: (null)

Example:

```
body agent control
{
#...
defaultcopytype => "digest";
}
```

Notes:

Sets the global default policy for comparing source and image in copy transactions.

5.2.13 dryrun

Type: (menu option)

Allowed input range:

```
                true
                false
                yes
                no
                on
                off
```

Synopsis: All talk and no action mode

Example:

```
body agent control
{
dryrun => "true";
}
```

Notes:

If set in the configuration, cfengine makes no changes to a system, only reports what it needs to do.

5.2.14 `editbinaryfilesize`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Integer limit on maximum binary file size to be edited

Example:

```
body agent control
{
edibinaryfilesize => "10M";
}
```

Notes:

The limit on editing binary files should generally be higher than for text files. Note the use of units allowed in the integer type.

5.2.15 `editfilesize`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Integer limit on maximum text file size to be edited

Example:

```
body agent control
{
editfilesize => "120k";
}
```

Notes:

The global setting for the file-editing safety-net. Note the use of special units is allowed.

5.2.16 `environment`

Type: slist

Allowed input range: `[A-Za-z_]+=.*`

Synopsis: List of environment variables to be inherited by children

Example:

```
body common control
{
bundlesequence => { "one" };
}

body agent control
{
environment => { "A=123", "B=456", "PGK_PATH=/tmp"};
}

bundle agent one
{
commands:

  "/usr/bin/env";
}
```

Notes:

This may be used to set the runtime environment of the agent process. The values of environment variables are inherited by child commands. Some interactive programs insist on values being set, e.g.

```
# Required by apt-cache, debian

environment => { "LANG=C"};
```

5.2.17 `exclamation`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false print exclamation marks during security warnings

Example:

```
body agent control
{
exclamation => "false";
}
```

Notes:

This affects only the output format of warnings.

5.2.18 `expireafter`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Global default for time before on-going promise repairs are interrupted

Example:

```
body action example
{
ifelapsed   => "120";
expireafter => "240";
}
```

Notes:

The locking time after which cfengine will attempt to kill and restart its attempt to keep a promise.

5.2.19 `files_single_copy`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of filenames to be watched for multiple-source conflicts

Example:

```
body agent control
{
single_copy => { "/etc/.*", "/special/file" };
}
```

Notes:

This list of regular expressions will ensure that files matching the patterns of the list are never copied from more than one source during a single run of `cf-agent`. This may be considered a protection against accidential overlap of copies from diverse remote sources, or as a first-come-first-served disambiguation tool for lazy-evaluation of overlapping file-copy promises.

5.2.20 `files_auto_define`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of filenames to define classes if copied
Example:

```
body agent control
{
files_auto_define => { "/etc/syslog\.c.*", "/etc/passwd" };
}
```

Notes:

Classes are automatically defined by the files that are copied. The file is named according to the prefixed ‘canonization’ of the file name. Canonization means that non-identifier characters are converted into underscores. Thus ‘`/etc/passwd`’ would canonize to ‘`_etc_passwd`’. The prefix ‘`auto_`’ is added to clarify the origin of the class. Thus in the example the copying of ‘`/etc/passwd`’ would lead to the class ‘`auto__etc_passwd`’ being defined automatically.

5.2.21 `fullencryption`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: Full encryption mode in server connections, includes directory listings
Example:

```
body agent control
{
fullencryption => "true";
}
```

Notes:

If true this encrypts all levels of the queries to the server during file transfers. The default is to not encrypt all aspects, since this can slow down transfer and basically only contributes to global warming for most users.

5.2.22 `hostnamekeys`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false label ppkeys by hostname not IP address

Example:

```
body server control
{
hostnamekeys => "true";
}
```

Notes:

Client side choice to base key associations on host names rather than IP address. This is useful for hosts with dynamic addresses.

5.2.23 `ifelapsed`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Global default for time that must elapse before promise will be rechecked

Example:

```
#local

body action example
{
ifelapsed   => "120";
expireafter => "240";
}
```

```
# global

body agent control
{
ifelapsed   => "120";
}
```

Notes:

This overrides the global settings. Promises which take a long time to verify should usually be protected with a long value for this parameter. This serves as a resource 'spam' protection. A cfengine check could easily run every 5 minutes provided resource intensive operations are not performed on every run. Using time classes like `Hr12` etc., is one part of this strategy; using `ifelapsed` is another which is not tied to a specific time.

5.2.24 `inform`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false set inform level default

Example:

```
body agent control
{
inform => "true";
}
```

Notes:

Equivalent to or overrides the command line option '`-I`'. Sets the default output level 'permanently' within the class context indicated.

5.2.25 `lastseen`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false record last observed time for all client-server connections (true)
Example:

```
reports:

  "Comment"

    lastseen => "10";
```

Notes:

After this time (hours) has passed, references to the external peer will be purged from this host's database.

5.2.26 `lastseenexpireafter`

Type: int
Allowed input range: `0,99999999999`
Synopsis: Time in days after which non-responsive last-seen hosts are purged
Example:

Notes:

```
body common control
{
lastseenexpireafter => "72";
}
```

5.2.27 `mountfilesystems`

Type: (menu option)
Allowed input range:

```
                true
                false
                yes
                no
                on
                off
```

Synopsis: true/false mount any filesystems promised

Example:

```
body agent control
{
mountfilesystems => "true";
}
```

Notes:

Issues the generic command to mount file systems defined in the file system table.

5.2.28 `nonalphanumfiles`

Type: (menu option)

Allowed input range:

```
                true
                false
                yes
                no
                on
                off
```

Synopsis: true/false warn about filenames with no alphanumeric content

Example:

```
body agent control
{
nonalphanumericfiles => "true";
}
```

Notes:

This test is applied in all recursive/depth searches.

5.2.29 `repchar`

Type: string

Allowed input range: .

Synopsis: The character used to canonize pathnames in the file repository

Example:

```
body agent control
{
repchar => "_";
}
```

Notes:

5.2.30 `default_repository`

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Path to the default file repository

Example:

```
body agent control
{
default_respository => "/var/cfengine/repository";
}
```

Notes:

If defined the default repository is the location where versions of files altered by cfengine are stored. This should be understood in relation to the policy for '`backup`' in copying, editing etc. If the backups are time-stamped, this becomes effective a version control repository.

5.2.31 `secureinput`

Type: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
```

```
               on
               off
```

Synopsis: true/false check whether input files are writable by unauthorized users

Example:

```
body agent control
{
secureinput => "true";
}
```

Notes:

If this is set, the agent will not accept an input file that is not owned by a privileged user.

5.2.32 `sensiblecount`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum number of files a mounted filesystem is expected to have

Example:

```
body agent control
{
sensiblecount => "20";
}
```

Notes:

5.2.33 `sensiblesize`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum number of bytes a mounted filesystem is expected to have

Example:

```
body agent control
{
sensiblesize => "20K";
}
```

Notes:

5.2.34 skipidentify

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: Do not send IP/name during server connection because address resolution is broken
Example:

```
body agent control
{
skipidentify => "true";
}
```

Notes:

Hosts that are not registered in DNS cannot supply reasonable credentials for a secondary confirmation of their identity to a cfengine server. This causes the agent to ignore its missing DNS credentials.

5.2.35 suspiciousnames

Type: slist
Allowed input range: `List of names to warn about if found during any file search`
Synopsis: (null)
Example:

```
body agent control
{
suspiciousnames => { ".mo", "lrk3", "rootkit" };
}
```

Notes:

If cfengine sees these names during recursive (depth) file searches it will warn about them.

5.2.36 `syslog`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false switches on output to syslog at the inform level

Example:

```
body agent example
{
syslog => "true";
}
```

Notes:

5.2.37 `timezone`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of allowed timezones this machine must comply with

Example:

```
body agent control
{
timezone => { "MET", "CET", "GMT+1" };
}
```

Notes:

5.2.38 `default_timeout`

Type: int
Allowed input range: `0,99999999999`
Synopsis: Maximum time a network connection should attempt to connect
Example:

```
body agent control
{
default_timeout => "10";
}
```

Notes:

The time is in seconds. It is not a guaranteed number, since it depends on system behaviour. under Linux, the kernel version plays a role, since not all system calls seem to respect the signals.

5.2.39 `verbose`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false switches on verbose standard output
Example:

```
body agent control
{
verbose => "true";
}
```

Notes:

5.3 server control promises

```
body server control

{
allowconnects         => { "127.0.0.1" , "::1" ,  ".*.example.org" };
allowallconnects      => { "127.0.0.1" , "::1" ,  ".*.example.org" };

# Uncomment me under controlled circumstances
#trustkeysfrom         => { "127.0.0.1" , "::1" ,  ".*.example.org" };
}
```

Server controls are mainly about determining access policy for the connection protocol: i.e. access to the server itself. Access to specific files must be granted in addition.

5.3.1 cfruncommand

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Path to the cf-agent command or cf-execd wrapper for remote execution

Example:

```
body server control

{
#..
cfruncommand => "/var/cfengine/bin/cf-agent";
}
```

Notes:

It is normal for this to point to the location of `cf-agent` but it could also point to the `cf-execd`, or even another program at your own risk.

5.3.2 maxconnections

Type: int

Allowed input range: `0,99999999999`

Synopsis: Maximum number of connections that will be accepted by cf-serverd

Example:

```
# client side

body agent control
{
maxconnections => "1000";
}

# server side

body server control
{
maxconnections => "1000";
}
```

Notes:

Watch out for kernel limitations for maximum numbers of open file descriptors which can limit this.

5.3.3 `denybadclocks`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false accept connections from hosts with clocks that are out of sync

Example:

```
body server control
{
#..
denybadclocks => "true";
}
```

Notes:

A possible form of attack on the fileserver is to request files based on time by setting the clocks incorrectly. This option prevents connections from clients whose clocks are drifting too far from the server clock. This serves as a warning about clock asynchronization and also a protection against Denial of Service attempts based on clock corruption.

5.3.4 allowconnects

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames that may connect to the server port

Example:

```
allowconnects         => {
                         "127.0.0.1" ,
                         "::1",
                         "2001.10.*" ,
                         "host.domain.tld",
                         "host[0-9]+\.domain.com"
                         };
```

Notes:

If a client's identity matches an entry in this list it is granted to permission to send data to the server port. Clients who are not in this list may not connect or send data to the server.

5.3.5 denyconnects

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames that may NOT connect to the server port

Example:

```
body server control
{
denyconnects => { "badhost.domain.evil" };
}
```

Notes:

Hosts or IP addresses that are explicitly denied access. This should only be used in special circumstances. One should never grant generic access to everything and then deny special cases. Since the default server behaviour is to grant no access to anything, this list is unnecessary unless you have already granted access to some set of hosts using a generic pattern, to which you intend to make an exception.

5.3.6 `allowallconnects`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames that may have more than one connection to the server port

Example:

```
allowallconnects      => {
                         "127.0.0.1" ,
                         "::1",
                         "2001.10.*" ,
                         "host.domain.tld",
                         "host[0-9]+\.domain.com"
                         };
```

Notes:

This list of regular expressions matches hosts that are allowed to connect an umlimited number of times up to the maximum connection limit. Without this, a host may only connect once (which is a very strong constraint, as the host must wait for the TCP FIN_WAIT to expire before reconnction can be attempted).

In cfengine 2 this corresponds to `AllowMultipleConnectionsFrom`.

5.3.7 `trustkeysfrom`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames from whom we accept public keys on trust

Example:

```
body server control
{
trustkeysfrom => {"hosts.*", "192.168.*"};
}
```

Notes:

If connecting clients' public keys have not already been trusted, this allows us to say 'yes' to accepting the keys on trust. Normally this should be an empty list except in controlled circumstances.

5.3.8 `allowusers`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of usernames who may execute requests from this server
Example:

```
allowusers => { "cfengine", "root" };
```

Notes:

The usernames listed in this list are those asserted as public key identities during client-server connections. These may or may not correspond to system identities on the server-side system.

5.3.9 `dynamicaddresses`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of IPs or hostnames for which the IP/name binding is expected to change
Example:

```
body server control
{
dynamicaddresses => { "dhcp_.*" };
}
```

Notes:

The addresses or hostnames here are expected to have non-permanent address-name bindings, we must therefor work harder to determine whether hosts credentials are trusted by looking for existing public keys in files that do not match the current hostname or IP.

5.3.10 `skipverify`

Type: slist
Allowed input range: (arbitrary string)

Synopsis: List of IPs or hostnames for which we expect no DNS binding and cannot verify
Example:

```
body server control
{
skipverify => { "special_host.*", "192.168.*" };
}
```

Notes:

Server side decision to ignore requirements of DNS identity confirmation.

5.3.11 `logallconnections`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false causes the server to log all new connections to syslog
Example:

```
body server control
{
logallconnections => "true";
}
```

Notes:

If set, the server will record connection attempts in syslog.

5.3.12 `logencryptedtransfers`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false log all successful transfers required to be encrypted
Example:

```
body server control
{
logencryptedtransfers => "true";
}
```

Notes:

If true the server will log all transfers of files which the server requires to encrypted in order to grant access (see `ifencrypted`) to syslog. These files are deemed to be particularly sensitive.

5.3.13 `hostnamekeys`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false store keys using hostname lookup instead of IP addresses
Example:

```
body server control
{
hostnamekeys => "true";
}
```

Notes:

Client side choice to base key associations on host names rather than IP address. This is useful for hosts with dynamic addresses.

5.3.14 auditing

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false activate auditing of server connections

Example:

```
body agent control
{
auditing  => "true";
}
```

Notes:

If this is set, cfengine will perform auditing on promises in the current configuration. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in cfengine 2.

5.3.15 bindtointerface

Type: string

Allowed input range: (arbitrary string)

Synopsis: IP of the interface to which the server should bind on multi-homed hosts

Example:

```
bindtointerface => "192.168.1.1";
```

Notes:

On multi-homed hosts, the server and client can bind to a specific interface for server traffic. The IP address of the interface must be given as the argument, not the device name.

5.3.16 serverfacility

Type: (menu option)

Allowed input range:

```
LOG_USER
LOG_DAEMON
LOG_LOCAL0
LOG_LOCAL1
LOG_LOCAL2
LOG_LOCAL3
LOG_LOCAL4
LOG_LOCAL5
LOG_LOCAL6
LOG_LOCAL7
```

Synopsis: Menu option for syslog facility level

Example:

```
body server control
{
serverfacility => "LOG_USER";
}
```

Notes:

See syslog notes.

5.4 monitor control promises

```
body monitor control()
   {
   #version => "1.2.3.4";

   threshold => "0.3";
   forgetrate => "0.7";
   histograms => "true";
   tcpdump => "false";
   tcpdumpcommand => "/usr/sbin/tcpdump -i eth1 -n -t -v";

   }
```

The system defaults will be sufficient for most users. This configurability potential will be a key to developing the integrated monitoring capabilities of cfengine however.

5.4.1 `forgetrate`

Type: real

Allowed input range: `0,1`

Synopsis: Decimal fraction [0,1] weighting of new values over old in 2d-average computation

Example:

```
body monitor control
{
threshold => "0.3";
forgetrate => "0.7";
histograms => "true";
}
```

Notes:

Configurable settings for the machine-learning algorithm that tracks system behaviour. This is only for expert users. This parameter effectively determines (together with the monitoring rate) how quickly cfengine forgets its previous history.

5.4.2 `monitorfacility`

Type: (menu option)

Allowed input range:

```
                LOG_USER
                LOG_DAEMON
                LOG_LOCAL0
                LOG_LOCAL1
                LOG_LOCAL2
                LOG_LOCAL3
                LOG_LOCAL4
                LOG_LOCAL5
                LOG_LOCAL6
                LOG_LOCAL7
```

Synopsis: Menu option for syslog facility

Example:

```
body monitor control
{
monitorfacility => "LOG_USER";
}
```

Notes:

See notes for syslog.

5.4.3 `histograms`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false store signal histogram data

Example:

```
body monitor control
{
histograms => "true";
}
```

Notes:

This is like the '`-H`' option to `cfenvd` in cfengine 2. It causes cfengine to learn the conformally transformed distributions of fluctuations about the mean.

5.4.4 `tcpdump`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false use tcpdump if found

Example:

```
body monitor control
{
tcpdump => "true";
}
```

Notes:

Interface with TCP stream if possible.

5.4.5 tcpdumpcommand

Type: string
Allowed input range: `[cC]:\\.*|/.*`
Synopsis: Path to the tcpdump command on this system
Example:

```
body monitor control
{
tcpdumpcommand => "/usr/sbin/tcpdump -i eth1";
}
```

Notes:

If this is defined, the monitor will try to interface with the TCP stream and monitor generic package categories for anomalies.

5.5 runagent control promises

```
body runagent control
{
# default port is 5308

hosts => { "127.0.0.1:5308", "eternity.iu.hio.no:80", "slogans.iu.hio.no" };

#output_to_file => "true";
}
```

The most important parameter here is the list of hosts that the agent will poll for connections. This is easily read in from a file list, however when doing so always have a stable input source that does not depend on the network (including a database or directory service) in any way: introducing such dependencies makes configuration brittle.

5.5.1 `hosts`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host or IP addresses to attempt connection with

Example:

```
body runagent control
{
network1::
  hosts => { "host1.example.org", "host2", "host3" };

network2::
  hosts => { "host1.example.com", "host2", "host3" };
}
```

Notes:

The complete list of contactable hosts. The values may be either numerical IP addresses or DNS names, optionally suffixed by a '`:`' and a port number. If no port number is given, the default cfengine port 5308 is assumed.

5.5.2 `port`

Type: int

Allowed input range: `1024,99999`

Synopsis: Default port for cfengine server

Example:

```
body copy_from example
{
portnumber => "5308";
}
```

Notes:

The standard or registered port number is tcp/5308. Cfengine does not presently use its registered udp port with the same number, but this could change in the future.

5.5.3 `force_ipv4`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false force use of ipv4 in connection
Example:

```
body copy_from example
{
force_ipv4 => "true";
}
```

Notes:

IPv6 should be harmless to most users unless you have a partially or misconfigured setup.

5.5.4 `trustkey`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false automatically accept all keys on trust from servers
Example:

```
body copy_from example
{
trustkey => "true";
}
```

Notes:

If the server's public key has not already been trusted, this allows us to accept the key in automated key-exchange.

Note that, as a simple security precaution, trustkey should normally be set to '`false`', to avoid key exchange with a server one is not one hundred percent sure about, though the risks for a client are rather low. On the server-side however, trust is often granted to many clients or to a whole network in which possibly unauthorized parties might be able to obtain an IP address, thus the trust issue is most important on the server side.

As soon as a public key has been exchanged, the trust option has no effect. A machine that has been trusted remains trusted until its key is manually revoked by a system administrator. Keys are stored in '`WORKDIR/ppkeys`'.

5.5.5 `encrypt`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false encrypt connections with servers
Example:

```
body copy_from example
{
servers  => { "remote-host.example.org" };
encrypt => "true";
}
```

Notes:

Client connections are encrypted with using a Blowfish randomly generated session key. The intial connection is encrypted using the public/private keys for the client and server hosts.

5.5.6 `background_children`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false parallelize connections to servers

Example:

```
body runagent control
{
background_children => "true";
}
```

Notes:

Causes the runagent to attempt parallelized connections to the servers.

5.5.7 `max_children`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Maximum number of simultaneous connections to attempt

Example:

```
body runagent control
{
max_children => "10";
}
```

Notes:

The maximum number of forked background processes allowed when parallelizing connections to servers.

5.5.8 `output_to_file`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether to send collected output to file(s)

Example:

```
body runagent control
{
output_to_file => "true";
}
```

Notes:

Filenames are chosen automatically and placed in the '`WORKDIR/outputs/`*`hostname`*`_runagent.out`'.

5.6 `executor` control promises

```
body executor control

{
splaytime  => "5";
mailto     => "cfengine@example.org";
mailfrom   => "cfengine@$(host).example.org";
smtpserver => "localhost";
schedule   => { "Min00_05", "Min30_35" }
}
```

These body settings determine the behaviour of `cf-execd`, including scheduling times and output capture to '`WORKDIR/outputs`' and relay via email. Note that the `splaytime` and `schedule` parameters are now coded here rather than (as previously) in the agent.

5.6.1 `splaytime`

Type: int

Allowed input range: `0,99999999999`

Synopsis: Time in minutes to splay this host based on its name hash

Example:

```
body executor control
{
splaytime => "2";
}
```

Notes:

A rough rule of thumb for scaling of small updates is set the splay time by:

splaytime = 1 + Number of clients / 50

5.6.2 `mailfrom`

Type: string

Allowed input range: `.*.*`

Synopsis: Email-address cfengine mail appears to come from

Example:

```
body executor control
{
mailfrom => "MrCfengine@example.org";
}
```

Notes:

5.6.3 `mailto`

Type: string

Allowed input range: `.*.*`

Synopsis: Email-address cfengine mail is sent to

Example:

```
body executor control
{
mailto => "cfengine_alias@example.org";
}
```

Notes:

The address to whom email is sent if an smtp host is configured.

5.6.4 smtpserver

Type: string
Allowed input range: `.*`
Synopsis: Name or IP of a willing smtp server for sending email
Example:

```
body executor control
{
smtpserver => "smtp.example.org";
}
```

Notes:

This should point to a standard port 25 server without encyption. If you are running secured or encrypted email then you should run a mail relay on localhost and point this to localhost.

5.6.5 mailmaxlines

Type: int
Allowed input range: `0,1000`
Synopsis: Maximum number of lines of output to send by email
Example:

```
body executor control
{
mailmaxlines => "100";
}
```

Notes:

This limit prevents anomalously large outputs from clogging up a system administrator's mailbox. The output is truncated in the email report, but the complete original transcript is stored in 'WORKDIR/outputs/*' where it can be viewed on demand. A reference to the appropriate file is given.

5.6.6 schedule

Type: slist

Allowed input range: (arbitrary string)

Synopsis: The class schedule for activating cf-execd

Example:

```
body executor control
{
schedule => { "Min00_05", "Min05_20", "Min30_35", "Min45_50" };
}
```

Notes:

The list should contain classes which are visible to the `cf-execd` daemon. In principle any defined class will cause the daemon to wake up and schedule the execution of the `cf-agent`.

5.6.7 executorfacility

Type: (menu option)

Allowed input range:

