CFENGINE

Edition 4.5 for version 1.4.9

Mark Burgess
Centre of Science and Technology,
Faculty of Engineering, Oslo College, Norway

This manual corresponds to CFENGINE Edition 4.5 for version 1.4.9 as last updated December 1997.

Foreword

Cfengine is the result of a continuing research project to help solve the problems of system administration in a big network. Cfengine is an expert system combined with a declarative language and a workhorse-robot.

Many people have contributed their experiences and wisdom to the cfengine project. I apologise for not being able to mention everyone here. Morten Hanshaugen and Hans Petter Holen made it possible to test cfengine on a variety of systems at the university of Oslo. I am grateful to Knut Borge for his experience and suggestions on many occaision. Ola Borrebaek and Richard Stallman have made key observations which have influenced the development of cfengine in important ways. Audun Tornquist did some initial work on the 'copy' feature and donated the backup help-script to the distribution. Gord Matzigkeit contributed an early autoconf setup. Andrew Ford contributed the self-documentation perl script. Ricky Ralston (Hewlett Packard) provided invaluable information on HPUX-10 and discovered a number of bugs and inaccuracies in the source code: our collaboration on making 1.3.0 the definitive system administration tool (before 1.4.0!) has been invaluable. David Masterson continues to provide me with the results of detailed tests and new auto configuration improvements. Brian White maintains the Debian linux package and has been helpful with bug reports. Rolf Ebert contributed the emacs cfengine mode file. Max Okumoto ran the source code through 'insight' and found lots of accidentals. Sergio Tessaris has done lots of testing and bug tracking. I am grateful to Ann-Mari Torvatn and Len Tower for reading through and helping to improve the documentation. Finally, Demosthenes Skipitaris and I added the new adaptive locks to cfengine 1.4.0.

For up to the minute information on cfengine, workshops, conferences and all that jazz see the web page:

```
http://www.iu.hioslo.no/cfengine
```

Bug reports and queries by mail to

```
bug-cfengine@prep.ai.mit.edu
```

(why not bug Jack Barron instead?) Two newsgroups are also available now for discussions and bug reports. The newsgroup gnu.cfengine.bug (with corresponding mailing list bug-cfengine@prep.ai.mit.edu) is for bug reports, and the group gnu.cfengine.help (with mailing list help-cfengine@prep.ai.mit.edu) is for general requests for help and user discussion.

Mark Burgess Oslo, 1997

STATE OF MIND (FOR THE CRITICS)

Kirk: I'm curious, Doctor, why is it called the M5?

Daystrom: Well you see, M1 to M4 were not entirely successful. This one is. M5 is ready to take control of your ship.

Kirk: Total control?

Daystrom: That is what it is designed for.

Kirk: There are some things that Men have to do to remain Men, your computer takes that away.

Daystrom: The computer can do your job ... One machine can do all those things that Men do now. Men can go on to do greater things...

–*ST: The ultimate computer*

Scientist: Mad, mad? Of course I'm mad! But I have tenure!

–*Cartoon from OMNI magazine, early 1980's*

PART I

Tutorial section

1 Overview

In this manual the word "host" is used to refer to a single computer system – i.e. a single machine which has a name termed its "hostname".

1.1 What is cfengine and who can use it?

Cfengine is a tool for setting up and maintaining BSD and System-5-like operating system optionally attached to a TCP/IP network. You can think of cfengine as a very high level language—much higher level than Perl or shell: a single statement can result in many hundreds of operations being performed on multiple hosts. Cfengine is good at performing a lot of common system administration tasks, and allows you to build on its strengths with your own scripts. You can also use it as a netwide front-end for `cron`. Once you have set up cfengine, you'll be free to use your time being like a human being, instead of playing R2-D2 with the system.

The main purpose of cfengine is to allow you to create a single, central system configuration which will define how every host on your network should be configured in an intuitive way. An interpreter runs on every host on your network and parses the master file (or file-set); the configuration of each host is checked against this file and then, if you request it, any deviations from the defined configuration are fixed automatically. You do not have to mention every host specifically by name in order to configure them : instead you can refer to the properties which distinguish hosts from one another. Cfengine uses a flexible system of "classes" which helps you to single out a specific group of hosts with a single statement.

Originally cfengine was conceived of as a tool only for the superuser, but during the course of its development it has become clear that it can also be used as a scripting language by ordinary users. It is a handy tool for tidying your old junk files and for making 'watchdog' scripts to manage the access rights and permissions on your files when collaborating with other users. As a bonus it contains a text editing language which can be used to perform controlled edits of line-based text files.

Cfengine grew out of the need to control the accumulation of complex shell scripts used in the automation of key system maintenance at Oslo. There were very many scripts, written in shell and in perl, performing tasks such as file tidying, find-database updates, process checking and several other tasks. In a heterogeneous environment, shell-scripts work very poorly: shell commands have differing syntax across different operating systems, the locations and names of key files differ. In fact, the non-uniformity of unix was a major headache. Scripts were filled with tests to determine what kind of operating system they were being run on, to the point where they became so complicated an unreadable that no-one was quite sure what they did anymore. Other scripts were placed only on the systems where they were relevant, out of sight and out of mind. It quickly became clear that our dream solution would be to replace this proliferation of scripts by a single file containing everything to be checked on every host on the network. By defining a new language, this file could hide all of the tests by using classes (a generalized 'switch/case' syntax) to label operations and improve the readability greatly. The gradual refinement of this idea resulted in the present day cfengine.

The remainder of this manual assumes that you know a little about BSD/System-5 systems and have everyday experience in using either the C-shell or the Bourne shell, or their derivatives. If you are experienced in system administration, you might like to skip the earlier chapters and turn straight to the example See ⟨undefined⟩ [Example configuration file], page ⟨undefined⟩. This is the probably quickest way to learn cfengine for the initiated. If you are not so familiar with system administration and would like a more gentle introduction, then we begin here...

1.2 Site configuration

To the system administrator of a small network, with just a few workstations or perhaps even a single mainframe system, it might seem superfluous to create a big fuss about the administration of the system. After all, it's easy to 'fix' things manually should any problems arise, making a link here, writing a script there and so on — and its probably not even worth writing down what you did because you know that it will always be easy to fix next time around too... But networks have a tendency to expand and—before you know it—you have five different types of operating system and each type of system has to be configured in a special way, you have to make patches to each system and you can't remember whether you fixed that host on the other side of the building... Also, you discover fairly quickly that what you thought of as BSD or System 5 is not as standard as you thought and that none of your simple scripts that worked on one system work on the others without a considerable amount of hacking and testing. You try writing a script to help you automate the task, but end up with an enormous number of '`if..then..else..`' tests which make it hard to see what is really going on.

To manage a network with many different flavours of operating system, in a systematic way, what is needed is a more disciplined way of making changes which is robust against re-installation. After all, it would be tragic to spend many hours setting up a system by hand only to lose everything in an unfortunate disk-crash a week or even a year later when you have forgotten what you had to do. Upgrades of the operating system software might delete your carefully worked out configuration. What is needed is a separate record of all of the patches required on all of the systems on the network; a record which can be compared to the state of each host at any time and which a suitable engine can use to fix any deviations from that reference standard.

The idea behind cfengine is to focus upon a few key areas of basic system administration and provide a language in which the transparency of a configuration program is optimal. It eliminates the need for lots of tests by allowing you to organize your network according to "classes". From a single configuration file (or set of files) you can specify how your network should be configured — and cfengine will then parse your file and carry out the instructions, warning or fixing errors