```
            LOG_USER
            LOG_DAEMON
            LOG_LOCAL0
            LOG_LOCAL1
            LOG_LOCAL2
            LOG_LOCAL3
            LOG_LOCAL4
            LOG_LOCAL5
            LOG_LOCAL6
            LOG_LOCAL7
```

Synopsis: Menu option for syslog facility level

Example:

```
body executor control
{
executorfacility => "LOG_USER";
}
```

Notes:

See the syslog manual pages.

5.6.8 exec_command

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: The full path and command to the executable run by default (overriding builtin)

Example:

```
exec_command => "$(sys.workdir)/bin/cf-agent -f failsafe.cf && $(sys.workdir)/bin/cf-agent";
```

Notes:

The command is run in a shell encapsulation so pipes and shell symbols may be used if desired. Unlike, cfengine 2, cfengine 3 does not automatically run a separate update sequence before its normal run. This can be handled using the approach in the example above.

5.7 knowledge control promises

```
body knowledge control

{
build_directory => ".";

sql_database => "my_knowledge";
sql_owner => "db_user";
sql_passwd => ""; # No passwd
sql_type => "mysql";

query_output => "html";

style_sheet => "http://www.example.org/css/style.css";
html_banner =>
               "
               <ul>
                <li>Item 1
                <li>Item 2
               </ul>
               ";
}
```

These parameters control the way in which knowledge data are stored and retrieved from a relational database and the output format of the queries.

5.7.1 `id_prefix`

Type: string
Allowed input range: `.*`
Synopsis: The LTM identifier prefix used to label topic maps (used for disambiguation in merging)
Example:

```
body knowledge control
{
id_prefix => "unique_prefix";
}
```

Notes:

Use to disambiguate indentifiers for a successful merging of topic maps, especially in Linear Topic Map (LTM) format using third party tools such as Ontopia's Omnigator.

5.7.2 `build_directory`

Type: string
Allowed input range: `.*`
Synopsis: The directory in which to generate output files

Example:

```
body knowledge control

{
#..
build_directory => "/tmp/builddir";
}

body reporter control

{
#..
build_directory => "/tmp/builddir";
}
```

Notes:

The directory where all auto-generated textual output is placed by `cf-know`. This includes manual generation, ontology and topic map data.

5.7.3 sql_type

Type: (menu option)
Allowed input range:

```
               mysql
               postgress
```

Synopsis: Menu option for supported database type
Example:

```
body knowledge control
{
sql_type => "mysql";
}
```

Notes:

5.7.4 sql_database

Type: string

Allowed input range: (arbitrary string)
Synopsis: Name of database used for the topic map
Example:

```
body knowledge control
{
sql_database => "cfengine_knowledge_db";
}
```

Notes:

The name of an SQL database for caching knowledge.

5.7.5 `sql_owner`

Type: string
Allowed input range: (arbitrary string)
Synopsis: User id of sql database user
Example:

```
body knowledge control
{
sql_owner => "db_owner";
}
```

Notes:

Part of the credentials for opening the database. This depends on the type of database.

5.7.6 `sql_passwd`

Type: string
Allowed input range: (arbitrary string)
Synopsis: Embedded password for accessing sql database
Example:

```
body knowledge control
{
sql_passwd => "";
}
```

Notes:

Part of the credentials for connecting to the database server. This is system dependent. If the server host is localhost a password might not be required.

5.7.7 `sql_server`

Type: string
Allowed input range: (arbitrary string)
Synopsis: Name or IP of database server (or localhost)
Example:

```
body knowledge control
{
sql_server => "localhost";
}
```

Notes:

The host name of IP address of the server. The default is to look on the localhost.

5.7.8 `query_output`

Type: (menu option)
Allowed input range:

```
               html
               text
```

Synopsis: Menu option for generated output format
Example:

```
body knowledge control
{
query_output => "html";
}
```

Notes:

5.7.9 `query_engine`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a dynamic web-page used to accept and drive queries in a browser

Example:

```
body knowledge control
{
query_engine => "http://www.example.org/script.php";
}

body reporter control
{
query_engine => "http://www.example.org/script.pl";
}
```

Notes:

When displaying topic maps in HTML format, `cf-know` will render each topic as a link to this URL with the new topic as an argument. Thus it is possible to make a dynamic web query by embedding cfengine in the web page as system call and passing the argument to it.

5.7.10 `style_sheet`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a style-sheet to be used in rendering html output (added to headers)

Example:

```
body knowledge control
{
style_sheet => "http://www.example.org/css/sheet.css";
}

body reporter control
{
style_sheet => "http://www.example.org/css/sheet.css";
}
```

Notes:

For formatting the HTML generated output of `cf-know`.

5.7.11 `html_banner`

Type: string
Allowed input range: (arbitrary string)
Synopsis: HTML code for a banner to be added to rendered in html after the header
Example:

```
body knowledge control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}

body reporter control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}
```

Notes:

This content is cited when generating HTML output from the knowledge agent.

5.7.12 `html_footer`

Type: string
Allowed input range: (arbitrary string)
Synopsis: HTML code for a page footer to be added to rendered in html before the end body tag
Example:

```
body reporter control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}

body knowledge control
{
```

```
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}
```

Notes:

This allows us to cite HTML code for formatting reports generated by the reporting and knowledge agents.

5.7.13 `graph_output`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false generate png visualization of topic map if possible (requires lib)
Example:

```
body knowledge control

{
# fix/override -g option
graph_output => "true";
}
```

Notes:

Equivalent to the use of the '`-g`' option for `cf-know`.

5.7.14 `graph_directory`

Type: string
Allowed input range: `[cC]:\\.*|/.*`
Synopsis: Path to directory where rendered .png files will be created
Example:

```
body knowledge control
{
graph_directory => "/tmp/output";
}
```

Notes:

A separate location where the potentially large number of '`.png`' visualizations of a knowledge representation are pre-compiled. This feature only works if the necessary graphics libraries are present.

5.7.15 `generate_manual`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false generate texinfo manual page skeleton for this version

Example:

```
body knowledge control
{
generate_manual => "true";
}
```

Notes:

Auto-creates a manual based on the self-documented code. As the promise syntax is extended the manual self-heals. The resulting manual is generated in Texinfo format, from which all other formats can be generated.

5.7.16 `manual_source_directory`

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Path to directory where raw text about manual topics is found (defaults to build_directory)

Example:

```
body knowledge control
{
manual_source => "/path/cfengine_manual_commentary";
}
```

Notes:

This is used in the self-healing documentation. The directory points to a location where the Texinfo sources for per-section commentary is maintained.

5.8 `reporter` control promises

```
body reporter control
{
reports => { "performance", "last_seen", "monitor_history" };
build_directory => "/tmp/nerves";
report_output => "html";
}
```

Determines a list of reports to write into the build directory. The format may be in text, html or xml format. The reporter agent `cf-report` replaces both `cfshow` and `cfenvgraph`. It no longer produces output to the console.

5.8.1 `reports`

Type: (option list)

Allowed input range:

```
audit
performance
all_locks
active_locks
hashes
classes
last_seen
monitor_now
monitor_history
```

Synopsis: A list of reports to generate

Example:

```
body reporter control
{
reports => { "performance", "classes"  };
}
```

Notes:

A list of report types that can be generated by this agent.

5.8.2 `report_output`

Type: (menu option)

Allowed input range:

```
               html
               text
               xml
```

Synopsis: Menu option for generated output format

Example:

```
body reporter control
{
report_output => "html";
}
```

Notes:

Sets the output format of embedded database reports.

5.8.3 `build_directory`

Type: string

Allowed input range: `.*`

Synopsis: The directory in which to generate output files

Example:

```
body knowledge control
```

```
{
#..
build_directory => "/tmp/builddir";
}

body reporter control

{
#..
build_directory => "/tmp/builddir";
}
```

Notes:

The directory where all auto-generated textual output is placed by `cf-know`. This includes manual generation, ontology and topic map data.

5.8.4 `auto_scaling`

Type: (menu option)
Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether to auto-scale graph output to optimize use of space
Example:

```
body reporter control
{
auto_scaling => "true";
}
```

Notes:

Automatic scaling is the default.

5.8.5 `error_bars`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether to generate error bars on graph output

Example:

```
body reporter control
{
error_bars => "true";
}
```

Notes:

The default is to produce error bars. Without error bars from cfengine's machine learning data there is no way to assess the significance of an observation about the system, i.e. whether it is normal or anomalous.

5.8.6 `time_stamps`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether to generate timestamps on the output directory

Example:

```
body reporter control
{
time_stamps => "true";
}
```

Notes:

This option is only necessary with the default build directory. This can be used to keep snapshots of the system but it will result in a lot of storage be consumed. For most purposes cfengine is programmed to forget the past at a predictable rate and there is no need to override this.

5.8.7 `query_engine`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a dynamic web-page used to accept and drive queries in a browser

Example:

```
body knowledge control
{
query_engine => "http://www.example.org/script.php";
}

body reporter control
{
query_engine => "http://www.example.org/script.pl";
}
```

Notes:

When displaying topic maps in HTML format, `cf-know` will render each topic as a link to this URL with the new topic as an argument. Thus it is possible to make a dynamic web query by embedding cfengine in the web page as system call and passing the argument to it.

5.8.8 `style_sheet`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of a style-sheet to be used in rendering html output (added to headers)

Example:

```
body knowledge control
{
style_sheet => "http://www.example.org/css/sheet.css";
}
```

```
body reporter control
{
style_sheet => "http://www.example.org/css/sheet.css";
}
```

Notes:

For formatting the HTML generated output of `cf-know`.

5.8.9 `html_banner`

Type: string

Allowed input range: (arbitrary string)

Synopsis: HTML code for a banner to be added to rendered in html after the header

Example:

```
body knowledge control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}

body reporter control
{
html_banner => "<img src=\"http://www.example.org/img/banner.png\">";
}
```

Notes:

This content is cited when generating HTML output from the knowledge agent.

5.8.10 `html_footer`

Type: string

Allowed input range: (arbitrary string)

Synopsis: HTML code for a page footer to be added to rendered in html before the end body tag

Example:

```
body reporter control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}

body knowledge control
{
html_footer => "
               <div id=\"footer\">Bottom of the page</div>
               ";
}
```

Notes:

This allows us to cite HTML code for formatting reports generated by the reporting and knowledge agents.

6 Bundles of common

```
bundle common globals
{
vars:

  "global_var" string => "value";

classes:

  "global_class" expression => "value";
}
```

Common bundles may only contain the promise types that are common to all bodies. Their main function is to define cross-component global definitions. Common bundles are observed by every agent, whereas the agent specific bundle types are ignored by components other than the intended recipient.

6.1 vars promises

Whereas most promise types are specific to a particular kind of interpretation that requires a typed interpreter (the bundle type), a number of promises can be made in any kind of bundle since they are of a generic input/output nature. These are listed below.

6.1.1 string

Type: string

Allowed input range: (arbitrary string)

Synopsis: A scalar string

Example:

```
vars:

 "xxx"    string => "Some literal string...";

 "yyy"    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

In cfengine previously lists were represented (as in the shell) using separted scalars, e.g. like the PATH variable. This design feature turned out to be an error of judgement which has resulted in much trouble. This is no longer supported in cfengine 3. By keeping lists an independent type many limitations have been removed.

6.1.2 `int`

Type: int
Allowed input range: `-99999999999,9999999999`
Synopsis: A scalar integer
Example:

```
vars:

 "scalar" int    => "16k";

 "ran"    int    => randomint(4,88);

 "dim_array" int =>  readstringarray("array_name","/etc/passwd","#[^\n]*",":",10,4000);
```

Notes:

Integer values may use suffices '`k`', '`K`', '`m`', '`M`', etc

'`k`'	The value multipled by 1000.
'`K`'	The value multipled by 1024.
'`m`'	The value multipled by 1000 * 1000.
'`M`'	The value multipled by 1024 * 1024.
'`g`'	The value multipled by 1000 * 1000 * 1000.
'`G`'	The value multipled by 1024 * 1024 * 1024.
'`%`'	A percentage between 1 and 100 - mainly for use in a storage context.

The value '`inf`' may also be used to represent an umlimited positive value.

6.1.3 `real`

Type: real
Allowed input range: `-9.99999E100,9.99999E100`
Synopsis: A scalar real number
Example:

```
vars:
 "scalar" real   => "0.5";
```

Notes:

Real numbers are not used in many places in cfengine, but they are useful for representing probabilties and performance data.

6.1.4 slist

Type: slist
Allowed input range: (arbitrary string)
Synopsis: A list of scalar strings
Example:

```
vars:

 "xxx"    slist  => {  "literal1",  "literal2" };

 "yyy"    slist  => {
                    readstringlist(
                                  "/home/mark/tmp/testlist",
                                  "#[a-zA-Z0-9 ]*",
                                  "[^a-zA-Z0-9]",
                                  15,
                                  4000
                                  )
                    };

 "zzz"    slist  => { readstringlist("/home/mark/tmp/testlist2","#[^\n]*",",",5,4000) };
```

Notes:

6.1.5 ilist

Type: ilist
Allowed input range: -99999999999,9999999999
Synopsis: A list of integers
Example:

```
vars:

  "variable_id"

       ilist => { "10", "11", "12" };
```

Notes:

Integer lists are lists of strings that are expected to be treated as integers. The typing in cfengine is dynamic, so the variable types are interchangable.

6.1.6 `rlist`

Type: rlist

Allowed input range: `-9.99999E100,9.99999E100`

Synopsis: A list of real numbers

Example:

```
vars:

  "varid" rlist => { "0.1", "0.2", "0.3" };
```

Notes:

6.1.7 `policy`

Type: (menu option)

Allowed input range:

```
               free
               overridable
               constant
```

Synopsis: The policy for (dis)allowing redefinition of variables

Example:

```
vars:

  "varid" string => "value..."
          policy => "constant";
```

Notes:

Variables can either be allowed to change their value dynamically (be redefined) or they can be constant. The use of private variable spaces in cfengine 3 makes it unlikely that variable redefinition would be necessary in cfengine 3.

6.2 classes promises

Whereas most promise types are specific to a particular kind of interpretation that requires a typed interpreter (the bundle type), a number of promises can be made in any kind of bundle since they are of a generic input/output nature. These are listed below.

6.2.1 or

Type: clist
Allowed input range: `[a-zA-Z0-9_!&|.()]+`
Synopsis: Combine class sources with inclusive OR
Example:

```
classes:

    "compound_test"

      or => { classmatch("linux_x86_64_2_6_22.*"), "suse_10_3" };
```

Notes:

A useful construction for writing expressions that contain special functions.

6.2.2 and

Type: clist
Allowed input range: `[a-zA-Z0-9_!&|.()]+`
Synopsis: Combine class sources with AND
Example:

```
classes:

  "compound_class" and => { classmatch("host[0-9].*"), "Monday", "Hr02" };
```

Notes:

If an expression contains a mixture of different object types that need to be ANDed together, this list form is more convenient than providing an expression.

6.2.3 `xor`

Type: clist
Allowed input range: `[a-zA-Z0-9_!&|.()]+`
Synopsis: Combine class sources with XOR
Example:

```
classes:

 "another_global" xor => { "any", "linux", "solaris"};
```

Notes:

6.2.4 `dist`

Type: rlist
Allowed input range: `-9.99999E100,9.99999E100`
Synopsis: Generate a probabilistic class distribution (strategy in cfengine 2)
Example:

```
classes:

  "my_dist"

    dist => { "10", "20", "40", "50" };
```

Notes:

Assign one exclusive class randomly weighted on a probability distribution. This will generate the following classes:

```
my_dist    (always)
my_dist_10 (10/120 of the time)
my_dist_20 (20/120 of the time)
my_dist_40 (40/120 of the time)
my_dist_50 (50/120 of the time)
```

This was previous called a ‘strategy’ in cfengine 2.

6.2.5 expression

Type: class
Allowed input range: `[a-zA-Z0-9_!&|.()]+`
Synopsis: Evaluate string expression of classes in normal form
Example:

```
classes:

  "class_name" expression => "solaris|(linux.specialclass)";
```

Notes:

A way of aliasing class combinations.

6.2.6 not

Type: class
Allowed input range: `[a-zA-Z0-9_!&|.()]+`
Synopsis: Evaluate the negation of string expression in normal form
Example:

```
classes:

   "others"  not => "linux|solaris";
```

Notes:

In file editing, this negates the effect of the promiser-pattern regular expression.

6.3 reports promises

Whereas most promise types are specific to a particular kind of interpretation that requires a typed interpreter (the bundle type), a number of promises can be made in any kind of bundle since they are of a generic input/output nature. These are listed below.

6.3.1 lastseen

Type: int

Allowed input range: `0,99999999999`
Synopsis: Integer time threshold in hours since current peers were last seen, report absence
Example:

```
reports:

  "Comment"

    lastseen => "10";
```

Notes:

After this time (hours) has passed, references to the external peer will be purged from this host's database.

6.3.2 `intermittency`

Type: real
Allowed input range: `0,1`
Synopsis: Real number threshold [0,1] of intermittency about current peers, report above
Example:

```
reports:

  "Comment"

    intermittency => "0.5";
```

Notes:

Report on cfengine peers in the neighbourhood watch whose observed irregularity of connection exceeds 0.5 scaled entropy units, meaning that they show an erratic pattern of connection.

6.3.3 `showstate`

Type: slist
Allowed input range: (arbitrary string)
Synopsis: List of services about which status reports should be reported to standard output
Example:

```
reports:

  "Comment"

    showstate => {"www_in", "ssh_out", "otherprocs" };
```

Notes:

The basic list of services is:

'users'	Users logged in
'rootprocs'	Privileged system processes
'otherprocs'	Non-privileged process
'diskfree'	Free disk on / partition
'loadavg'	% kernel load utilization
'netbiosns_in'	netbios name lookups (in)
'netbiosns_out'	netbios name lookups (out)
'netbiosdgm_in'	netbios name datagrams (in)
'netbiosdgm_out'	netbios name datagrams (out)
'netbiosssn_in'	netbios name sessions (in)
'netbiosssn_out'	netbios name sessions (out)
'irc_in'	IRC connections (in)
'irc_out'	IRC connections (out)
'cfengine_in'	cfengine connections (in)
'cfengine_out'	cfengine connections (out)
'nfsd_in'	nfs connections (in)
'nfsd_out'	nfs connections (out)
'smtp_in'	smtp connections (in)

'smtp_out'	smtp connections (out)
'www_in'	www connections (in)
'www_out'	www connections (out)
'ftp_in'	ftp connections (in)
'ftp_out'	ftp connections (out)
'ssh_in'	ssh connections (in)
'ssh_out'	ssh connections (out)
'wwws_in'	wwws connections (in)
'wwws_out'	wwws connections (out)
'icmp_in'	ICMP packets (in)
'icmp_out'	ICMP packets (out)
'udp_in'	UDP dgrams (in)
'udp_out'	UDP dgrams (out)
'dns_in'	DNS requests (in)
'dns_out'	DNS requests (out)
'tcpsyn_in'	TCP sessions (in)
'tcpsyn_out'	TCP sessions (out)
'tcpack_in'	TCP acks (in)
'tcpack_out'	TCP acks (out)
'tcpfin_in'	TCP finish (in)
'tcpfin_out'	TCP finish (out)
'tcpmisc_in'	TCP misc (in)
'tcpmisc_out'	TCP misc (out)
'webaccess'	Webserver hits
'weberrors'	Webserver errors
'syslog'	New log entries (Syslog)

‘messages’	New log entries (messages)
‘temp0’	CPU Temperature 0
‘temp1’	CPU Temperature 1
‘temp2’	CPU Temperature 2
‘temp3’	CPU Temperature 3
‘cpu’	%CPU utilization (all)
‘cpu0’	%CPU utilization 0
‘cpu1’	%CPU utilization 1
‘cpu2’	%CPU utilization 2
‘cpu3’	%CPU utilization 3

6.3.4 printfile (compound body)

Type: (ext body)

‘number_of_lines’

Type: int

Allowed input range: `0,99999999999`

Synopsis: Integer maximum number of lines to print from selected file

Example:

```
body printfile example
{
number_of_lines => "10";
}
```

Notes:

‘file_to_print’

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Path name to the file that is to be sent to standard output

Example:

```
body printfile example
{
file_to_print   => "/etc/motd";
```

```
           number_of_lines => "10";
           }
```

Notes:

Include part of a file in a report.

6.3.5 `friend_pattern`

Type: string
Allowed input range: (arbitrary string)
Synopsis: Regular expression to keep selected hosts from the friends report list
Example:

```
reports:

  linux::

   "Friend status report"

          lastseen => "0"
     friend_patten => "host1|host2|.*\.domain\.tld";
```

Notes:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/bodypart_fr
""
```

6.4 `*` promises in '`agent`'

Most promise bodies belong to one and only one type of promise. The generic '*' promises bodies can be added to any promise type in `cf-agent`, hence the star which means 'any'.

The body attributes described below can be added to any promise rule in the agent. These promises address matters of a completely general nature – how cfengine behaves as it attempts to keep a promise, comments about the promises etc.

```
files:
   "/etc/passwd" -> "Security team"

      perms  => p("644"),
      action => justcheck,
      comment => "This was decided in internal procedures XYZ123";
```

6.4.1 action (compound body)

Type: (ext body)

'action_policy'

Type: (menu option)

Allowed input range:

```
fix
warn
nop
```

Synopsis: Whether to repair or report about non-kept promises

Example:

The following example shows a simple use of transaction control, causing the promise to be verified as a separate background process.

```
body action background

{
action_policy => "warn";
}
```

Notes:

The action settings allow general transaction control to be implemented on promise verification. Action bodies place limits on how often to verify the promise and what classes to raise in the case that the promise can or cannot be kept.

Note that actions can be added to sub-bundles like methods and editing bundles, and that promises within these do not inherit action settings at higher levels. Thus, in the following example there are two levels of action setting:

```
###########################################################
#
# Warn if line matched
#
```

```
########################################################

body common control

{
bundlesequence  => { "testbundle" };
}

########################################################

bundle agent testbundle

{
files:

  "/var/cfengine/inputs/.*"

       edit_line => DeleteLinesMatching(".*cfenvd.*"),
       action => WarnOnly;
}

########################################################

bundle edit_line DeleteLinesMatching(regex)
  {
  delete_lines:

    "$(regex)" action => WarnOnly;

  }

########################################################

body action WarnOnly
{
action_policy => "warn";
}
```

The `action` setting for the `files` promise means that file edits will not be committed to disk, only warned about. This is a master-level promise that overrides anything that happens during the editing. The `action` setting for the edit bundle means that the internal memory modelling of the file will only warn about changes rather than committing them to the memory model. This makes little difference to the end result, but it means that cfengine will report

```
Need to delete line - ... - but only a warning was promised
```

Instead of

```
Deleting the prpomised line ...
Need to save file - but only a warning was promised
```

In either case, no changes will be made to the disk, but the messages given by `cf-agent` will differ.

'`ifelapsed`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Number of minutes before next allowed assessment of promise

Example:

```
#local

body action example
{
ifelapsed   => "120";
expireafter => "240";
}

# global

body agent control
{
ifelapsed   => "120";
}
```

Notes:

This overrides the global settings. Promises which take a long time to verify should usually be protected with a long value for this parameter. This serves as a resource 'spam' protection. A cfengine check could easily run every 5 minutes provided resource intensive operations are not performed on every run. Using time classes like `Hr12` etc., is one part of this strategy; using `ifelapsed` is another which is not tied to a specific time.

'`expireafter`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Number of minutes before a repair action is interrupted and retried

Example:

```
body action example
{
ifelapsed   => "120";
expireafter => "240";
}
```

Notes:

The locking time after which cfengine will attempt to kill and restart its attempt to keep a promise.

'log_string'

Type: string

Allowed input range: (arbitrary string)

Synopsis: A message to be written to the log when the promise is verified

Example:

```
promise-type:

 "promiser"

   attr => "value",
   action => log_me("checked $(this.promiser)");

# ..

body action log_me(s)
{
log_string => "$(s)";
}
```

Notes:

'log_level'

Type: (menu option)

Allowed input range:

```
inform
verbose
error
log
```

Synopsis: The reporting level sent to syslog

Example:

```
body action example
{
log_level => "inform";
}
```

Notes:

Use this as an alternative to auditing to use the syslog mechanism to centralize or manage messaging from cfengine. A backup of these messages will still be kept in '`WORKDIR/outputs`' if you are using `cf-execd`.

'`audit`' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false switch for detailed audit records of this promise

Example:

```
body action example
{
# ...

audit => "true";
}
```

Notes:

If this is set, cfengine will perform auditing on this specific promise. This means that all details surrounding the verification of the current promise will be recorded in the audit database. The database may be inspected with `cf-report`, or `cfshow` in cfengine 2.

'`background`'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false switch for parallelizing the promise repair

Example:

```
body action example
{
background => "true";
}
```

Notes:

If possible, background the verification of the current promise. This is advantageous only if the verification might take a significant amount of time, e.g. in remote copying of filesystem/disk scans.

'`report_level`'

Type: (menu option)

Allowed input range:

```
inform
verbose
error
log
```

Synopsis: The reporting level for standard output

Example:

```
body action example
{
report_level => "verbose";
}
```

Notes:

In cfengine 2 one would say 'inform=true' or 'syslog=true', etc. This replaces these levels since they act as encapsulating super-sets.

'measurement_class'

Type: string

Allowed input range: (arbitrary string)

Synopsis: If set performance will be measured and recorded under this identifier

Example:

```
body action measure
{
measurement_class => "$(this.promiser) long job scan of /usr";
}
```

Notes:

By setting this string you switch on performance measurement of the promise, and also give it a name. The identifier forms a partial identity for optional performance scanning of promises.

ID:promise-type:promiser.

These can be seen identifying using cf-reports, e.g. in the generated file 'performance.html'.

6.4.2 classes (compound body)

Type: (ext body)

'promise_repaired'

Type: slist

Allowed input range: [a-zA-Z0-9_$.]+

Synopsis: A list of classes to be defined

Example:

```
body classes example
{
promise_repaired => { "change_happened" };
}
```

Notes:

If a promise is 'repaired' it means that a corrective action had to be taken to keep the promise.

'repair_failed'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: A list of classes to be defined

Example:

```
body classes example
{
repair_failed => { "unknown_error" };
}
```

Notes:

A promise could not be repaired because the corrective action failed for some reason.

'repair_denied'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: A list of classes to be defined

Example:

```
body classes example
{
repair_denied => { "permission_failure" };
}
```

Notes:

A promise could not be kept because access to a key resource was denied.

'repair_timeout'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: A list of classes to be defined

Example:

```
body classes example
{
repair_timeout => { "too_slow", "did_not_wait" };
}
```

Notes:

A promise maintenance repair timed-out waiting for some dependent resource.

'promise_kept'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: A list of classes to be defined

Example:

```
body classes example
{
promise_kept => { "success", "kaplah" };
}
```

Notes:

This class is set if no action was necessary by `cf-agent` because the promise concerned was aready kept without further action required.

'persist_time'

Type: int

Allowed input range: `0,99999999999`

Synopsis: A number of minutes the specified classes should remain active

Example:

```
body classes example
{
persist_time => "10";
}
```

Notes:

By default classes are ephemeral entities that disappear when `cf-agent` terminates. By setting a persistence time, they can last even when the agent is not running.

'`timer_policy`'

Type: (menu option)

Allowed input range:

```
          absolute
          reset
```

Synopsis: Whether a persistent class restarts its counter when rediscovered

Example:

```
body classes example
{
timer_policy => "reset";
}
```

Notes:

The in most cases resetting a timer will give a more honest appraisal of which classes are currently important, but if we want to activate a response of limited duration as a rare event then an asbolute time limit is useful.

6.4.3 `ifvarclass`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Extended classes ANDed with context

Example:

The generic example has the form:

```
     promise-type:

       "promiser"

          ifvarclass => "$(program)_running|($(program)_notfound&Hr12)";
```

A specific example would be:

```
bundle agent example
```

```
{
commands:

 any::

    "/bin/echo This is linux"

       ifvarclass => "linux";


    "/bin/echo This is solaris"

       ifvarclass => "solaris";

}
```

Notes:

This is an additional class expression that will be evaluated after the '*class*::' classes have selected promises. It is provided in order to enable a channel between variables and classes. The result is thus the logical AND of the ordinary classes and the variable classes.

This function is provided so that one can form expressions that link variables and classes, e.g.

```
# Check that all components are running

vars:

  "component" slist => { "cf-monitord", "cf-serverd" };

processes:

  "$(component)" restart_class => canonify("start_$(component)");

commands:

   "/var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
```

Notice that the function `canonify()` is provided to convert a general variable input into a string composed only of legal characters, using the same algorithm that cfengine uses.

6.4.4 comment

Type: string

Allowed input range: (arbitrary string)

Synopsis: Retained comment about this promise's real intention

Example:

```
comment => "This comment follows the data for reference ...",
```

Notes:

Comments written in code follow the program, they are not merely discarded. They appear in reports and error messages.

6.5 * promises in ‘edit_line’

Most promise bodies belong to one and only one type of promise. The generic ‘*’ promises bodies can be added to any promise type in `cf-agent`, hence the star which means ‘any’.

The body attributes described below can be added to any promise rule in the agent. These promises address matters of a completely general nature – how cfengine behaves as it attempts to keep a promise, comments about the promises etc.

```
files:

   "/etc/passwd" -> "Security team"

      perms  => p("644"),
      action => justcheck,
      comment => "This was decided in internal procedures XYZ123";
```

6.5.1 `select_region` (compound body)

Type: (ext body)

‘`select_start`’

Type: string

Allowed input range: `.*`

Synopsis: Regular expression matching start of edit region

Example:

```
body select_region example(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

Notes:

See also `select_end`. These delimiters mark out the region of a file to be edited. In the example, it is assumed that the file has section markes

```
[section 1]

lines.
lines...


[section 2]

lines ....
etc..
```

'`select_end`'

Type: string

Allowed input range: `.*`

Synopsis: Regular expression matches end of edit region from start

Example:

```
body select_region example(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

Notes:

See also `select_start`. These delimiters mark out the region of a file to be edited. In the example, it is assumed that the file has section markes

```
[section 1]

lines.
lines...
```

```
[section 2]

lines ....
etc..
```

7 Bundles of agent

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.1 `commands` promises in '`agent`'

```
commands:

  "/path/to/command args"

             args => "more args",
             contain => contain_body,
             module => true/false;
```

Command *containment* allows you to make a 'sandbox' around a command, to run it as a non-privileged user inside an isolated directory tree. Cfengine `modules` are commands that support a simple protocol (see below) in order to set additional variables and classes on execution from user defined code. Modules are intended for use as system probes rather than additional configuration promises.

In cfengine 3 commands and processes have been separated cleanly. Restarting of processes must be coded as a separate command. This stricter type separation will allow more careful conflict analysis to be carried out.

Output from commands executed here is quoted inline, but prefixed with the letter '`Q`' to distinguish it from other output, e.g. from `reports`.

Commands were called `shellcommands` in cfengine previously. Modules were

NOTE: a common mistake in using cfengine is to embed many shell commands instead of using the built-in functionality. Use of cfengine internals is preferred as it assures convergence and proper integrated checking. Extensive use of shell commands will make a cfengine execution very heavyweight like other management systems. To minimize the system cost of execution, always use cfengine internals.

```
bundle agent example

{
commands:

  "/bin/sleep 10"
     action  => background;

  "/bin/sleep"
     args => "20",
     action  => background;

}
```

7.1.1 args

Type: string

Allowed input range: (arbitrary string)

Synopsis: Alternative string of arguments for the command (concatenated with promiser string)

Example:

```
commands:

  "/bin/echo one"
```

```
   args => "two three";
```

Notes:

Sometimes it is convenient to separate the arguments to a command from the command itself. The final arguments are the concatenation with one space. So in the example above the command would be

```
 /bin/echo one two three
```

7.1.2 contain (compound body)

Type: (ext body)

'useshell' **Type**: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false embed the command in a shell environment (true)

Example:

```
body contain example
{
useshell => "true";
}
```

Notes:

The default is to use a shell when executing commands, but this has both resource and security consequences. A shell consumes an extra process and inherits environment variables, reads commands from files and performs other actions beyond the control of cfengine. If one does not need shell functionality such as piping through multiple commands then it is best to manage without it.

'umask' **Type**: (menu option)

Allowed input range:

```
0
77
22
27
72
```

Synopsis: The umask value for the child process

Example:

```
body contain example
{
umask => "077";
}
```

Notes:

Sets the internal umask for the process.

'exec_owner'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The user name or id under which to run the process

Example:

```
body contain example
{
exec_owner => "mysql_user";
}
```

Notes:

This is part of the restriction of privilege for child processes when running `cf-agent` as the root user, or a user with privileges.

'exec_group'

Type: string

Allowed input range: (arbitrary string)

Synopsis: The group name or id under which to run the process

Example:

```
body contain example
{
exec_group => "nogroup";
}
```

Notes:

This is part of the restriction of privilege for child processes when running `cf-agent` as the root user, or a user with privileges.

‘`exec_timeout`’

Type: int

Allowed input range: `1,3600`

Synopsis: Timeout in seconds for command completion

Example:

```
body contain example
{
exec_timeout => "30";
}
```

Notes:

Attempt to time-out after this number of seconds. This cannot be guaranteed as not all commands are willing to be interrupted in case of failure.

‘`chdir`’

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Directory for setting current/base directory for the process

Example:

```
body contain example

{
chdir => "/containment/directory";
}
```

Notes:

This command has the effect of placing the running command into a current working directory equal to the parameter given, i.e. it works like the '`cd`' shell command.

'`chroot`' **Type**: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Directory of root sandbox for process

Example:

```
body contain example

{
chroot => "/private/path";
}
```

Notes:

Sets the path of the directory that will be experienced as the top-most root directory for the process. In security parlance, this creates a 'sandbox' for the process.

'`preview`' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false preview command when running in dry-run mode (with -n)

Example:

```
body contain example
{
preview => "true";
}
```

Notes:

Previewing shell scripts during a dry-run is a potentially misleading activity. It should only be used on scripts that make no changes to the system. It is cfengine best practice to never write change-functionality into user-written scripts except as a last resort: cfengine can apply its safety checks to user defined scripts.

'no_output'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false discard all output from the command

Example:

```
body contain example
{
no_output => "true";
}
```

Notes:

This is equivalent to piping standard output and error to '/dev/null'.

7.1.3 module

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to expect the cfengine module protocol

Example:

```
commands:

   "/masterfiles/user_script"

     module => "true";
```

Notes:

If true, the module protocol is supported for this script, i.e. it is treated as a user module. A plug-in module may be written in any language, it can return any output you like, but lines which begin with a '+' sign are treated as classes to be defined (like '-D'), while lines which begin with a '-' sign are treated as classes to be undefined (like '-N'). Lines starting with '=' are variables/macros to be defined. Any other lines of output are cited by cfagent, so you should normally make your module completely silent. Here is an example module written in perl.

```
#!/usr/bin/perl
#
# module:myplugin
#

  # lots of computation....

if (special-condition)
   {
   print "+specialclass";
   }
```

Modules inherit the environment variables from cfagent and accept arguments, just as a regular command does.

```
#!/bin/sh
#
# module:myplugin
#

/bin/echo $*
```

`cf-agent` defines the classes as an environment variable so that programs have access to these. E.g. try the following module:

```
#!/usr/bin/perl

print "Decoding $ENV{CFALLCLASSES}\n";

@allclasses = split (":","$ENV{CFALLCLASSES}");

while ($c=shift(@allclasses))
  {
  $classes{$c} = 1;
  print "$c is set\n";
```

```
        }
```

Modules define variables in `cf-agent` by outputting strings of the form

```
   =variablename=value
```

These variables end up in a context which has the same name as the module. When the `$(allclasses)` variable becomes too large to manipulate conveniently, you can access the complete list of currently defined classes in the file '`/var/cfengine/state/allclasses`'.

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.2 `files` promises in '`agent`'

Files promises are an umbrella concept for all attributes of files. Operations fall basically into three categories: create, delete and edit.

```
files:

  "/path/file_object"

      perms => perms_body,
      ... ;
```

Prior to version 3, file promises were scattered into many different types such as `files`, `tidy`, `copy`, `links`, etc. File handling in cfengine 3 is more integrated than in cfengine 3. This helps both the logic and the efficiency of implementation. File handling is now more powerful, and uses regular expressions everywhere for pattern matching. The old 'wildcard/globbing' expressions '`*`' and '`?`' are deprecated, and everything is based consistently on Perl Compatible Regular Expressions where these are available. If PCRE is not available on the local system, POSIX extended regular expressions are used.

There is a natural ordering in file processing that obviates the need for the actionsequence. The trick of using multiple actionsequence items with different classes, e.g.

```
 actionsequence = ( ... files.one  ..  files.two )
```

can now be handled more elegantly using bundles. The natural ordering uses that fact that some operations are mutually exclusive and that some operations do not make sense in reverse order. For example, editing a file and then copying onto it would be nonsense. Similarly, you cannot both remove a file and rename it.

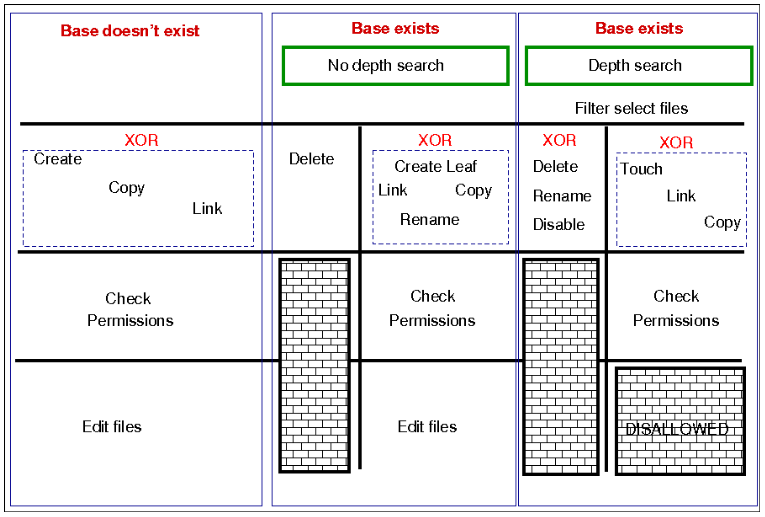
File copying

One of the first things users of cfengine 2 will notice is that copying is now 'backwards'. Instead of the default object being source and the option being the destination, in cfengine 3 the destination is paramount and the source is an option. This is because the model of voluntary cooperation tells us that it is the object that is changed which is the agent making the promise. One cannot force change onto a destination with cfengine, one can only invite change from a source.

Normal ordering

Cfengine 3 no longer has an 'action sequence'. Ordering of operations has, in most cases, a natural ordering which is assumed by the agent. For instance: 'delete then create' (normal ordering) makes sense, but 'create then delete' does not. This sort of principle can be extended to deal with all aspects of file promises.

The diagram below shows the ordering. Notice that the same ordering applies regardless of file type (plain-file or directory).



The pseudo-code for this logic is shown in the diagram and below:

```
for each file promise-object
   {
   if (depth_search)

     do
       DepthSearch (HandleLeaf)
     else
       (HandleLeaf)
     done
   }

HandleLeaf()
  {
  Does leaf-file exist?

    NO:  create
    YES: rename,delete,touch,

    do
     for all servers in {localhost, @(servers)}
        {
        if (server-will-provide)
           do
             if (depth_search)
                embedded source-depth-search (use file source)
                break
             else
```

```
                (use file source)
                break
             done
           done
         }
     done

   Do all links (always local)

   Check Permissions

   Do edits
   }
```

Depth searches (recursion) during searches

In cfengine 2 there was the concept of recursion during file searches. Recursion is now called "depth-search". In addition, it was possible to specify wildcards in the base-path for this search. Cfengine 3 replaces the 'globbing' symbols with standard regular expressions:

```
      Cfengine 2                Cfengine 3

/one/*/two/thr*/four    /one/.*/two/thr.*/four
```

When we talk about a depth search, it refers to a search for file objects which starts from the one or more matched base-paths as shown in the example above.

Local and remote searches

There are two distinct kinds of depth search:

- A local search over promiser agents.
- A remote search over provider agents.

When we are *copying* or *linking* to a file source, it is the search over the *remote* source that drives the content of a promise (the promise is a promise to use what the remote source provides). In general, the sources are on a different device to the images that make the promises. For all other promises, we search over existing local objects.

If we specify depth search together with copy of a directory, then the implied remote source search is assumed, and it is made after the search over local base-path objects has been made. If you mix complex promise body operations in a single prmose, this could lead to confusion about the resulting behaviour, and a warning is issued. In general it is not recommended to mix searches without a full understanding of the consequences, but this might occasionally be useful.

Depth search is not allowed with editfiles promises.

File editing in cfengine 3

Cfengine 2 assumed that all files were line-edited, because it was based on Unix traditions. Since then many new file formats have emerged, including XML. Cfengine 3 opens up the possibiltiy for multiple models of file editing. Line based editing is still present and is both much simplified and much more powerful than previously.

File editing is not just a single kind of promise but a whole range of ‘promises within files’. It is therefore not merely a body to a single kind of promise, but a bundle of sub-promises. After all, inside each file is a new world of objects that can make promises, quite separate from files’ external attributes.

A typical file editing stanza has the elements in the following example.

```
######################################################################
#
# File editing
#
######################################################################

body common control

{
version => "1.2.3";
bundlesequence  => { "outerbundle"  };
}

########################################################

bundle agent outerbundle

{
files:

  "/home/mark/tmp/cf3_test"

       create    => "true",     # Like autocreate in cf2
       edit_line => inner_bundle;
}

########################################################

bundle edit_line inner_bundle
  {
  vars:

   "edit_variable" string => "private edit variable";

  replace_patterns:

   # replace shell comments with C comments

   "#(.*)"

      replace_with => C_comment,
     select_region => MySection("New section");
  reports:

    someclass::

      "This is file $(edit.filename)"
  }

########################################
# Bodies for the library ...
########################################

body replace_with C_comment

{
replace_value => "/* $(1) */"; # backreference 0
occurrences => "all";          # first, last all
}

########################################################

body select_region MySection(x)

{
select_start => "\[$(x)\]";
select_end => "\[.*\]";
}
```

There are several things to notice:

- The line-editing promises are all convergent promises about patterns within the file. They have bodies, just like other attributes do and these allow us to make simple templates about file editing while extending the power of the basic primitives.
- All pattern matching is through perl compatible regular expressions
- Editing takes place within a marked region (which defaults to the whole file).
- Search/replace functions now allow back-references.
- The line edit model now contains a field or column model for dealing with tabular files such as Unix '`passwd`' and '`group`' files. We can now apply powerful convergent editing operations to single fields inside a table, to append, order and delete items from lists inside fields.
- The special variable `$(edit.filename)` contains the name of the file being edited within an edit bundle.

In the example above, back references are used to allow conversion of comments from shell-style to C-style.

```
bundle agent example
{
files:

  "/home/mark/tmp" -> "Security team"

       changes      => lay_a_tripwire,
       depth_search => recurse("inf"),
       action       => background;
}

#########################################################

body changes lay_a_tripwire

{
hash           => "md5";
report_changes => "content";
update         => "yes";
}
```

7.2.1 file_select (compound body)

Type: (ext body)

'leaf_name'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regexes that match an acceptable name

Example:

```
body file_select example
{
leaf_name => { "S[0-9]+[a-zA-Z]+", "K[0-9]+[a-zA-Z]+" };
file_result => "leaf_name";
}
```

Notes:

This pattern matches only the node name of the file, not its path.

'`path_name`'

Type: slist

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: List of pathnames to match acceptable target

Example:

```
body file_select example
{
leaf_name => { "prog.pid", "prog.log" };
path_name => { "/etc/.*", "/var/run/.*" };

file_result => "leaf_name.path_name"
}
```

Notes:

Path name and leaf name can be conveniently tested for separately by use of appropriate regular expressions.

'`search_mode`'

Type: string

Allowed input range: `[0-7augorwxst,+-]+`

Synopsis: Mode mask for acceptable files

Example:

```
body file_select example
{
search_mode => "644";
}
```

Notes:

The mode may be specified in symbolic or numerical form with '+' and '-' constraints.

'`search_size`'

Type: irange [int,int]

Allowed input range: `0,inf`

Synopsis: Integer range of file sizes

Example:

```
body file_select example
{
search_size => irange("0","20k");
file_result => "size";
}
```

Notes:

'search_owners'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: List of acceptable user names or ids for the file

Example:

```
body file_select example
{
search_owners => { "mark", "jeang", "student_.*" };
file_result => "owner";
}
```

Notes:

A list of regular expressions.

'search_groups'

Type: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: List of acceptable group names or ids for the file

Example:

```
body file_select example
{
search_group => { "users", "special_.*" };
file_result => "group";
}
```

Notes:

'search_bsdflags'

Type: string

Allowed input range: `[(arch|archived|dump|opaque|sappnd|sappend|schg|schange|simmutable|sunl`

Synopsis: String of flags for bsd file system flags expected set

Example:

```
body file_select xyz
{
search_bsdflags => "archived|dump";
}
```

Notes:

Extra BSD file system flags.

'ctime'

Type: irange [int,int]

Allowed input range: `0,4026531839`

Synopsis: Range of change times (ctime) for acceptable files

Example:

```
body files_select example
{
ctime => irange(ago(1,0,0,0,0,0),now);
file_result => "ctime";
}
```

Notes:

The file's change time refers to both modification of content and attributes such as permissions.

'mtime'

Type: irange [int,int]

Allowed input range: `0,4026531839`

Synopsis: Range of modification times (mtime) for acceptable files

Example:

```
body files_select example
{
mtime => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}
```

Notes:

The file's modification time refers to both modification of content but not other attributes such as permissions.

'atime' **Type**: irange [int,int]

Allowed input range: `0,4026531839`

Synopsis: Range of access times (atime) for acceptable files

Example:

```
body file_select

{
# files accessed in the last hour

atime     => irange(ago(0,0,0,1,0,0),now);
file_result => "atime";
}


body file_select

{
# files accessed since 00:00 1st Jan 2000

atime     => irange(on(2000,1,1,0,0,0),now);
file_result => "atime";
}
```

Notes:

A range of times during which a file was accessed can be specified in a `file_select` body. (Like file filters in cfengine 2.)

'exec_regex'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Matches file if this regular expression matches any full line returned by the command

Example:

```
body file_select example
{
exec_regex => "SPECIAL_LINE: .*";
exec_program => "/path/test_program $(this.promiser)";
file_result => "exec_program.exec_regex";
}
```

Notes:

The regular expression must be used in conjuection with the `exec_program` test. In this way the program must both return exit status 0 and its output must match the regular expression.

'exec_program'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Execute this command on each file and match if the exit status is zero

Example:

```
body file_select example
{
exec_program => "/path/test_program $(this.promiser)";
file_result => "exec_program";
}
```

Notes:

This is part of the customizable file search criteria. If the user-defined program returns exit status 0, the file is considered matched.

'file_types'

Type: (option list)

Allowed input range:

```
               plain
               reg
               symlink
               dir
               socket
               fifo
               door
               char
               block
```

Synopsis: List of acceptable file types from menu choices

Example:

```
body file_select filter
{
file_types => { "plain","symlink" };

file_result => "file_results";
}
```

Notes:

File types vary in details between operating systems. The main POSIX types are provided here as menu options.

'issymlinkto'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regular expressions to match file objects

Example:

```
body file_select example
{
issymlinkto => { "/etc/[^/]*", "/etc/init.d/[a-z0-9]*" };
}
```

Notes:

A list of regular expressions. If the file is a symbolic link which points to files matched by one of these expressions, the file will be selected.

'file_result'

Type: string

Allowed input range: `[(leaf_name|path_name|file_types|mode|size|owner|group|atime|ctime|mtim` `regex|exec_program)[|&!.]*]*`

Synopsis: Logical expression combining classes defined by file search criteria

Example:

```
body file_select any_age

{
mtime       => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}

body file_select pdf_files_1dayold

{
mtime       => irange(ago(0,0,1,0,0,0),now);
leaf_name   => { ".*.pdf" , ".*.fdf" };

file_result => "leaf_name&mtime";
}
```

Notes:

Sets the criteria for file selection outcome during file searches. The syntax is the same as for a class expression since the file selection is a classification of the file-search in the same way that system classes are a classification of the abstact host-search.

7.2.2 `copy_from` (compound body)

Type: (ext body)

'source'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Reference source file from which to copy

Example:

```
body copy_from example
```

```
{
source => "/path/to/source";
}

# or

body link_from example
{
source => "/path/to/source";
}
```

Notes:

For remote copies this refers to the file name on the remote server.

'servers' **Type**: slist

Allowed input range: (arbitrary string)

Synopsis: List of servers in order of preference from which to copy

Example:

```
body copy_from example
{
servers => { "primary.example.org", "secondary.example.org",
                 "tertiary.other.domain" };
}
```

Notes:

The servers are tried in order until one of them succeeds.

'portnumber'

Type: int

Allowed input range: `1024,99999`

Synopsis: Port number to connect to on server host

Example:

```
body copy_from example
{
portnumber => "5308";
}
```

Notes:

The standard or registered port number is tcp/5308. Cfengine does not presently use its registered udp port with the same number, but this could change in the future.

'copy_backup'

Type: (menu option)

Allowed input range:

```
          true
          false
          timestamp
```

Synopsis: Menu option policy for file backup/version control

Example:

```
body copy_from example
{
copy_backup => "timestamp";
}
```

Notes:

Determines whether a backup of the previous version is kept on the system. This should be viewed in connection with the system repository, since a defined repository affects the location at which the backup is stored.

'stealth' **Type**: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false whether to preserve time stamps on copied file

Example:

```
body copy_from example
{
stealth => "true";
}
```

Notes:

Preserves file access and modification times on the promiser files.

'preserve' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to preserve file permissions on copied file

Example:

```
body copy_from example
{
preserve => "true";
}
```

Notes:

Whether or not the copy preserves the permissions on the source files.

'linkcopy_patterns'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of patterns matching symbolic links that should be replaced with copies

Example:

```
body copy_from mycopy(from)
```

```
{
source            => "$(from)";
linkcopy_patterns => { ".*" };
}
```

Notes:

The pattern matches the last node filename (i.e. without the absolute path).

'copylink_patterns'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of patterns matching files that should be linked instead of copied

Example:

```
body copy_from example
{
copylink_patterns => { "special_node1", "other_node.*" };
}
```

Notes:

The matches are performed on the last node of the filename, i.e. the file without its path.

'compare' **Type**: (menu option)

Allowed input range:

```
               atime
               mtime
               ctime
               digest
               hash
```

Synopsis: Menu option policy for comparing source and image file attributes

Example:

```
body copy_from example

{
compare => "digest";
}
```

Notes:

The default copy method is 'mtime' (modification time), meaning that the source file is copied to the destination (promiser) file, if the source file has been modified more recently than the destination.

'link_type'

Type: (menu option)

Allowed input range:

```
symlink
hardlink
relative
absolute
none
```

Synopsis: Menu option for type of links to use when copying

Example:

```
body link_from example
{
link_type => "hard";
}
```

Notes:

What kind of link should be used to link files. Users are advised to be wary of 'hard links' (see Unix manual pages for the 'ln' command). The behaviour of non-symbolic links is often precarious and unpredictable.

'type_check'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false compare file types before copying and require match

Example:

```
body copy_from example
{
type_check => "false";
}
```

Notes:

File types at source and destination should normally match in order for updates to overwrite them. This option allows this checking to be switched off.

'force_update'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false force copy update always

Example:

```
body copy_from example
{
force_update => "true";
}
```

Notes:

Warning: this is a non-convergent operation. Although the end point might stabilize in content, the operation will never quiesce. Use of this feature is not recommended except in exceptional circumstances since it creates a busy-dependency. If the copy is a network copy, the system will be disturbed by network disruptions.

'force_ipv4'

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false force use of ipv4 on ipv6 enabled network

Example:

```
body copy_from example
{
force_ipv4 => "true";
}
```

Notes:

IPv6 should be harmless to most users unless you have a partially or misconfigured setup.

'copy_size'

Type: irange [int,int]

Allowed input range: `0,inf`

Synopsis: Integer range of file sizes that may be copied

Example:

```
body copy_from example
{
copy_size => irange("0","50000");
}
```

Notes:

The use of the irange function is optional. Ranges may also be specified as a comma separated numbers.

'trustkey' **Type**: (menu option)

Allowed input range:

```
                true
                false
                yes
                no
                on
                off
```

Synopsis: true/false trust public keys from remote server if previously unknown

Example:

```
body copy_from example
{
trustkey => "true";
}
```

Notes:

If the server's public key has not already been trusted, this allows us to accept the key in automated key-exchange.

Note that, as a simple security precaution, trustkey should normally be set to '`false`', to avoid key exchange with a server one is not one hundred percent sure about, though the risks for a client are rather low. On the server-side however, trust is often granted to many clients or to a whole network in which possibly unauthorized parties might be able to obtain an IP address, thus the trust issue is most important on the server side.

As soon as a public key has been exchanged, the trust option has no effect. A machine that has been trusted remains trusted until its key is manually revoked by a system administrator. Keys are stored in '`WORKDIR/ppkeys`'.

'`encrypt`' **Type**: (menu option)

Allowed input range:

```
                true
                false
                yes
                no
                on
                off
```

Synopsis: true/false use encrypted data stream to connect to remote host

Example:

```
body copy_from example
```

```
{
servers  => { "remote-host.example.org" };
encrypt => "true";
}
```

Notes:

Client connections are encrypted with using a Blowfish randomly generated session key. The intial connection is encrypted using the public/private keys for the client and server hosts.

'verify' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false verify transferred file by hashing after copy (resource penalty)

Example:

```
body copy_from example
{
verify => "true";
}
```

Notes:

This is a highly resource intensive option, not recommended for large file transfers.

'purge' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false purge files on client that do not match files on server when depth_search

Example:

```
body copy_from example
{
purge => "true";
}
```

Notes:

Purging files is a potentially dangerous matter during a file copy it implies that any promiser (destination) file which is not matched by a source will be deleted. Since there is no source, this means the file will be irretrievable. Great care should be exercised when using this feature.

'`check_root`'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false check permissions on the root directory when depth_search

Example:

```
body copy_from example
{
check_root => "true";
}
```

Notes:

When copying files recursively (by depth search), this flag determines whether the permissions of the root directory should be set from the root of the source. The default is to check only copied file objects and subdirectories within this root (false).

'findertype'

Type: (menu option)

Allowed input range:

```
MacOSX
```

Synopsis: Menu option for default finder type on MacOSX

Example:

```
body copy_from example
{
findertype => "MacOSX";
}
```

Notes:

This applies only to the Macintosh OSX variants.

7.2.3 link_from (compound body)

Type: (ext body)

'source' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: The source file to which the link should point

Example:

```
body copy_from example
{
source => "/path/to/source";
}

# or

body link_from example
{
source => "/path/to/source";
}
```

Notes:

For remote copies this refers to the file name on the remote server.

'link_type'

Type: (menu option)

Allowed input range:

```
                symlink
                hardlink
                relative
                absolute
                none
```

Synopsis: The type of link used to alias the file

Example:

```
body link_from example
{
link_type => "hard";
}
```

Notes:

What kind of link should be used to link files. Users are advised to be wary of 'hard links' (see Unix manual pages for the 'ln' command). The behaviour of non-symbolic links is often precarious and unpredictable.

'copy_patterns'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A set of patterns that should be copied ansd synchronized instead of linked

Example:

```
body link_from example
{
copy_patterns =>  { "special_node1", "/path/special_node2" };
}
```

Notes:

During the linking of files, it is sometimes useful to buffer changes with an actual copy, especially if the link is to an emphemeral file system. This list of patterns matches files that arise during a linking policy. A positive match means that the file should be copied and updated by modification time.

'when_no_source'

Type: (menu option)

Allowed input range:

```
          force
          delete
          nop
```

Synopsis: Behaviour when the source file to link to does not exist

Example:

```
body link_from example
{
when_no_file => "force";
}
```

Notes:

If we try to create a link to a file that does not exist a link, how should cfengine respond? The options are to force the creation to a file that does not (yet) exist, delete any existing link, or do nothing.

'link_children'

Type: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false whether to link all directory's children to source originals

Example:

```
body link_from example
{
link_children => "true";
}
```

Notes:

If the promiser is a directory, instead of copying the children, link them to the source.

'when_linking_children'

Type: (menu option)

Allowed input range:

```
override_file
if_no_such_file
```

Synopsis: Policy for overriding existing files when linking directories of children

Example:

```
body link_from example
{
when_linking_children => "if_no_such_file";
}
```

Notes:

The options refer to what happens if the directory exists already and is already partially populated with files. If the directory being copied from contains a file with the same name as that of a link to be created, we must decide whether to override the existing destination object with a link or simply omit the automatic linkage for files that already exist. The latter case can be used to make a copy of one directory with certain fields overridden.

7.2.4 perms (compound body)

Type: (ext body)

'mode' **Type**: string

Allowed input range: [0-7augorwxst,+-]+

Synopsis: File permissions (like posix chmod)

Example:

```
body perms example
{
```

```
mode => "a+rx,o+w";
}
```

Notes:

The mode string may be symbolic or numerical, like `chmod`.

‘owners’ **Type**: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: List of acceptable owners or user ids, first is change target

Example:

```
body perms example
{
owners => { "mark", "wwwrun", "jeang" };
}
```

Notes:

The first user is the reference value that cfengine will set the file to if none of the list items matches the true state of the file.

‘groups’ **Type**: slist

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: List of acceptable groups of group ids, first is change target

Example:

```
body perms example
{
groups => { "users", "administrators" };
}
```

Notes:

The first named group is the list is the defaul that will be configured if the file does not match an element of the list.

‘rxdirs’ **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false add execute flag for directories if read flag is set

Example:

```
body perms rxdirs
{
rxdirs => "false";
}
```

Notes:

Default behaviour is to set the ‘x’ flag on directories automatically if the ‘r’ flag is specified when specifying multiple files in a single promise.

‘bsdflags’ **Type**: (option list)

Allowed input range:

```
arch
archived
dump
opaque
sappnd
sappend
schg
schange
simmutable
sunlnk
sunlink
uappnd
uappend
uchg
uchange
uimmutable
uunlnk
uunlink
```

Synopsis: List of menu options for bsd file system flags to set

Example:

```
body perms example

{
#..
bsdflags => { "uappnd","uchg","uunlnk","nodump",
              "opaque","sappnd","schg","sunlnk" };
}
```

Notes:

The free BSD Unices and MacOSX have additional filesystem flags which can be set. Refer to the BSD `chflags` documentation for this.

7.2.5 `changes` (compound body)

Type: (ext body)

'`hash`'

Type: (menu option)

Allowed input range:

```
md5
sha1
best
```

Synopsis: Hash files for change detection

Example:

```
body changes example
{
hash => "md5";
}
```

Notes:

The `best` option cross correlates the best two available algorithms known in the OpenSSL library.

'`report_changes`'

Type: (menu option)

Allowed input range:

```
all
stats
content
none
```

Synopsis: Specify criteria for change warnings

Example:

```
body changes example
{
report_changes => "content";
}
```

Notes:

Files can change in permissions and contents, i.e. external or internal attributes. If '`all`' is chosen all attributes are checked.

'`update_hashes`'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Update hash values immediately after change warning

Example:

```
body changes example
{
update_hashes => "true";
}
```

Notes:

If this is positive, file hashes should be updated as soon as a change is registered so that multiple warnings are not given about a single change. This applies to addition and removal too.

7.2.6 delete (compound body)

Type: (ext body)

'dirlinks' **Type**: (menu option)

Allowed input range:

```
delete
tidy
keep
```

Synopsis: Menu option policy for dealing with symbolic links to directories during deletion

Example:

```
body delete example
{
dirlinks => "keep";
}
```

Notes:

Links to directories are normally removed just like any other link or file objects. By keeping directory links, you preserve the logical directory structure of the file system.

'rmdirs' **Type**: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to delete empty directories during recursive deletion

Example:

```
body delete example
{
```

```
rmdirs => "true";
}
```

Notes:

When deleting files recursively from a base directory, should we delete empty directories also, or keep the directory structure intact?

7.2.7 `rename` (compound body)

Type: (ext body)

‘`newname`’

Type: string

Allowed input range: (arbitrary string)

Synopsis: The desired name for the current file

Example:

```
body rename example(s)
{
newname => "$(s)";
}
```

Notes:

‘`disable_suffix`’

Type: string

Allowed input range: (arbitrary string)

Synopsis: The suffix to add to files when disabling (.cfdisabled)

Example:

```
body rename example
{
disable => "true";
disable_suffix => ".nuked";
}
```

Notes:

To make disabled files in a particular manner, use this string suffix. The default value is '.cf-disabled'.

'disable' **Type**: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false automatically rename and remove permissions

Example:

```
body rename example
{
disable => "true";
disable_suffix => ".nuked";
}
```

Notes:

Disabling a file means making is impotent in the context in which it has an effect. For executables this means preventing execution, for an information file it means making the file unreadable.

'rotate' **Type**: int

Allowed input range: `0,99`

Synopsis: Maximum number of file rotations to keep

Example:

```
body rename example
{
rotate => "4";
}
```

Notes:

Used for log rotation.

'disable_mode'

Type: string

Allowed input range: `[0-7augorwxst,+-]+`

Synopsis: The permissions to set when a file is disabled

Example:

```
body rename example
{
disable_mode => "0600";
}
```

Notes:

To disable an executable it is not enough to rename it, you should also remove the executable flag.

7.2.8 `repository`

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Name of a repository for versioning

Example:

```
files:

 "/path/file"

   copy_from => source,
   repository => "/var/cfengine/repository";
```

Notes:

A local repository for this object, overrides the default.

7.2.9 `edit_line`

Type: (ext bundle) (Separate Bundle)

7.2.10 `edit_xml`

Type: (ext bundle) (Separate Bundle)

7.2.11 edit_defaults (compound body)

Type: (ext body)

'edit_backup'

Type: (menu option)

Allowed input range:

```
          true
          false
          timestamp
          rotate
```

Synopsis: Menu option for backup policy on edit changes

Example:

```
body edit_defaults example
{
edit_backup => "timestamp";
}
```

Notes:

'max_file_size'

Type: int

Allowed input range: 0,99999999999

Synopsis: Do not edit files bigger than this number of bytes

Example:

```
body edit_defaults example
{
max_file_size => "50K";
}
```

Notes:

A local, per-file sanity check to make sure the file editing is sensible. If this is set to zero, the check is disabled and any size may be edited.

'empty_file_before_editing'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Baseline memory model of file to zero/empty before commencing promised edits

Example:

```
body edit_defaults example
{
empty_file_before_editing => "true";
}
```

Notes:

Emptying a file before reconstructing its contents according to a fixed recipe allows an ordered procedure to be convergent.

7.2.12 `depth_search` (compound body)

Type: (ext body)

'`include_dirs`'

Type: slist

Allowed input range: `.*`

Synopsis: List of regexes of directory names to include in depth search

Example:

```
body depth_search example
{
include_dirs => { "subdir1", "subdir2", "pattern.*" };
}
```

Notes:

This is the complement of `exclude_dirs`.

'exclude_dirs'

Type: slist

Allowed input range: `.*`

Synopsis: List of regexes of directory names NOT to include in depth search

Example:

```
body depth_search
{
# no dot directories
exclude_dirs => { "\..*" };
}
```

Notes:

Directory names are treated specially when searching recursively through a file system.

'include_basedir'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false include the start/root dir of the search results

Example:

```
body depth_search example
{
include_basedir => "true";
}
```

Notes:

When checking files recursively (with `depth_search`) the promiser is a directory. This parameter determines whether that initial directory should be considered part of the promise or simply a boundary which marks the edge of the search. If true, the promiser directory will also promise the same attributes as the files inside it.

'depth'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Maximum depth level for search

Example:

```
body depth_search example
{
depth => "inf";
}
```

Notes:

This was previous called 'recurse' in earlier versions of cfengine. Note that the value '`inf`' may be used for an unlimited value.

'xdev'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false exclude directories that are on different devices

Example:

```
body depth_search example
{
xdev => "true";
}
```

Notes:

'traverse_links'

Type: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false traverse symbolic links to directories (false)

Example:

```
body depth_search example
{
traverse_links => "true";
}
```

Notes:

If this is true, `cf-agent` will treat symbolic links to directories as if they were directories. Normally this is considered a potentially dangerous assumption and links are not traversed.

'`rmdeadlinks`'

Type: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false remove links that point to nowhere

Example:

```
body depth_search example
{
rmdeadlinks => "true";
}
```

Notes:

If we find links that point to non-existence files, should we delete them or keep them?

7.2.13 `touch`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to touch time stamps on file
Example:

```
files:

 "/path/file"

   touch => "true";
```

Notes:

7.2.14 `create`

Type: (menu option)
Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false whether to create non-existing file
Example:

```
files:

  "/path/plain_file"

     create =>   "true";

  "/path/dir/."

     create =>   "true";
```

Notes:

Directories are created by using the '`/.`' to signify a directory type. Note that, if no permissions are specified, mode 600 is chosen for a file, and mode 755 is chosen for a directory. If you cannot accept these defaults, you *should* specify permissions.

7.2.15 move_obstructions

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether to move obstructions to file-object creation

Example:

```
files:

  "/tmp/testcopy"

    copy_from    => mycopy("/tmp/source"),
    move_obstructions => "true",
    depth_search => recurse("inf");
```

Notes:

If we have promised to make file '`X`' a link, but it already exists as a file, or vice-versa, or if a file is blocking the creation of a directory etc, then normally cfengine will report an error. If this is set, existing objects will be moved aside to allow the system to heal without intervention. Files and directories are saved/renamed, but symbolic links are deleted.

7.2.16 `transformer`

Type: string
Allowed input range: `[cC]:\\.*|/.*`
Synopsis: Shell command (with full path) used to transform current file
Example:

```
 "/home/mark/tmp/testcopy"

    file_select => pdf_files,
    transformer => "/usr/bin/gzip $(this.promiser),
    depth_search => recurse("inf");
```

Notes:

A command to execute on finding a file.

7.2.17 `pathtype`

Type: (menu option)
Allowed input range:

```
               literal
               regex
```

Synopsis: Menu option for interpreting promiser file object
Example:

```
files:

   "/var/.*/lib"

      pathtype => "regex", #default
         perms => system;

   "/var/.*/lib"

      pathtype => "literal",
         perms => system;
```

Notes:

If the keyword `literal` is invoked, a path looking like a regular expression will be treated as a literal string. Thus in the example, one case implies an iteration over all files/directories matching the regular expression, while the other means a single literal object with a name composed of dots and stars.

7.2.18 acl (compound body)

Type: (ext body)

'acl_method'

Type: (menu option)

Allowed input range:

```
append
prepend
set
```

Synopsis: Editing method for access control list

Example:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

Notes:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

'acl_type' **Type**: (menu option)

Allowed input range:

```
solaris
linux
ntfs
afs
```

Synopsis: Access control list type for the affected file system

Example:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

Notes:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

'acl_entry'

Type: slist

Allowed input range: `.*`

Synopsis: Native settings for access control entry

Example:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

Notes:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/
""
```

7.3 `insert_lines` promises in '`edit_line`'

This promise is part of the line-editing model. It inserts lines into the file at a specified location. The location is determined by body-attributes. The promise object referred to can be a literal line of a file-reference from which to read lines.

```
insert_lines:

  "literal line or file reference"

     location => location_body,
     ...;
```

```
body common control

{
any::
  bundlesequence  => {
                     example
                     };
}

#######################################################

bundle agent example

{
files:

  "/var/spool/cron/crontabs/root"

     edit_line => addline;
}

#######################################################
# For the library
#######################################################

bundle edit_line addline

{
insert_lines:

 "0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/cfengine/bin/cf-execd -F";

}
```

By parameterizing the editing bundle, one can make generic and reusable editing bundles.

7.3.1 `location` (compound body)

Type: (ext body)

'`select_line_matching`'

> **Type**: string
>
> **Allowed input range**: `.*`
>
> **Synopsis**: Regular expression for matching file line location
>
> **Example**:

```
body location example
{
select_line_matching => "^Expression match.* whole line$";
}
```

Notes:

The '^$' are not necessary, just remember that the expression must match a whole line, not a fragment within a line.

'before_after'

Type: (menu option)

Allowed input range:

```
before
after
```

Synopsis: Menu option, point cursor before of after matched line

Example:

```
body location append

{
#...
before_after => "before";
}
```

Notes:

Determines whether an edit will occur before or after the currently matched line.

'first_last'

Type: (menu option)

Allowed input range:

```
first
last
```

Synopsis: Menu option, choose first or last occurrence of match in file

Example:

```
body location example
{
first_last => "last";
}
```

Notes:

In multiple matches, decide whether the first or last occurrence of the matching pattern in the case affected by the change. In principle this could be generalized to more cases but this seems like a fragile quality to evaluate, and only these two cases are deemed of reproducible significance.

7.3.2 insert_type

Type: (menu option)
Allowed input range:

```
literal
string
file
```

Synopsis: Type of object the promiser string refers to (default literal)
Example:

```
body insert_lines example
{
insert_type => "file";
}
```

Notes:

The default is to treat the promiser as a literal string. This is used to tell cfengine that the string is non-literal and should be interpreted as a filename from which to import lines.

7.3.3 insert_select (compound body)

Type: (ext body)

'insert_if_startwith_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if it starts with a string in the list

Example:

```
body insert_select example
{
insert_if_startwith_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The list contains literal strings to search for in an secondary file (not the main file being edited). If the string is found as the first characters (at the start) of a line in the file, that line from the secondary file will be inserted at the present location in the primary file.

'`insert_if_not_startwith_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if it DOES NOT start with a string in the list

Example:

```
body insert_select example
{
insert_if_not_startwith_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The complement of `insert_if_startwith_from_list`.

'`insert_if_match_from_list`'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if it fully matches a regex in the list

Example:

```
body insert_select example
{
insert_if_match_from_list => { ".*find_.*_1.*", ".*find_.*_2.*" };
}
```

Notes:

The list contains regular expressions to search for in an secondary file (not the main file being edited). If the regex matches a complete line of the file, that line from the secondary file will be inserted at the present location in the primary file.

'insert_if_not_match_from_list'

Type: slist

Allowed input range: `.*Insert line if it DOES NOT fully match a regex in the list`

Synopsis: (null)

Example:

```
body insert_select example
{
insert_if_not_match_from_list => { ".*find_.*_1.*", ".*find_.*_2.*" };
}
```

Notes:

The complement of `insert_if_match_from_list`.

'insert_if_contains_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if a regex in the list match a line fragment

Example:

```
body insert_select example
{
insert_if_contains_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The list contains literal strings to search for in an secondary file (not the main file being edited). If the string is found in a line of the file, that line from the secondary file will be inserted at the present location in the primary file.

'insert_if_not_contains_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Insert line if a regex in the list DOES NOT match a line fragment

Example:

```
body insert_select example
{
insert_if_not_contains_from_list => { "find_me_1", "find_me_2" };
}
```

Notes:

The complement of `insert_if_contains_from_list`.

7.3.4 expand_scalars

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: Expand any unexpanded variables

Example:

```
body insert_lines example
{
insert_type    => "file";
expand_scalars => "true";
}
```

Notes:

A way of incorporating templates with variable expansion into file operations. Variables should be named and scoped appropriately for the bundle in which this promise is made.

In cfengine 2 `editfiles` this was called '`ExpandVariables`'.

7.4 field_edits promises in 'edit_line'

Certain types of text file (e.g. the 'passwd' and 'group' files in Unix) are tabular in nature, with field separators (e.g. ':' or ','). This promise assumes a parameterizable model for editing the fields of such files, using a regular expression to separate major fields and a character to separate sub-fields. First you match the line with a regular expression, then a field_edits body describes the separators for fields and one level of sub-fields, along with policies for editing these fields, ordering the items within them etc.

```
field_edits:

    "regex matching line"

              edit_field => body;
```

```
bundle agent example

{
vars:

 "userset" slist => { "one-x", "two-x", "three-x" };

files:

  "/tmp/passwd"

       create    => "true",
       edit_line => SetUserParam("mark","6","/set/this/shell");

  "/tmp/group"

       create    => "true",
       edit_line => AppendUserParam("root","4","@(userset)");
}

########################################################

bundle edit_line SetUserParam(user,field,val)
  {
  field_edits:

   "$(user).*"
```

```
      # Set field of the file to parameter

      edit_field => col(":","$(field)","$(val)","set");
  }

########################################################

bundle edit_line AppendUserParam(user,field,allusers)
  {
  vars:

    "val" slist => { @(allusers) };

  field_edits:

   "$(user).*"

      # Set field of the file to parameter

      edit_field => col(":","$(field)","$(val)","alphanum");

  }

########################################
# Bodies
########################################

body edit_field col(split,col,newval,method)

{
field_separator => "$(split)";
select_field    => "$(col)";
value_separator  => ",";
field_value     => "$(newval)";
field_operation => "$(method)";
extend_fields => "true";
}
```

Field editing allows us to edit tabular files in a unique way, adding and removing data from addressable fields. The ‘passwd’ and ‘group’ files are classic examples of tabular files, but there are many ways to use this feature, e.g. edit a string

```
VARIABLE="one two three"
```

View this line as a tabular line separated by '"' and with sub-separator given by the space.

7.4.1 edit_field (compound body)

Type: (ext body)

'field_separator'

> **Type**: string
>
> **Allowed input range**: `.*`
>
> **Synopsis**: The regular expression used to separate fields in a line
>
> **Example**:

```
body edit_field example
{
field_separator => ":";
}
```

> **Notes**:
>
> Most tabular files are separated by simple characters, but by allowing a general regular expression one can make creative use of this model to edit all kinds of line-based text files.

'select_field'

> **Type**: int
>
> **Allowed input range**: `0,99999999999`
>
> **Synopsis**: Integer index of the field required 1..n
>
> **Example**:

```
body field_edits example
{
select_field => "5";
}
```

> **Notes**:
>
> Numering starts from 1 not from 0.

'value_separator'

> **Type**: string
>
> **Allowed input range**: `^.$`

Synopsis: Character separator for subfields inside the selected field

Example:

```
body field_edit example
{
value_separator => ",";
}
```

Notes:

For example, elements in the group file are separated by ':', but the lists of users in these fields are separated by ','.

'field_value'

Type: string

Allowed input range: `.*`

Synopsis: Set field value to a fixed value

Example:

```
body edit_field example(s)
{
field_value => "$(s)";
}
```

Notes:

Set a field to a constant value, e.g. reset the value to a constant default, empty the field, or set it fixed list.

'field_operation'

Type: (menu option)

Allowed input range:

```
prepend
append
alphanum
delete
set
```

Synopsis: Menu option policy for editing subfields

Example:

```
body edit_field example
{
field_operation => "append";
}
```

Notes:

The method by which to edit a field in multi-field/column editing of tabular files.

'extend_fields'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false add new fields at end of line if necessary to complete edit

Example:

```
body edit_field example
{
extend_fields => "true";
}
```

Notes:

Blank fields in a tabular file can be eliminated or kept depending in this setting. If in doubt, set this to true.

'allow_blank_fields'

Type: (menu option)

Allowed input range:

```
true
```

```
                    false
                    yes
                    no
                    on
                    off
```

Synopsis: true/false allow blank fields in a line (do not purge)

Example:

```
body edit_field example
{
# ...
allow_blank_fields => "true";
}
```

Notes:

When editing a file using the field or column model, blank fields, especially at the start and end are generally discarded. If this is set to true, cfengine will retain the blank fields and print the appropriate number of field separators.

7.5 `replace_patterns` promises in '`edit_line`'

This promise refers to arbitrary text patterns in a file. The pattern is expressed as a regular expression and must be compatible with the default model for regular expressions on your system. The default model is PCRE (Perl Compatible Regular Expressions) if available.

```
replace_patterns:

 "search pattern"

    replace_with => replace_body,
    ...;
```

```
bundle edit_line upgrade_cfexecd
  {
  replace_patterns:

    "cfexecd" replace_with => With("cf-execd");
  }

########################################

body replace_with With(x)

{
replace_value => "$(x)";
occurrences => "all";
}
```

This is a straightforward search and replace function. In this case only for line editing the regular expression may match a line fragment – it need not match the entire line.

7.5.1 replace_with (compound body)

Type: (ext body)

'replace_value'

Type: string

Allowed input range: `.*`

Synopsis: Value used to replace regular expression matches in search

Example:

```
body replace_with example(s)
{
replace_value => "$(s)";
}
```

Notes:

'occurrences'

Type: (menu option)

Allowed input range:

```
all
first
```

Synopsis: Menu option to replace all occurrences or just first

Example:

```
body replace_with example
{
occurrences => "first";
}
```

Notes:

A policy for string replacement.

7.6 `delete_lines` promises in '`edit_line`'

This promise assures that certain lines matching regular expression patterns exactly will not be present in a text file. If the lines are found, the default promise is to remove them.

```
bundle edit_line example
  {
  delete_lines:

    "olduser.*";

  }
```

7.6.1 `not_matching`

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false negate match criterion

Example:

```
delete_lines:

  # edit /etc/passwd

  "mark.*|root.*" not_matching => "true";
```

Notes:

The negation of an expression (for convenience).

7.6.2 delete_select (compound body)

Type: (ext body)

'delete_if_startwith_from_list'

Type: slist

Allowed input range: .*

Synopsis: Delete line if it starts with a string in the list

Example:

```
body delete_select example(s)
{
delete_if_startwith_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if they begin with the sub-strings listed.

'delete_if_not_startwith_from_list'

Type: slist

Allowed input range: .*

Synopsis: Delete line if it DOES NOT start with a string in the list

Example:

```
body delete_select example(s)
{
delete_if_not_startwith_from_list => { @(s) };
}
```

Notes:

Delete lines from a file unless they start with the sub-strings in the list given.

'delete_if_match_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if it fully matches a regex in the list

Example:

```
body delete_select example(s)
{
delete_if_match_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if they completely match the regular expressions listed.

'delete_if_not_match_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if it DOES NOT fully match a regex in the list

Example:

```
body delete_select example(s)
{
delete_if_not_match_from_list => { @(s) };
}
```

Notes:

Delete lines from a file unless they fully match regular expressions in the list.

'delete_if_contains_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if a regex in the list match a line fragment

Example:

```
body delete_select example(s)
{
delete_if_contains_from_list => { @(s) };
}
```

Notes:

Delete lines from a file if they contain the sub-strings listed.

'delete_if_not_contains_from_list'

Type: slist

Allowed input range: `.*`

Synopsis: Delete line if a regex in the list DOES NOT match a line fragment

Example:

```
body delete_select discard(s)
{
delete_if_not_contains_from_list => { "substring1", "substring2" };
}
```

Notes:

Delete lines from the file which do not contain the sub-strings listed.

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.7 `interfaces` promises in ‘`agent`’

Interfaces promises describe the configurable aspects relating to network interfaces. Most workstations and servers have only a single network interface, but routers and multi-homed hosts often have multiple interfaces. Interface promises include attributes such as the IP address identity, assumed netmask and routing policy in the case of multi-homed hosts. For virtual machines and hosts, the list of interfaces can be quite large.

```
interfaces:

  "interface name"

    tcp_ip => tcp_ip_body,
    ...;
```

Fill me in (/home/mark/LapTop/Cfengine3/trunk/docs/promise_interfaces_example.texinfo)

Fill me in (/home/mark/LapTop/Cfengine3/trunk/docs/promise_interfaces_notes.texinfo)

7.7.1 tcp_ip (compound body)

Type: (ext body)

'ipv4_address'

Type: string

Allowed input range: `[0-9.]+/[0-4]+`

Synopsis: IPv4 address for the interface

Example:

```
body tcp_ip example
{
ipv4_address => "123.456.789.001";
}
```

Notes:

The address will be checked and if necessary set. Today few hosts will be managed in this way: address management will be handled by other services like DHCP.

'ipv4_netmask'

Type: string

Allowed input range: `[0-9.]+/[0-4]+`

Synopsis: Netmask for the interface

Example:

```
body tcp_ip example
{
ipv4_netmask => "255.255.254.0";
}
```

Notes:

In many cases the CIDR form of address will show the netmask as '/23', but this offers and 'old style' alternative.

'ipv6_address'

Type: string

Allowed input range: `[0-9a-fA-F:]+/[0-9]+`

Synopsis: IPv6 address for the interface

Example:

```
body tcp_ip example
{
ipv6_address => "2001:700:700:3:20f:1fff:fe92:2cd3/64";
}
```

Notes:

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.8 methods promises in 'agent'

Methods are compound promises that refer to whole bundles of promises. Methods may be parameterized. Methods promises are written in a form that is ready for future development. The promiser object is an abstract identifier that refers to a collection (or pattern) of lower level objects that are affected by the promise-bundle. Since the use of these identifiers is for the future, you can simply use any string here for the time being.

```
methods:

  "any"

     usebundle => method_id("parameter",...);
```

Methods are useful for encapsulating repeatedly used configuration issues and iterating over parameters.

In cfengine 2 methods referred to separate sub-programs executed as separate processes. Methods are now implemented as bundles that are run inline.

```
bundle agent example
{
vars:

 "userlist" slist => { "mark", "jeang", "jonhenrik", "thomas", "eben" };

methods:

 "any" usebundle => subtest("$(userlist)");

}

###########################################

bundle agent subtest(user)

{
commands:

 "/bin/echo Fix $(user)";

reports:
 linux::

  "Finished doing stuff for $(user)";
}
```

Methods offer powerful ways to encapsulate multiple issues pertaining to a set of parameters.

7.8.1 `usebundle`

Type: (ext bundle) (Separate Bundle)

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.9 `packages` promises in ‘`agent`’

```
vars:

 "match_package" slist => {
                          "apache2",
                          "apache2-mod_php5",
                          "apache2-prefork",
                          "php5"
                          };
packages:

   "$(match_package)"

        package_policy => "add",
        package_method => yum;
```

Software packaging is a core paradigm in operating system release management today, and cfengine supports a generic approach to integration with native operating support for packaging. Package promises allow cfengine to make promises the state of software packages *conditionally*, given the assumption that a native package manager will perform the actual manipulations. Since no agent can make unconditional promises about another, this is the best that can be achieved.

Packages are treated as black-boxes with three labels:

- A package name.
- A version string.
- An architecture name.

Package managers are treated as black boxes that may support some or all of the following promise types:

- List installed packages
- Add packages
- Delete packages
- Reinstall (repair) packages
- Upgrade packages
- Patch packages
- Verify packages

If these services are promised by a package manager, `cf-agent` promises to use the service and encapsulate it within the overall cfengine framework.

Domain knowledge

Cfengine does not maintain operating system specific expert knowledge internally, rather it uses a generic model for dealing with promises about packages (which depend on the behaviour of an external package manager). The approach is to define package system details in body-constraints that can be written once and for all, for each package system.

Package promises are like `commands` promises in the sense that cfengine promises nothing about the outcome of executing a command. All it can promise is to interface with it, starting it and using the results in good faith. Packages are basically ‘outsourced’, to invoke IT parlance.

The possibility of a cfengine package format that enables more guaranteeable behaviour for special purposes has not been excluded for the future, but in any case `cf-agent` must support native package formats used by operating system maintainers as these are a core part of modern operating systems.

Behaviour

A package promise consists of a name, a version and an architecture, *(n,v,a)*, and behaviour to be promised about packages that match criteria based on these. The components *(n,v,a)* can be determined in one of two different ways:

- They may be specified independently, e.g.

```
packages:

  "mypackage"

     package_policy => "add",
     package_method => rpm,
     package_select => ">=",
     package_architectures => { "x86_64", "i586" },
     package_version => "1.2.3";
```

- They may be extracted from a package identifier or filename, using pattern matching, e.g.:

```
package_list_name_regex    => "[^|]+\|[^|]+\|\s+([^\s|]+).*";
package_list_version_regex => "[^|]+\|[^|]+\|[^|]+\|\s+([^\s|]+).*";
package_list_arch_regex    => "[^|]+\|[^|]+\|[^|]+\|[^|]+\|\s+([^\s]+).*";
```

When scanning a list of installed packages different managers present the information *(n,v,a)* in quite different forms and pattern extraction is necessary. When making a promise about a specific package, the cfengine user may choose one or the other model.

Smart and dumb package systems

Package managers vary enormously in their capabilities and in the kinds of promises they make. There are broadly two types

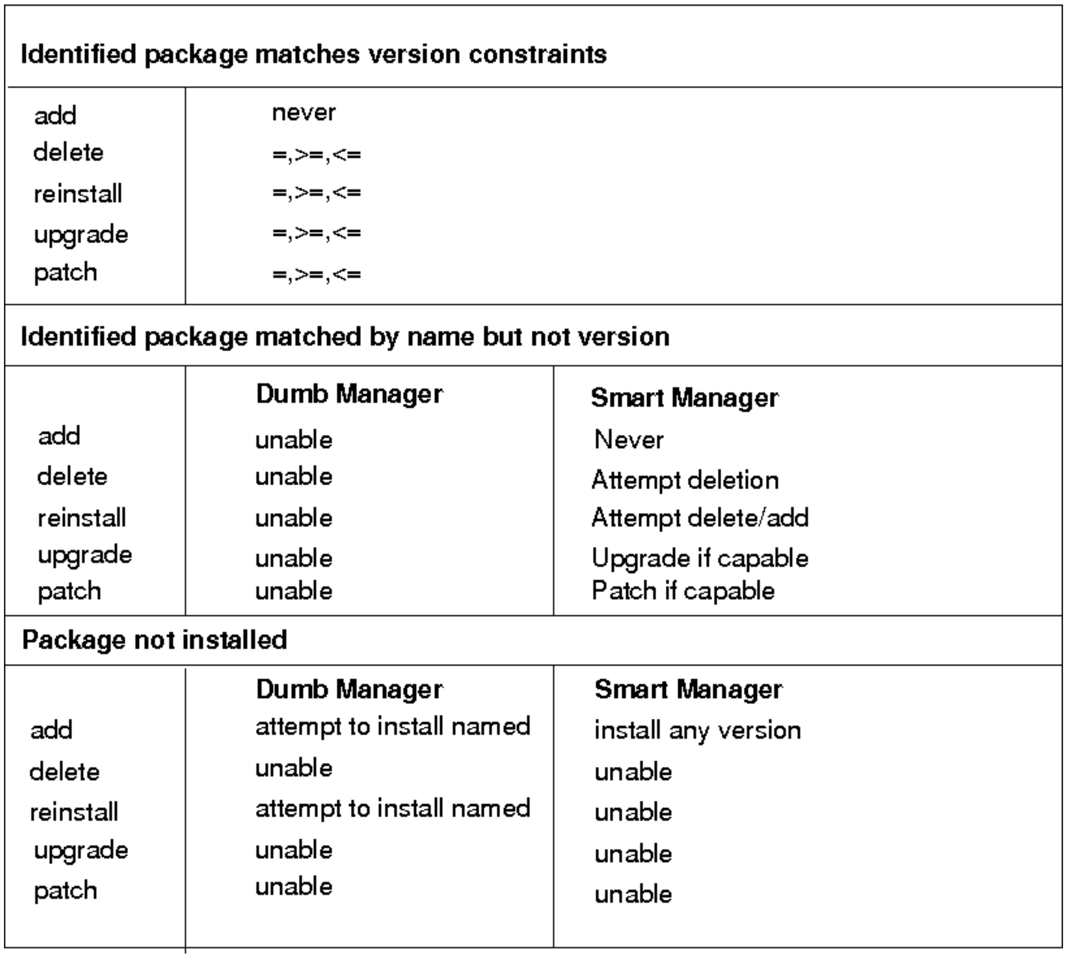
- Smart package systems tha resolve dependencies and require only a symbolic package name.
- Dumb package managers that do not resolve dependencies and need filename input.

Normal ordering for packages is the following:

- Delete
- Add
- Upgrade
- Patch

Promise repair logic

We can discuss package promise repair in the following table.



Identified package matches version constraints		
add	never	
delete	=,>=,<=	
reinstall	=,>=,<=	
upgrade	=,>=,<=	
patch	=,>=,<=	
Identified package matched by name but not version		
	Dumb Manager	**Smart Manager**
add	unable	Never
delete	unable	Attempt deletion
reinstall	unable	Attempt delete/add
upgrade	unable	Upgrade if capable
patch	unable	Patch if capable
Package not installed		
	Dumb Manager	**Smart Manager**
add	attempt to install named	install any version
delete	unable	unable
reinstall	attempt to install named	unable
upgrade	unable	unable
patch	unable	unable

```
bundle agent packages
{
vars:

 # Test the simplest case -- leave everything to the yum smart manager

 "match_package" slist => {
                          "apache2",
                          "apache2-mod_php5",
                          "apache2-prefork",
                          "php5"
                          };
packages:
  "$(match_package)"

     package_policy => "add",
     package_method => yum;

}
```

Packages promises can be very simple if the package manager is of the smart variety that handles details for you. If you need to specify architecture and version numbers of packages, this adds some complexity, but the options are flexible and designed for maximal adaptability.

7.9.1 `package_policy`

Type: (menu option)
Allowed input range:

```
               add
               delete
               reinstall
               update
               patch
               verify
```

Synopsis: Criteria for package installation/upgrade on the current system
Example:

```
packages:

  "$(match_package)"

     package_policy => "add",
     package_method => "xyz";
```

Notes:

This decides what fate is intended for the named package.

7.9.2 `package_method` (compound body)

Type: (ext body)

'`package_changes`'
  Type: (menu option)
  Allowed input range:

```
individual
bulk
```

Synopsis: Menu option - whether to group packages into a single aggregate command

Example:

```
body package_method rpm

{
package_changes => "bulk";
}
```

Notes:

This indicate whether the package manager is capable of handling package operations in bulk, i.e. with by given multiple arguments. If this is set to 'bulk' then multiple arguments will be passed to the package commands. If set to 'individual' packages will be handled one by one. This might add a significant overhead to the operations, and also affect the ability of the operating system's package manager to handle dependencies.

'package_file_repositories'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: A list of machine-local directories to search for packages

Example:

```
body package_method filebased
{
file_repositories => { "/package/repos1", "/packages/repos2" };
}
```

Notes:

If specified, cfengine will assume that the package installation occurs by filename and will search the named paths for a package matching the pattern `package_name_convention`. If found the name will be prefixed to the package name in the package commands.

'package_list_command'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to obtain a list of installed packages

Example:

```
body package_method rpm

{
package_list_command => "/bin/rpm -qa --queryformat \"%{name} %{version}-%{release}\n\"";
}
```

Notes:

This command should provide a complete list of the packages installed on the system. It might also list packages that are not installed. Those should be filtered out using the `package_installed_regex`.

'`package_list_version_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package version string

Example:

```
body package_method rpm

{
package_list_version_regex => "[^\s]+ ([^.]+).*";
}
```

Notes:

This regular expression should containe exactly one back-reference (parenthesis) that marks the version string of packages listed as installed.

'`package_list_name_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package name string

Example:

```
body package_method rpm
```

```
{
package_list_name_regex    => "([^\s]+).*";
}
```

Notes:

This regular expression should contain a single back reference (parenthesis) that marks the name of the package from the package listing.

'package_list_arch_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package architecture string

Example:

```
body package_method rpm
{
package_list_arch_regex    => "[^|]+\|[^|]+\|[^|]+\|[^|]+\|\s+([^\s]+).*";
}
```

Notes:

A regular expression that contains exactly one back reference (parenthesis) which marks the location in the listed package at which the architecture is specified. If no architecture is specified for the given package manager, then do not define this.

'package_version_regex'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package version string

Example:

```
body package_method rpm
{
package_version_regex => "[^\s]+ ([^.]+).*";
}
```

Notes:

If the version of a package is not specified separately using `package_version`, then this should be a regular expression that contains exactly one back-reference that matches the version string in the promiser.

'`package_name_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package name string

Example:

```
body package_method rpm
{
package_name_regex => "([^\s]).*";
}
```

Notes:

This regular expression is only used when the *promiser* contains not only the name of the package, but its version and archiecture also. In that case, this expression should contain a single back-reference (parenthesis) to extract the name of the package from the string.

'`package_arch_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression with one backreference to extract package architecture string

Example:

```
body package_method rpm

{
package_list_arch_regex    => "[^.]+\.([^.]+)";
}
```

Notes:

This is for use when extracting architecture from the name of the promiser, i.e. when the architecture is not specified using the `package_architectures` list. It is a regular expression that contains exactly one back reference (parenthesis) which marks the location in the *promiser* at which the architecture is specified. If no architecture is specified for the given package manager, then do not define this.

'`package_installed_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression which matches packages that are already installed

Example:

```
body package_method yum
{
package_installed_regex => ".*installed.*";
}
```

Notes:

This regular expression should match lines in the output of the list command that are actually installed packages. If all the lines match then it can be set of '`.*`', however most package systems output prefix lines and a variety of human padding that needs to be ignored.

'`package_add_command`'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to install a package to the system

Example:

```
body package_method rpm
{
package_add_command => "/bin echo /bin/rpm -i ";
}
```

Notes:

This command should install a package when appended with the package referernce id, formed using the `package_name_convention`, using the model of (name,version,architecture).

'package_delete_command'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to remove a package from the system

Example:

```
body package_method rpm

{
package_delete_command => "/bin/rpm -e --nodeps";
}
```

Notes:

The command that deletes a package from the system when appended with the package reference identifier specified by `package_name_convention`.

'package_update_command'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to update to the latest version a currently installed package

Example:

```
body package_method zypper
{
package_update_command => "/usr/bin/zypper -non-interactive update";
}
```

Notes:

If supported this should be a command that updates the version of a single currently installed package. If only bulk updates are supported, consider running this as a single command under `commands`.

'package_patch_command'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to update to the latest patch release of an installed package

Example:

```
body package_method zypper

{
package_patch_command => "/usr/bin/zypper -non-interactive patch";
}
```

Notes:

If the package manager supports patching, this command should patch a named package. If only patching of alll packages is supported then consider running that as a batch operation in `commands`.

'`package_verify_command`'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Command to verify the correctness of an installed package

Example:

```
body package_method rpm

{
package_verify_command => "/bin/rpm -V";
}
```

Notes:

If available, this is a command to verify an already installed package. Such commands are not necessarily meaningful in the context of a tool like cfengine which patches the system by 'other means'.

'`package_noverify_regex`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression to match verification failure output

Example:

```
body package_method xyz

{
```

```
package_noverify_regex => ".*problem.*";
}
```

Notes:

A regular expression to match output from a package verification command. If the ourput string matches this expression, the package is deemed broken.

'`package_noverify_returncode`'

Type: int

Allowed input range: `-99999999999,9999999999`

Synopsis: Integer return code indicating package verification failure

Example:

```
body package_method xyz
{
package_noverify_returncode => "-1";
}
```

Notes:

For use if a package verification command uses the return code as the signal for a failed package verification.

'`package_name_convention`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: This is how the package manager expects the file to be referred to, e.g. $(name).$(arch)

Example:

```
body package_method rpm

{
package_name_convention => "$(name).$(arch).rpm";
}
```

Notes:

This sets the pattern for naming the package in the way expected by the package manager. Three special variables are defined from the extracted data, in a private context for use: '`$(name)`', '`$(version)`' and '`$(arch)`'.

7.9.3 package_version

Type: string

Allowed input range: (arbitrary string)

Synopsis: Version reference point for determining promised version

Example:

```
packages:

  "mypackage"

     package_policy => "add",
     package_method => rpm,
     package_select => "=",
     package_version => "1.2.3";
```

Notes:

Used for specifying the targeted package version when the version is written separately from the name of the command.

7.9.4 package_architectures

Type: slist

Allowed input range: (arbitrary string)

Synopsis: Select the architecture for package selection

Example:

```
packages:

  "$(exact_package)"

     package_policy => "add",
     package_method => rpm,
     package_architectures => { "x86_64" };
```

Notes:

It is possible to specify a list of packages of different architectures if it is desirable to install multiple architectures on the host. If no value is specified, cfengine makes no promise about the result; the package manager's behaviour prevails.

7.9.5 `package_select`

Type: (menu option)

Allowed input range:

```
               >
               <
               ==
               !=
               >=
               <=
```

Synopsis: A criterion for first acceptable match relative to "package_version"

Example:

```
packages:

  "$(exact_package)"

     package_policy => "add",
     package_method => xyz,
     package_select => ">=",
     package_architectures => { "x86_64" },
     package_version => "1.2.3-456";
```

Notes:

This selects the operator that compares the promiser to the state of the system packages currently installed. If the criterion matches, the policy action is scheduled for promise-keeping.

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.10 `processes` promises in '`agent`'

Process promises refer to items in the system process table. Note that this is not the same as commands (which are instructions). A process is a command in some state of execution (with a Process Control Block). Promiser objects here are patterns that match line fragments in the system process table.

```
processes:

  "regex contained in process line"

      process_select => process_filter_body,
      restart_class => "activation class for process",
      ..;
```

In cfengine 2 there was a restart clause for directly executing a command to restart a process. In cfengine 3 there is instead a class to activate. You must then desribe a `command` in that class to restart the process.

```
commands:

  restart_me::

   "/path/executable" ... ;
```

This rationalizes complex restart-commands and avoids unnecessary overlap between `processes` and `commands`.

The `process_stop` is also arguably a command, but it should be an ephemeral command that does not lead to a persistent process. It is intended only for commands of the form '`/etc/inetd service stop`', not for processes that persist. Processes are restarted at the end of a bundle's execution, but stop commands are executed immediately.

Take care to note that process table formats differ between operating systems, and the use of simple patterns such as program-names is recommended. For more sophisticated matches, users should use the `process_select` *feature.*

Note: `process_select` was previously called process `filters` in cfengine 2 and earlier.

```
bundle agent example
{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:

 any_procs::

   "Found processes out of range";
}

########################################################

body process_select proc_finder

{
stime_range => irange(ago(0,0,0,5,30,0),ago(0,0,0,0,20,0));
process_result => "stime";
}

########################################################
```

```
body process_count anyprocs

{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

In cfengine 2, one has two separate actions:

```
processes
shellcommands
```

In cfengine 3 we have

```
processes
commands
```

Cfengine 2 got this ontology about right intuitively, but not quite. It allowed a '`restart`' command to appear in a process promise, which is really a command execution. This has been changed in cfengine 3 so that there is a cleaner separation. Let's see why.

Executions are about jobs, services, scripts etc. They are properties of an executable file. The referring 'promiser' is a file object. On the other hand a process is a property of a "process identifier" which is a kernel instantiation, a quite different object altogether. So it makes sense to say that

- A "PID" (which is not an executable) promises to be reminded of a signal, e.g.

  ```
  kill signal pid
  ```

- An "command" promises to start or stop itself with a parameterized specification.

  ```
  exec command argument1 argument2 ...
  ```

Neither the file nor the pid necessarily promise to respond to these activations, but they are nonetheless physically meaningful phenomena or attributes associated with these objects.

- Executable files do not listen for signals as they have no active state.
- PIDs do not run themselves or stop themselves with new arguments, but they can use signals as they are running.

Executions lead to processes for the duration of their lifetime, so these two issues are related, although the promises themselves are not.

Services verus processes:

A service is an abstraction that requires processes to run and files to be configured. It makes a lot of sense to wrap services in modular bundles. Starting and stopping a service can be handled in at least two ways. Take the web service as an example.

We can start the service by promising an execution of a daemon (e.g. `httpd`). Normally this execution does not terminate without intervention. We can terminate it in one of two ways:

- Using a process signal, by promising a signal to processes matching a certain pid search
- Using an execution of a termination command, e.g. '`/etc/init.d/apache stop`'.

The first case makes sense if we need to qualify the termination by searching for the processes. The processes section of a cfengine 3 policy includes a control promise to search for matching processes. If matches are found, signals can be sent to precisely each specific process.

Classes can also be defined, in principle triggering an execution of the stop script, but then the class refers only to the presence of matching pids, not to the individual pids concerned. So it becomes the responsibility of the execution to locate and interact with the pids necessary.

Want it running?:

How do we say simply that we want a service running? In the agent control promises, we could check each service individually.

```
bundlesequence => { Update, Service("apache"), Service("nfsd") };
```

or

```
bundlesequence => { Update, @(globals.all_services)  };
```

The bundle for this can look like this:

```
bundle agent Service("$(service)")

{
processes:

  "$(service)"

      process_count => up("$(service)");

commands:

   "$daemons[$(service)]"

      ifvarclass => "$(service)_up",
      args       => "$args[$(service)]";

}
```

An alternative would be self-contained:

```
bundle agent Service

{
vars:

  "service" slist => { "apache", "nfsd", "bind" };

processes:

  "$(service)"

      process_count => up("$(service)");

commands:

   "$daemons[$(service)]"
```

```
      ifvarclass => "$(service)_up",
      args       => "$args[$(service)]";

}

######################
# Parameterized body
######################

body process_count("$(s)")

{
match_range => "[0,10]";
out_of_range_define => "$(s)_up";
}
```

Is this a step backwards? The cfengine 3 approach might seem like a step backwards from the simple cfengine 2 statement:

```
processes:

  "httpd" restart "/etc/init.d/apache restart"
```

However, it allows several improvements.

You can do other things in between stopping and starting the service, like file editing, or security sweeps. You can use templates to simplify the syntax in bulk for several process checks or restarts.

```
processes:

  "$(service.list)"
```

If you don't want any delay in stopping and starting the service, then place these promises in a private bundle with nothing in between them.

7.10.1 signals

Type: (option list)

Allowed input range:

```
               hup
               int
               trap
               kill
               pipe
               cont
               abrt
               stop
               quit
               term
               child
               usr1
               usr2
               bus
               segv
```

Synopsis: A list of menu options representing signals to be sent to a process

Example:

```
processes:

 cfservd_out_of_control::

   "cfservd"

        signals         => { "stop" , "term" },
        restart_class   => "start_cfserv";

 any::

   "snmpd"

        signals         => { "term" , "kill" };
```

Notes:

Signals are presented as an ordered list until the first one succeeds.

7.10.2 `process_stop`

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: A command used to stop a running process

Example:

```
processes:

 "snmpd"

        process_stop => "/etc/init.d/snmp stop";
```

Notes:

As an alternative to sending a termination or kill signal to a process, one may call a 'stop script' to perform a graceful shutdown.

7.10.3 process_count (compound body)

Type: (ext body)

'match_range'

Type: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Integer range for acceptable number of matches for this process

Example:

```
body process_count example
{
match_range => irange("10","50");
}
```

Notes:

This is a numerical range for the number of occurrences of the process in the process table. As long as it falls within the specified limits, the promise is considered kept.

'in_range_define'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of classes to define if the matches are in range

Example:

```
body process_count example
{
in_range_define => { "class1", "class2" };
}
```

Notes:

Classes are defined if the processes that are found in the process table satisfy the promised process count, i.e. if the promise about the number of processes matching the other criteria is kept.

'out_of_range_define'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of classes to define if the matches are out of range

Example:

```
body process_count example(s)
{
out_of_range_define => { "process_anomaly", "anomaly_$(s)"};
}
```

Notes:

Classes to activate remedial promises conditional on this promise failure to be kept.

7.10.4 process_select (compound body)

Type: (ext body)

'process_owner'

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of regexes matching the user of a process

Example:

```
body process_select example
{
process_owner => { "wwwrun", "nobody" };
}
```

Notes:

Regular expression should match a legal user name on the system.

'pid' **Type**: irange [int,int]

Allowed input range: 0,99999999999

Synopsis: Range of integers matching the process id of a process

Example:

```
body process_select example
{
ppid => irange("1","10");
process_result => "ppid";
}
```

Notes:

'ppid' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the parent process id of a process

Example:

```
body process_select example
{
ppid => irange("407","511");
}
```

Notes:

'pgid' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the parent group id of a process

Example:

```
body process_select example
{
pgid => irange("1","10");
process_result => "pgid";
}
```

Notes:

'rsize' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the resident memory size of a process

Example:

```
body process_select
{
rsize => irange("4000","8000");
}
```

Notes:

'vsize' **Type**: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the virtual memory size of a process

Example:

```
body process_select example
{
vsize => irange("4000","9000");
}
```

Notes:

'status' **Type**: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the status field of a process

Example:

```
body process_select example
```

```
{
status => "Z";
}
```

Notes:

For instance, characters in the set ‘NRS<sl+..’

‘ttime_range’

Type: irange [int,int]

Allowed input range: 0,4026531839

Synopsis: Range of integers matching the total elapsed time of a process

Example:

```
body process_select example
{
ttime_range => irange(0,accumulated(0,1,0,0,0,0));
}
```

Notes:

This is total accumulated time for a process.

‘stime_range’

Type: irange [int,int]

Allowed input range: 0,4026531839

Synopsis: Range of integers matching the start time of a process

Example:

```
body process_select example
{
stime_range => irange(ago(0,0,0,1,0,0,),now);
}
```

Notes:

The calculation of time from process table entries is sensitive to Daylight Savings Time (Summer/Winter Time) so calculations could be a hour off. This is for now a bug to be fixed.

‘command’ **Type**: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the command/cmd field of a process

Example:

```
body select_process example

{
command => "cf-.*";

process_result => "command";
}
```

Notes:

This expression should match the entire `COMMAND` field of the process table (not just a fragment). This field is usually the last field on the line and thus starts with the first non-space character and ends with the end of line.

‘tty’ **Type**: string

Allowed input range: (arbitrary string)

Synopsis: Regular expression matching the tty field of a process

Example:

```
body process_select example
{
tty => "pts/[0-9]+";
}
```

Notes:

‘priority’ **Type**: irange [int,int]

Allowed input range: `-20,+20`

Synopsis: Range of integers matching the priority field (PRI/NI) of a process

Example:

```
body process_select example
{
priority => irange("-5","0");
}
```

Notes:

`threads`

Type: irange [int,int]

Allowed input range: `0,99999999999`

Synopsis: Range of integers matching the threads (NLWP) field of a process

Example:

```
body process_select example
{
threads => irange(1,5);
}
```

Notes:

`process_result`

Type: string

Allowed input range: `[(process_owner|pid|ppid||pgid|rsize|vsize|status|command|ttime|stime|t`

Synopsis: Boolean class expression returning the logical combination of classes set by a process selection test

Example:

```
body process_select proc_finder(p)

{
process_owner  => { "avahi", "bin" };
command        => "$(p)";
pid            => "100,199";
vsize          => "0,1000";
process_result => "command.(process_owner|rsize)";
}
```

Notes:

A logical combination of the process selection classifiers. The syntax is the same as that for class expressions. There should be no spaces in the expressions.

7.10.5 `restart_class`

Type: string

Allowed input range: `[a-zA-Z0-9_$.]+`

Synopsis: A class to be set if the process is not running, so that a command: rule can be referred to restart the process

Example:

```
processes:

   "cfservd"

     restart_class => "start_cfserv";

commands:

  start_cfserv::

    "/usr/local/sbin/cfservd";
```

Notes:

This is a signal to restart a process that should be running, if it is not running. Processes are signalled first and then restarted later, at the end of bundle execution, after all possible corrective actions have been made that could influence their execution.

```
bundle agent main(parameter)

{
vars:

  "sys_files"    slist        => {
                                 "/etc/passwd",
                                 "/etc/services"
                                 };
files:

  "$(sys_files)" perms        => p("root","0644"),
                 changes      => trip_wire;

  "/etc/shadow"  perms        => p("root","0600"),
                 changes      => trip_wire;

  "/usr"         changes      => trip_wire,
                 depth_search => recurse("inf");

  "/tmp"         delete       => tidy,
                 file_select  => days("2"),
                 depth_search => recurse("inf");

}
```

Agent bundles contain user-defined promises for `cf-agent`. The types of promises and their corresponding bodies are detailed below.

7.11 `storage` promises in ‘`agent`’

Storage promises refer to disks and filesystem properties.

```
storage:

   "/disk volume or mountpoint"

     volume => volume_body,
     ...;
```

In cfengine 2, storage promises were divided into `disks` or `required`, and `misc_mounts` types. The old mount-models for binary and home servers has been deprecated and removed from cfengine 3. Users who use these models can reconstruct them from the low-level tools.

`bundle agent storage`

```
{
storage:

  "/usr" volume  => mycheck("10%");
  "/mnt" mount   => nfs("nfsserv.example.org","/home");

}

#######################################################

body volume mycheck(free)   # reusable template

{
check_foreign  => "false";
freespace      => "$(free)";
sensible_size  => "10000";
sensible_count => "2";
}

body mount nfs(server,source)

{
mount_type => "nfs";
mount_source => "$(source)";
mount_server => "$(server)";
edit_fstab => "true";
}
```

7.11.1 `mount` (compound body)

Type: (ext body)

'`mount_type`'

Type: (menu option)

Allowed input range:

```
nfs
nfs2
nfs3
nfs4
```

Synopsis: Protocol type of remote file system

Example:

```
body mount example
{
mount_type => "nfs3";
}
```

Notes:

This field is mainly for future extensions.

'mount_source'

Type: string

Allowed input range: `[cC]:\\.*|/.*`

Synopsis: Path of remote file system to mount

Example:

```
body mount example
{
mount_source "/location/disk/directory";
}
```

Notes:

This is the location on the remote device, server, SAN etc.

'mount_server'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Hostname or IP or remote file system server

Example:

```
body mount example
{
mount_server => "nfs_host.example.org";
}
```

Notes:

Hostname or IP address, this could be on a SAN.

'mount_options'

Type: slist

Allowed input range: `List of option strings to add to the file system table ("fstab")`

Synopsis: (null)

Example:

```
body mount example
{
mount_options => { "rw", "acls" };
}
```

Notes:

This list is concatenated in a form appropriate for the filesystem. The options must be legal options for the system mount commands.

'edit_fstab'

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false add or remove entries to the file system table ("fstab")

Example:

```
body mount example
{
edit_fstab => "true";
}
```

Notes:

The default behaviour is to not place edits in the file system table.

'unmount' **Type**: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false unmount a previously mounted filesystem

Example:

```
body mount example
{
unmount => "true";
}
```

Notes:

7.11.2 volume (compound body)

Type: (ext body)

'check_foreign'

Type: (menu option)

Allowed input range:

```
          true
          false
          yes
          no
          on
          off
```

Synopsis: true/false verify storage that is mounted from a foreign system on this host

Example:

```
body volume example

{
#..
check_foreign  => "false";
}
```

Notes:

Cfengine will not normally perform sanity checks on filesystems which are not local to the host. If `true` it will ignore a partition's network location and ask the current host to verify storage located physically on other systems.

'`freespace`'

Type: string

Allowed input range: `[0-9]+[mb%]`

Synopsis: Absolute or percentage minimum disk space that should be available before warning

Example:

```
body volume example1
{
freespace => "10%";
}

body volume example2
{
freespace => "50M";
}
```

Notes:

The amount of freespace that is promised on a storage device. Once this promise is found not to be kept, warnings are generated.

'`sensible_size`'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum size in bytes that should be used on a sensible-looking storage device

Example:

```
body volume example
{
sensible_size => "20K";
}
```

Notes:

```
body volume control
{
sensible_size => "20K";
}
```

'sensible_count'

Type: int

Allowed input range: `0,99999999999`

Synopsis: Minimum number of files that should be defined on a sensible-looking storage device

Example:

```
body volume example
{
sensible_count => "20";
}
```

Notes:

Files must be readable by the agent, i.e. it is assumed that the agent has privileges on volumes being checked.

'scan_arrivals'

Type: (menu option)

Allowed input range:

```
true
false
yes
no
on
off
```

Synopsis: true/false generate pseudo-periodic disk change arrival distribution

Example:

```
body volume example
{
scan_arrivals => "true";
}
```

Notes:

This operation should not be left ‘on’ for more than a single run (maximum once per week). It causes cfengine to perform an extensive disk scan noting the schedule of changes between files. This can be used for a number of analyses including optimum backup schedule computation.

8 Bundles of server

```
bundle server access_rules()

{
access:

  "/home/mark/PrivateFiles"

    admit   => { ".*\.example\.org" };

  "/home/mark/.cfagent/bin/cf-agent"

    admit   => { ".*\.example\.org" };

roles:

  ".*"  authorize => { "mark" };
}
```

Bundles in the server describe access promises on specific file and class objects supplied by the server to clients.

8.1 access promises in 'server'

Access promises are conditional promises made by the server about file objects. The promise has two consequences. For file copy requests, the file becomes transferrable to the remote client according to the conditions specified in the server promse (i.e. if the connection encryption requirements are met, and if the client has been granted appropriate privileges with maproot (like its NFS counterpart) to be able to see file objects not owned by the server process owner).

The promise has two mutally exclusive attributes 'admit' and 'deny'. Use of 'admit' is preferred as mistakes and omissions can easily be made when excluding from a group.

When access is granted to a directory, the promise is automatically given about all of its contents and sub-directories. The access promise allows overlapping promises to be made, and these are kept in a first-come-first-served fashion. Thus file objects (promisers) should be listed in order of most-specific file first. In this way, specific promises will override less specific ones.

```
access:

   "/path/file_object"

     admit   => { "hostname", "ipv4_address", "ipv6_address"  };
```

8.1.1 Access Example

```
#########################################################
# Server config
#########################################################

body server control

{
allowconnects         => { "127.0.0.1" , "::1" };
allowallconnects      => { "127.0.0.1" , "::1" };
trustkeysfrom         => { "127.0.0.1" , "::1" };
}

#########################################################

bundle server access_rules()

{
access:

  "/home/mark/LapTop"

    admit   => { "127.0.0.1" };
}
```

Entries may be literal addresses of IPv4 or IPv6, or any name registered in the POSIX `gethostbyname` service.

8.1.2 `admit`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to grant access to file objects

Example:

```
access:

  "/home/mark/LapTop"

    admit   => { "127.0.0.1", "192.168.0.1/24", ".*.domain.tld"  };
```

Notes:

Admit promises grant access to file objects on the server. Arguments may be IP addresses or hostnames, provided DNS name resolution is active. In order to reach this stage, a client must first have passed all of the standard connection tests in the control body.

The lists may contain network addresses in CIDR notation or regular expressions to match the IP address or name of the connecting host.

8.1.3 deny

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to deny access to file objects

Example:

```
bundle server access_rules()

{
access:

  "/path"

    admit   => { ".*\.example\.org" },
    deny    => { "badhost_1\.example\.org", "badhost_1\.example\.org" };
}
```

Notes:

Denial is for special exceptions. A better strategy is always to grant on a need to know basis. A security policy based on exceptions is a weak one.

8.1.4 maproot

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of host names or IP addresses to grant full read-privilege on the server

Example:

```
access:

 "/home"

       admit => { "backup_host.example.org" },
 ifencrypted => "true",

     # Backup needs to have access to all users

     maproot => { "backup_host.example.org" };
```

Notes:

Normally users authenticated by the server are granted access only to files owned by them and no-one else. Even if the `cf-serverd` process runs with root privileges on the server side of a client-server connection, the client is not automatically granted access to download files owned by non-privileged users. If `maproot` is true then remote `root` users are granted access to all files.

A typical case where mapping is important is in making backups of many user files.

8.1.5 `ifencrypted`

Type: (menu option)

Allowed input range:

```
               true
               false
               yes
               no
               on
               off
```

Synopsis: true/false whether the current file access promise is conditional on the connection from the client being encrypted

Example:

```
access:

   "/path/file"
```

```
    admit     => { ".*.example.org" },
    ifencrypted => "true";
```

Notes:

If this flag is true a client cannot access the file object unless its connection is encrypted.

8.2 roles promises in 'server'

Roles promises are server-side decisions about which users are allowed to define soft-classes on the server's system during remote invocation of `cf-agent`. This implements a form of Role Based Access Control (RBAC) for pre-assigned class-promise bindings. The user names cited must be attached to trusted public keys in order to be accepted.

```
roles:

  "regex"

     authorize => { "usernames", ... };
```

It is worth re-iterating here that it is not possible to send commands or modify promise definitions by remote access. At best users may try to send classes when using `cf-runagent` in order to activate sleeping promises. This mechanism limits their ability to do this.

```
bundle server access_rules()

{
roles:

  # Allow mark

  "Mark_.*"  authorize => { "mark" };
}
```

In this example user '`mark`' is granted permission to remotely activate classes matching the regular expression when using the `cf-runagent` to activate cfengine. In this way one can implement a form of Role Based Access Control (RBAC), provided users do not have privileged access on the host directly.

8.2.1 `authorize`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of public-key user names that are allowed to activate the promised class during remote agent activation

Example:

```
roles:

  ".*"  authorize => { "mark", "marks_friend" };
```

Notes:

Part of Role Based Access Control (RBAC) in cfengine. The users listed in this section are granted access to set certain classes by using the remote `cf-runagent`. The user-names will refer to public key identities already trusted on the system.

9 Bundles of knowledge

```
bundle knowledge system

{
topics:

 Troubleshooting::

  "Segmentation fault"
       association => a("is caused by","Bad memory reference","can cause");

  "Remote connection problem";
  "Web server not running";
  "Print server not running";
  "Bad memory reference";
}
```

Knowledge bundles describe topic maps, i.e. Topics, Associations and Occurrences (of topics in documents). This is for knowledge modelling and has no functional effect on a system.

9.1 topics promises in 'knowledge'

Topic promises are part of the knowledge management engine. A topic is any string that refers to a concept or subject that we wish to include in a knowledge base. If a topic has a very long name, it is best to made the promiser object a short name and use the `comment` field to add the long explanation (e.g. unique acronym and full text).

```
 topics:

  "topic string"

   comment => "long name..",
   ...;
```

Topics form associative structures based entirely on an abstract space of natural language. Actually, this is only slightly more abstract than files, processes and commands etc. The main difference in knowledge management is that there are no corrective or maintenance operations associated with knowledge promises.

Class membership in knowledge management is subtly different from other parts of cfengine. If a topic lies in a certain class context, the topic uses it as a type-label. This is used for disambiguation of subject-area in searches rather than for disambiguation of rules between physical environments.

```
bundle knowledge example
{
topics:

   "Distro"
      comment     => "Distribution of linux",
      association => a("is a packaging of","Linux","is packaged as a");
}
```

Topics are basically identifiers, where the comment field here is a long form of the subject string. Associations form semantic links between topics. Topics can appear multiple times in order to form multiple associations.

9.1.1 `association` (compound body)

Type: (ext body)

'`forward_relationship`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of forward association between promiser topic and associates

Example:

```
body association example
{
forward_relation => "is bigger than";
}
```

Notes:

'`backward_relationship`'

Type: string

Allowed input range: (arbitrary string)

Synopsis: Name of backward/inverse association from associates to promiser topic

Example:

```
body association example
{
# ..
backward_relationship => "is less than";
}
```

Notes:

Denotes the inverse name which is used to 'moralizing' the association graph.

`'associates'`

Type: slist

Allowed input range: (arbitrary string)

Synopsis: List of associated topics by this forward relationship

Example:

```
body association example(literal,scalar,list)

{
#...
associates => { "literal", $(scalar),  @(list)};
}
```

Notes:

An element of an association which is a list of topics to which the current topic is associated.

9.1.2 `comment`

Type: string

Allowed input range: (arbitrary string)

Synopsis: Retained comment about this promise's real intention

Example:

```
comment => "This comment follows the data for reference ...",
```

Notes:

Comments written in code follow the program, they are not merely discarded. They appear in reports and error messages.

9.2 `occurrences` promises in '`knowledge`'

Occurrences are documents or information resources that discuss topics. An occurrence promise asserts that a particular document of text resource in fact represents information about one or more topics. This is used to construct references to actual information in a topic map.

```
occurrences:

  topic_name::

    "URL reference or literal string"

        represents => { "sub-topic disambiguator", ... },
        representation => "literal or url";
```

```
Mark_Burgess::

    "http://www.iu.hio.no/~mark"
           represents => { "Home Page" };

lvalue::

    "A variable identifier, i.e. the left hand side of an '=' association. The promiser in a variab
           represents => { "Definitions" },
           representation => "literal";

Editing_Files::

"http://www.cfengine.org/confdir/customizepasswd.html"
  represents => { "Setting up users" };
```

Occurrences are pointers to information about topics. This might be a literal text string or a URL reference to an external document.

9.2.1 `represents`

Type: slist

Allowed input range: (arbitrary string)
Synopsis: List of subtopics that disambiguate the context of this reference
Example:

```
occurrences:

  Promise_Theory::

    "A theory of autonomous actors that offer certainty through promises"

      represents     => { "Definitions" },
      representation => "literal";
```

Notes:

The sub-topic represented by the document reference in a knowledge base.

9.2.2 representation

Type: (menu option)
Allowed input range:

```
literal
url
db
file
web
```

Synopsis: How to interpret the promiser string e.g. actual data or reference to data
Example:

```
occurrences:

  Promise_Theory::

    "A theory of autonomous actors that offer certainty through promises"

      represents     => { "Definitions" },
      representation => "literal";
```

Notes:

The form of knowledge representation in a topic map occurrence reference.

10 Special functions

10.1 Function accessedbefore

Synopsis: accessedbefore(2 args) returns type class
True if arg1 was accessed before arg2 (atime)
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { accessedbefore("/tmp/earlier","/tmp/later"), "linux" };

reports:

  do_it::

    "The secret changes have been accessed after the reference time";

}
```

Notes:

The function accesses the `atime` fields of a file and makes a comparison.

```
touch /tmp/reference
touch /tmp/secretfile

/usr/local/sbin/cf-agent -f ./unit_accessed_before.cf -K
R: The secret changes have been accessed after the reference time
```

10.2 Function accumulated

Synopsis: accumulated(6 args) returns type int

Convert an accumulated amount of time into a system representation

Example:

```
bundle agent testbundle

{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:

 any_procs::

   "Found processes in range";
}

########################################################

body process_select proc_finder

{
ttime_range => irange(accumulated(0,0,0,0,2,0),accumulated(0,0,0,0,20,0));
process_result => "ttime";
}

########################################################

body process_count anyprocs

{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

Notes:

In the example we look for processes that have accumulated between 2 and 20 minutes of total run time.

ARGUMENTS:

'Years'	The year, e.g. 2009
'Month'	The Month, 1-12
'Day'	The day 1-31
'Hours'	The hour 0-23
'Minutes'	The minutes 0-59
'Seconds'	The number of seconds 0-59

10.3 Function ago

Synopsis: ago(6 args) returns type int

Convert a time relative to now to an integer system representation

Example:

```
bundle agent testbundle

{
processes:

 ".*"

    process_count   => anyprocs,
    process_select  => proc_finder;

reports:

 any_procs::

   "Found processes out of range";
}

########################################################

body process_select proc_finder

{
stime_range => irange(ago(0,0,0,5,30,0),ago(0,0,0,0,20,0));
process_result => "stime";
}

########################################################

body process_count anyprocs
```

```
{
match_range => "0,0";
out_of_range_define => { "any_procs" };
}
```

Notes:

The ago function measures time relative to now.

ARGUMENTS:

‘Years’	The year, e.g. 2009
‘Months’	The Month, 1-12
‘Days’	The day 1-31
‘Hours’	The hour 0-23
‘Minutes’	The minutes 0-59
‘Seconds’	The number of seconds 0-59

10.4 Function canonify

Synopsis: canonify(1 args) returns type string

Convert an abitrary string into a legal class name

Example:

```
commands:

   "/var/cfengine/bin/$(component)"

       ifvarclass => canonify("start_$(component)");
```

Notes:

This is for use in turning arbitrary text into class data.

10.5 Function changedbefore

Synopsis: changedbefore(2 args) returns type class

True if arg1 was changed before arg2 (ctime)

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { changedbefore("/tmp/earlier","/tmp/later"), "linux" };

reports:

  do_it::

    "The derived file needs updating";

}
```

Notes:

Change times include both file permissions and file contents. Comparisons like this are normally used for updating files (like the 'make' command).

10.6 Function classify

Synopsis: classify(1 args) returns type class
True if the canonicalization of the argument is a currently defined class
Example:

```
classes:

 "i_am_the_policy_host" expression => classify("master.example.org");
```

Notes:

This function returns true of the classification (canonical form) of the argument is already a defined class. This is useful for transforming variables into classes for instance. See also `canonify()`.

10.7 Function classmatch

Synopsis: classmatch(1 args) returns type class

True if the regular expression matches any currently defined class

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { classmatch(".*_cfengine_com"), "linux" };

reports:

  do_it::

    "Host matches pattern";

}
```

Notes:

The regular expression is matched against the current list of defined classes.

10.8 Function execresult

Synopsis: execresult(2 args) returns type string

Execute named command and assign output to variable

Example:

```
body common control

{
```

```
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "my_result" string => execresult("/bin/ls /tmp","noshell");

reports:

  linux::

    "Variable is $(my_result)";

}
```

Notes:

The second argument decides whether a shell will be used to encapsulate the command. This is necessary in order to combine commands with pipes etc, but remember that each command requires a new process that reads in files beyond cfengine's control. Thus using a shell is both a performance hog and a potential security issue.

Note: you should never use this function to execute comands that make changes to the system. Such an operation is beyind cfengine's ability to guarantee convergence, and on multiple passes and during syntax verification, these function calls are executed resulting in system changes that are 'covert'. Calls to `execresult` should be for discovery and information extraction only.

10.9 Function fileexists

Synopsis: fileexists(1 args) returns type class
True if the named file can be accessed
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################
```

```
bundle agent example

{
classes:

  "exists" expression => fileexists("/etc/passwd");

reports:

  exists::

    "File exists";

}
```

Notes:

The user must have access permissions to the file for this to work faithfully.

10.10 Function getindices

Synopsis: getindices(1 args) returns type slist

Get a list of keys to the array whose id is the argument and assign to variable

Example:

```
body common control

{
any::

  bundlesequence  => { "testsetvar" };
}


#######################################################

bundle agent testsetvar

{
vars:

  "v[index_1]" string => "value_1";
  "v[index_2]" string => "value_2";
  "parameter_name" slist => getindices("v");

reports:

  Yr2008::

   "Found index: $(parameter_name)";

}
```

Notes:

Make sure you specify the correct scope when supplying the name of the variable.

10.11 Function getgid

Synopsis: getgid(1 args) returns type int

Return the integer group id of the named group on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "gid" int => getgid("users");

reports:

  Yr2008::

    "Users gid is $(gid)";

}
```

Notes:

If the named group does not exist, the variable will not be defined.

10.12 Function getuid

Synopsis: getuid(1 args) returns type int

Return the integer user id of the named user on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "uid" int => getuid("mark");

reports:

  Yr2008::

    "Users gid is $(uid)";

}
```

Notes:

If the named user is not registered the variable will not be defined.

10.13 Function groupexists

Synopsis: groupexists(1 args) returns type class

True if group or numerical id exists on this host

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "gname" expression => groupexists("users");
  "gid"   expression => groupexists("100");

reports:

  gname::

    "Group exists by name";

  gid::

    "Group exists by id";

}
```

Notes:

The group may be specified by name or number.

10.14 Function hash

Synopsis: hash(2 args) returns type string

Return the hash of arg1, type arg2 and assign to a variable

Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "md5" string => hash("Cfengine is not cryptic","md5");

reports:

  Yr2008::

    "Hashed to: $(md5)";

}
```

Notes:

Hash functions are extremely sensitive to input. You should not expect to get the same answer from this function as you would from every other tool, since it depends on how whitespace and end of file characters are handled.

10.15 Function hostrange

Synopsis: hostrange(2 args) returns type class

True if the current host lies in the range of enumerated hostnames specified

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example
```

```
{
classes:

  "compute_nodes" expression => hostrange("cpu-","01-32");

reports:

  compute_nodes::

    "No computer is a cluster";

}
```

Notes:

This is a pattern matching function for non-regular (enumerated) expressions.

10.16 Function hostinnetgroup

Synopsis: hostinnetgroup(1 args) returns type class

True if the current host is in the named netgroup

Example:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/function_ho
""
```

Notes:

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/function_ho
""
```

10.17 Function iprange

Synopsis: iprange(1 args) returns type class

True if the current host lies in the range of IP addresses specified

Example:

```
body common control

{
```

```
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "adhoc_group_1" expression => iprange("128.39.89.10-15");
  "adhoc_group_2" expression => iprange("128.39.74.1/23");

reports:

  adhoc_group_1::

    "Some numerology";

  adhoc_group_2::

    "The masked warriors";
}
```

Notes:

Pattern matching based on IP addresses.

10.18 Function irange

Synopsis: irange(2 args) returns type irange [int,int]

Define a range of integer values for cfengine internal use

Example:

Notes:

Not currently used.

10.19 Function isdir

Synopsis: isdir(1 args) returns type class

True if the named object is a directory

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "isdir" expression => isdir("/etc");

reports:

  isdir::

    "Directory exists..";

}
```

Notes:

The cfengine process must have access to the object concerned in order for this to work.

10.20 Function isgreaterthan

Synopsis: isgreaterthan(2 args) returns type class
True if arg1 is numerically greater than arg2, else compare strings like strcmp
Example:

```
body common control

{
bundlesequence  => { "test"  };
}
```

```
#########################################################

bundle agent test

{
classes:

  "ok" expression => isgreaterthan("1","0");

reports:

  ok::

    "Assertion is true";

 !ok::

  "Assertion is false";

}
```

Notes:

The comparison is made numerically if possible. If the values are strings, the result is identical to that of comparing with '`strcmp()`'.

10.21 Function islessthan

Synopsis: islessthan(2 args) returns type class
True if arg1 is numerically less than arg2, else compare strings like NOT strcmp
Example:

```
body common control

{
bundlesequence  => { "test"  };
}

#########################################################

bundle agent test

{
classes:
  "ok" expression => islessthan("0","1");

reports:

  ok::

    "Assertion is true";

 !ok::

  "Assertion is false";

}
```

Notes:

The complement of `isgreaterthan`. The comparison is made numerically if possible. If the values are strings, the result is identical to that of comparing with '`strcmp()`'.

10.22 Function islink

Synopsis: islink(1 args) returns type class
True if the named object is a symbolic link
Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "isdir" expression => islink("/tmp/link");

reports:

  isdir::
    "Directory exists..";

}
```

Notes:

The link node must both exist and be a symbolic link. Hard links cannot be detected using this function. A hard link is a regular file or directory.

10.23 Function isnewerthan

Synopsis: isnewerthan(2 args) returns type class

True if arg1 is newer (modified later) than arg2 (mtime)

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "do_it" and => { isnewerthan("/tmp/later","/tmp/earlier"), "linux" };

reports:

  do_it::

    "The derived file needs updating";

}
```

Notes:

This function compares the modification time of the file, referring to changes of content only.

10.24 Function isplain

Synopsis: isplain(1 args) returns type class

True if the named object is a plain/regular file

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "isplain" expression => isplain("/etc/passwd");

reports:

  isplain::

    "File exists..";

}
```

Notes:

10.25 Function isvariable

Synopsis: isvariable(1 args) returns type class

True if the named variable is defined

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "bla" string => "xyz..";

classes:

  "exists" expression => isvariable("bla");

reports:

  exists::

    "Variable exists: \"$(bla)\"..";

}
```

Notes:

The variable need only exist. This says nothing about its value. Use `regcmp` to check variable values.

10.26 Function now

Synopsis: now(0 args) returns type int

Convert the current time into system representation

Example:

```
body file_select zero_age
{
mtime       => irange(ago(1,0,0,0,0,0),now);
file_result => "mtime";
}
```

Notes:

10.27 Function on

Synopsis: on(6 args) returns type int

Convert an exact date/time to an integer system representation

Example:

```
body file_select zero_age
{
mtime       => irange(on(2000,1,1,0,0,0),now);
file_result => "mtime";
}
```

Notes:

An absolute date. Note that in process matching dates could be wrong by an hour depending on Daylight Savings Time / Summer Time. This is a known bug to be fixed.

ARGUMENTS:

'Years'	The year, e.g. 2009
'Month'	The Month, 1-12
'Day'	The day 1-31
'Hours'	The hour 0-23
'Minutes'	The minutes 0-59
'Seconds'	The number of seconds 0-59

10.28 Function randomint

Synopsis: randomint(2 args) returns type int

Generate a random integer between the given limits

Example:

```
vars:

 "ran"    int => randomint(4,88);
```

Notes:

The limits must be integer values and the resulting numbers are based on the entropy of the md5 algorithm.

10.29 Function readfile

Synopsis: readfile(2 args) returns type string

Read max number of bytes from named file and assign to variable

Example:

```
vars:

 "xxx"

    string => readfile( "/home/mark/tmp/testfile" , "33" );
```

Notes:

The file (fragment) is read into a single scalar variable.

10.30 Function readintarray

Synopsis: readintarray(6 args) returns type int

Read an array of integers from a file and assign the dimension to a variable

Example:

```
vars:

  "dim_array"

     int =>  readintarray("array_name","/tmp/array","#[^\n]*",":",10,4000);
```

Notes:

Reads a two dimensional array from a file. One dimension is separated by the character specified in the argument, the other by the the lines in the file. The first field of the lines names the first array argument.

```
1: 5:7:21:13
2:19:8:14:14
3:45:1:78:22
4:64:2:98:99
```

Results in

```
array_name[1][0]   1
array_name[1][1]   5
array_name[1][2]   7
array_name[1][3]   21
```

```
      array_name[1][4]   13
      array_name[2][0]   2
      array_name[2][1]   19
      array_name[2][2]   8
      array_name[2][3]   14
      array_name[2][4]   14
      array_name[3][0]   3
      array_name[3][1]   45
      array_name[3][2]   1
      array_name[3][3]   78
      array_name[3][4]   22
      array_name[4][0]   4
      array_name[4][1]   64
      array_name[4][2]   2
      array_name[4][3]   98
      array_name[4][4]   99
```

10.31 Function readintlist

Synopsis: readintlist(5 args) returns type ilist

Read and assign a list variable from a file of separated ints

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "mylist" ilist => { readintlist("/tmp/listofint","#.*","[\n]",10,400) };

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

'filename'	The name of a text file containing text to be split up as a list.
'comment'	A regex pattern which is to be ignored in the file
'split'	A regex pattern which is to be used to split up the file into items
'maxent'	The maximum number of list items to read from the file
'maxsize'	The maximum number of bytes to read from the file

Notes:

10.32 Function readrealarray

Synopsis: readrealarray(6 args) returns type int

Read an array of real numbers from a file and assign the dimension to a variable

Example:

```
vars:

  "dim_array"

     int =>  readrealarray("array_name","/tmp/array","#[^\n]*",":",10,4000);
```

Notes:

See the notes for `readintarray`.

10.33 Function readreallist

Synopsis: readreallist(5 args) returns type rlist

Read and assign a list variable from a file of separated real numbers

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example
```

```
{
vars:

  "mylist" ilist => { readreallist("/tmp/listofreal","#.*","[\n]",10,400) };

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

‘`filename`’	The name of a text file containing text to be split up as a list.
‘`comment`’	A regex pattern which is to be ignored in the file
‘`split`’	A regex pattern which is to be used to split up the file into items
‘`maxent`’	The maximum number of list items to read from the file
‘`maxsize`’	The maximum number of bytes to read from the file

Notes:

10.34 Function readstringarray

Synopsis: readstringarray(6 args) returns type int
Read an array of strings from a file and assign the dimension to a variable
Example:

```
vars:

  "dim_array"

     int =>  readstringarray("array_name","/tmp/array","#[^\n]*",":",10,4000);
```

Returns an integer number of keys in the array.
ARGUMENTS:

‘`array_name`’	The name to be used for the container array.
‘`filename`’	The name of a text file containing text to be split up as a list.
‘`comment`’	A regex pattern which is to be ignored in the file

'split'	A regex pattern which is to be used to split up the file into items
'maxent'	The maximum number of list items to read from the file
'maxsize'	The maximum number of bytes to read from the file

Notes:

Reads a two dimensional array from a file. One dimension is separated by the character specified in the argument, the other by the the lines in the file. The first field of the lines names the first array argument.

```
     at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
     avahi:x:103:105:User for Avahi:/var/run/avahi-daemon:/bin/false
     beagleindex:x:104:106:User for Beagle indexing:/var/cache/beagle:/bin/bash
     bin:x:1:1:bin:/bin:/bin/bash
     daemon:x:2:2:Daemon:/sbin:/bin/bash
```

Results in a systematically indexed map of the file. Some samples are show below to illustrate the pattern.

```
     ...
     array_name[daemon][0]   daemon
     array_name[daemon][1]   x
     array_name[daemon][2]   2
     array_name[daemon][3]   2
     array_name[daemon][4]   Daemon
     array_name[daemon][5]   /sbin
     array_name[daemon][6]   /bin/bash
     ...
     array_name[at][3]       25
     array_name[at][4]       Batch jobs daemon
     array_name[at][5]       /var/spool/atjobs
     array_name[at][6]       /bin/bash
     ...
     array_name[games][3]    100
     array_name[games][4]    Games account
     array_name[games][5]    /var/games
     array_name[games][6]    /bin/bash
     ...
```

10.35 Function readstringlist

Synopsis: readstringlist(5 args) returns type slist

Read and assign a list variable from a file of separated strings

Example:

```
body common control

{
bundlesequence  => { "example" };
}

#########################################################

bundle agent example

{
vars:

  "mylist" ilist => { readstringlist("/tmp/listofint","#.*","[\n]",10,400) };

reports:

  Yr2008::

    "List entry: $(mylist)";

}
```

ARGUMENTS:

‘`filename`’	The name of a text file containing text to be split up as a list.
‘`comment`’	A regex pattern which is to be ignored in the file
‘`split`’	A regex pattern which is to be used to split up the file into items
‘`maxent`’	The maximum number of list items to read from the file
‘`maxsize`’	The maximum number of bytes to read from the file

Notes:

The following example file would be split into a list of the first ten Green letters.

```
alpha
beta
gamma # This is a comment
delta
epsilon
zeta
eta
theta
iota
kappa
lambda
mu
nu
etc
```

10.36 Function readtcp

Synopsis: readtcp(4 args) returns type string

Connect to tcp port, send string and assign result to variable

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
vars:

  "my80" string => readtcp("www.cfengine.com","80","GET /index.html",400);

reports:

  Yr2008::

    "Server returned: $(my80)";

}
```

‘hostnameip’	The host name or IP address of a tcp socket.
‘port’	The port number to connect to.
‘sendstring’	A string to send to the TCP port to illicit a response
‘maxbytes’	The maximum number of bytes to read in response.

Notes:

If the send string is empty, no data are sent or received from the socket. Then the function only tests whether the TCP port is alive and returns an empty variable.

Note that on some systems the timeout mechanism does not seem to successfully interrupt the waiting system calls so this might hang if you send a query string that is incorrect. This should not happen, but the cause has yet to be diagnosed.

10.37 Function regarray

Synopsis: regarray(2 args) returns type class

True if arg1 matches any item in the associative array with id=arg2

Example:

```
body common control

{
bundlesequence  => { "testbundle"  };
}

###########################################

bundle agent testbundle
{
vars:

  "myarray[0]" string => "bla1";
  "myarray[1]" string => "bla2";
  "myarray[3]" string => "bla";
  "myarray"    string => "345";
  "not"        string => "345";

classes:

  "ok" expression => regarray("myarray","b.*2");

reports:

 ok::

    "Found in list";

 !ok::

    "Not found in list";

}
```

Notes:

Tests whether an associative array contains elements matching a certain regular expression. The result is a class.

ARGUMENTS:

'array_name'	The name of the array, with no '$()' surrounding, etc.
'regex'	A regular expression to match the content.

10.38 Function regcmp

Synopsis: regcmp(2 args) returns type class
True if arg1 is a regular expression matching arg2
Example:

```
bundle agent subtest(user)

{
classes:

  "invalid" not => regcmp("[a-z][a-z][a-z][a-z]","$(user)");

reports:

 !invalid::

  "User name $(user) is valid at 4 letters";

 invalid::

  "User name $(user) is invalid";
}
```

Notes:

Compares a string to a regular expression.
ARGUMENTS:

'regex'	A regular expression to match the test data.
'string'	Test data for the regular expression.

10.39 Function regline

Synopsis: regline(2 args) returns type class
True if arg1 is a regular expression matching a line in file arg2
Example:

```
bundle agent testbundle

{
files:

  "/tmp/testfile" edit_line => test;
}

########################################################

bundle edit_line test
{
classes:

    "ok" expression => regline(".*XYZ.*","$(edit.filename)");

reports:

 ok::

   "File $(edit.filename) has a line with \"XYZ\" in it";

}
```

Notes:

Note that the regular expression must match an entire line of the file in order to give a true result. This function is useful for editfiles applications, where one might want to set a class for detecting the presence of a string which does not exactly match one being inserted. e.g.

```
bundle edit_line upgrade_cfexecd
  {
  classes:

    # Check there is not already a crontab line, not identical to
    # the one proposed below...

    "exec_fix" not => regline(".*cf-execd.*","$(edit.filename)");

  insert_lines:

    exec_fix::

     "0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/cfengine/bin/cf-execd -F";
  reports:

    exec_fix::

     "Added a 5 minute schedule to crontabs";
  }
```

10.40 Function reglist

Synopsis: reglist(2 args) returns type class
True if arg1 matches any item in the list with id=arg2
Example:

```
vars:

 "nameservers" slist => {
                        "128.39.89.10",
                        "128.39.74.16",
                        "192.168.1.103"
                        };
classes:

  "am_name_server" expression => reglist("@(nameservers)","$(sys.ipv4[eth0])");
```

Notes:

Matches a list of test strings to a regular expression.
ARGUMENTS:

‘`list`’	The list of strings.
‘`regex`’	The scalar regular expression string.

10.41 Function returnszero

Synopsis: returnszero(2 args) returns type class
True if named shell command has exit status zero
Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "my_result" expression => returnszero("/usr/local/bin/mycommand","noshell");

reports:

  !my_result::

    "Command failed";

}
```

Notes:

This is the complement of `execresult`, but it returns a class result rather than the output of the command.

10.42 Function rrange

Synopsis: rrange(2 args) returns type rrange [real,real]

Define a range of real numbers for cfengine internal use

Example:

```
?
```

Notes:

This is not yet used.

10.43 Function selectservers

Synopsis: selectservers(6 args) returns type int

Select tcp servers which respond correctly to a query and return their number, set array of names

Example:

```
body common control

{
bundlesequence  => { "test"  };
}

############################################################

bundle agent test

{
vars:

 "hosts" slist => { "slogans.iu.hio.no", "eternity.iu.hio.no", "nexus.iu.hio.no" };

 "up_servers" int =>  selectservers("@(hosts)","80","","","100","alive_servers");

classes:

  "someone_alive" expression => isgreaterthan("$(up_servers)","0");

  "i_am_a_server" expression => regarray("up_servers","$(host)|$(fqhost)");

reports:

  someone_alive::

    "Number of active servers $(up_servers)";
    "First server $(alive_servers[0]) fails over to $(alive_servers[1])";

}
```

Notes:

This function selects all the TCP ports that are active and functioning from an ordered list and builds an array of their names. This allows us to select a current list of failover alternatives that are pretested.

‘`hostlist`’	A list of host names or IP addresses to attempt to connect to.
‘`port`’	The port number for the service.
‘`sendstr`’	An optional string to send to the server to illicit a response.

'regex_on_reply'
: If a string is sent, this regex must match the resulting reply.

'maxbytesread_reply'
: The maximum number of bytes to read as the server's reply.

'array_name'
: The name of the array to build containing the names of hosts that pass the above tests. The array is ordered `array_name[0],..` etc.

10.44 Function splitstring

Synopsis: splitstring(3 args) returns type slist

Convert a string in arg1 into a list of max arg3 strings by splitting on a regular expression in arg2

Example:

```
bundle agent test

{
vars:

  "split1" slist => splitstring("one:two:three",":","10");
  "split2" slist => splitstring("alpha:xyz:beta","xyz","10");

reports:

 linux::

  "Found key $(split1)";
  "Found key $(split2)";

}
```

Returns a list of strings from a string.

ARGUMENTS:

'string'	The string to be split.
'regex'	A regular expression that is to be the delimiter.
'maxent'	The maximum number of list items to be created.

Notes:

If the maximum number is insufficient to accomodate all entries, the final entry will contain the rest of the string.

10.45 Function strcmp

Synopsis: strcmp(2 args) returns type class

True if the two strings match exactly

Example:

```
body common control

{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "same" expression => strcmp("test","test");

reports:

  same::

    "Strings are equal";

 !same::

    "Strings are not equal";
}
```

Notes:

10.46 Function usemodule

Synopsis: usemodule(2 args) returns type class

Execute cfengine module script and set class if successful

Example:

```
body common control
```

```
   {
   any::

      bundlesequence  => {
                         test
                         };
   }

##################################################################

bundle agent test

{
classes:

  # returns $(user)

  "done" expression => usemodule("getusers","");

commands:

  "/bin/echo promiser text" args => "test $(user)";
}
```

Notes:

Modules must reside in '`WORKDIR/modules`' but no longer require a special naming convention.

ARGUMENTS:

'`Module name`'
: The name of the module without its leading path, since it is assuemed to be in the registered modules directory.

'`Argument string`'
: Any command link arguments to pass to the module.

10.47 Function userexists

Synopsis: userexists(1 args) returns type class

True if user name or numerical id exists on this host

Example:

```
body common control
```

```
{
bundlesequence  => { "example" };
}

###########################################################

bundle agent example

{
classes:

  "ok" expression => userexists("root");

reports:

  ok::

    "Root exists";

 !ok::

    "Root does not exist";
}
```

Notes:

Checks whether the user is in the password database for the current host. The argument must be a user name or user id.

11 Special Variables

11.1 Variable context `const`

Cfengine defines a number of variables for embedding unprintable values or values with special meanings in strings.

11.1.1 Variable const.dollar

```
reports:

  some::

   "The value of $(const.dollar)(const.dollar) is $(const.dollar)";

   "But the value of \$(dollar) is \$(dollar)";
```

11.1.2 Variable const.endl

```
reports:

  "A newline with either $(const.endl) or with $(const.n) is ok";
```

11.1.3 Variable const.n

```
reports:

  "A newline with either $(const.n) or with $(const.endl) is ok";
```

11.1.4 Variable const.r

```
reports:
  "A carriage return character is $(const.r)";
```

11.2 Variable context sys

System variables are derived from cfengine's automated discovery of system values. They are provided as variables in order to make automatically adaptive rules for configuration, e.g.

```
files:

 any::

  "$(sys.resolv)"

      create        => "true",
      edit_line     => doresolv("@(this.list1)","@(this.list2)"),
      edit_defaults => reconstruct;
```

The above rule requires no class specification because the variable itself is class-specific.

11.2.1 Variable sys.arch

The variable gives the kernel's short architecture description.

```
# arch = x86_64
```

11.2.2 Variable sys.cdate

The date of the system in canonical form, i.e. in the form of a class.

```
# cdate = Sun_Dec__7_10_39_53_2008_
```

11.2.3 Variable sys.class

This variable contains the name of the hard-class category for this host, i.e. its top level operating system type classification.

```
# class = linux
```

11.2.4 Variable sys.date

The date of the system as a text string.

```
# date = Sun Dec  7 10:39:53 2008
```

11.2.5 Variable sys.domain

The domain name as divined by cfengine. If the DNS is in use, it could be possible to derive the domain name from its DNS regisration, but in general there is no way to discover this value automatically. The `common control` body permits the ultimate specification of this value.

```
# domain = example.org
```

11.2.6 Variable sys.fqhost

The fully qualified name of the host. In order to compute this value properly, the domain name must be defined.

```
# fqhost = host.example.org
```

11.2.7 Variable sys.fstab

The location of the system filesystem (mount) table.

```
# fstab = /etc/fstab
```

11.2.8 Variable sys.host

The name of the current host, according to the kernel. It is undefined whether this is qualified or unqualified with a domain name.

```
# host = myhost
```

11.2.9 Variable sys.interface

The assumed (default) name of the main system interface on this host.

```
# interface = eth0
```

11.2.10 Variable sys.ipv4

All four octets of the IPv4 address of the system interface named as the associative array index, e.g. '$(ipv4_1[$(interface)])'.

```
# The IP on the default interface
#        ipv4 = 192.168.1.101

# The octets of all interfaces as an associative array
# ipv4_1[le0] = 192
# ipv4_2[le0] = 192.168
# ipv4_3[le0] = 192.168.1
#   ipv4[le0] = 192.168.1.101
```

11.2.11 Variable sys.ipv4[wlan0]

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/var_sys_ipv
""
```

11.2.12 Variable sys.ipv4_1[wlan0]

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/var_sys_ipv
""
```

11.2.13 Variable sys.ipv4_2[wlan0]

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/var_sys_ipv
""
```

11.2.14 Variable sys.ipv4_3[wlan0]

```
Fill me in (/home/mark/LapTop/CfengineProjects/CompanyDocuments/trunk/Cfengine3Reference/var_sys_ipv
""
```

11.2.15 Variable sys.long_arch

The long architecture name for this system kernel. This name is sometimes quite unwieldy but can be useful for logging purposes.

```
# long_arch = linux_x86_64_2_6_22_19_0_1_default__1_SMP_2008_10_14_22_17_43__0200
```

11.2.16 Variable sys.maildir

The name of the system email spool directory.

```
# maildir = /var/spool/mail
```

11.2.17 Variable sys.os

The name of the operating system according to the kernel.

```
# os = linux
```

11.2.18 Variable sys.ostype

Another name for the operating system.

```
# ostype = linux_x86_64
```

11.2.19 Variable sys.release

The kernel release of the operating system.

```
# release = 2.6.22.19-0.1-default
```

11.2.20 Variable sys.resolv

The location of the system resolver file.

```
# resolv = /etc/resolv.conf
```

11.2.21 Variable sys.uqhost

The unqualified name of the current host. See also `sys.fqhost`.

```
# uqhost = myhost
```

11.2.22 Variable sys.workdir

The location of the cfengine work directory and cache. For the system privileged user this is normally:

```
# workdir = /var/cfengine
```

For non-privileged users it is in the user's home directory:

```
# workdir = /home/user/.cfagent
```

12 Logs and records

Cfengine writes numerous logs and records to its private workspace, referred to as '`WORKDIR`'. This chapter makes some brief notes about these files. Cfengine approaches monitoring and reporting from the viewpoint of scalability so there is no default centralizatio of reporting information, as this is untenable for more than a few hundred hosts. Instead, in the classic cfengine way, every host is reponsible for its own data. Solutions for centralization and netwide reporting will be given elsewhere.

The filenames referred to in this section are all relative to the cfengine work directory '`WORKDIR`'.

12.1 Embedded Databases

The embedded databases can be viewed and printed using the reporting tool `cf-report`.

'`cf_Audit.db`'
: A compressed database of auditing information. This file grows very large is auditing is switched on. By default, only minor information about cfengine runs are recorded. This file should be archived and deleted regularly to avoid choking the system.

'`cf_LastSeen.db`'
: A database of hosts that last contacted this host, or were contacted by this host which includes the times at which they were last observed.

'`cf_classes.db`'
: A database of classes that have been defined on the current host, including their relative frequences, scaled like a probability.

'`checksum_digests.db`'
: The database of hash values used in cfengine's change management functions.

'`performance.db`'
: A database of last, average and deviation times of jobs recorded by `cf-agent`. Most promises take an immeasurablely short time to check, but longer tasks such as command execution and file copying are measured by default. Other checks can be instrumented by setting a `measurement_class` in the `action` body of a promise.

'`cfengine_lock_db`'
: A database of active and inactive locks and their expiry times. Deleting this database will reset all lock protections in cfengine.

'`state/cf_observations.db`'
: This database contains the current state of the observational history of the host as recorded by `cf-monitord`.

'`state/cf_state.db`'
: A database of persistent classes active on this current host.

12.2 Text logs

'`promise.log`'
: A time-stamped log of the percentage fraction of promises kept after each run.

'`cf3.HOSTNAME.runlog`'
: A time-stamped log of when each lock was released. This shows the last time each individual promise was verified.

'`cfagent.HOSTNAME.log`'
: Although ambiguously named (for historical reasons) this log contains the current list of setuid/setgid programs observed on the system. Cfengine warns about new additions to this list.

'`state/file_hash_event_history`'
: A time-stamped log of which files have experienced content changes since the last observation, as determined by the hashing algorithms in cfengine.

12.3 Reports in outputs

The '`outputs`' directory contains a time-stamped list of outputs generated by `cf-agent`. These are collected by `cf-execd` and are often E-mailed as reports. However, not all hosts have an E-mail capability or are online, so the reports are kept here. Reports are not tidied automatically, so you should delete these files after a time to avoid a build up.

12.4 State information

The cfengine components keep their current process identifier number in 'pid files' in the work directory: e.g.

```
cf-execd.pid
cf-serverd.pid
```

Most other state data refer to the running condition of the host and are generated by `cf-monitord` (`cfenvd` in earlier versions of cfengine).

'`state/env_data`'
: This file contains a list of currently discovered classes and variable values that characterize the anomaly alert environment. They are altered by the monitor daemon.

'`state/all_classes`'
: A list of all the classes that were defined the last time that cfengine was run.

'`state/cf_*`'
: All files that begin with this prefix refer to cached data that were observed by the monitor daemon, and may be used by `cf-agent` in `reports` with `showstate`.

Table of Contents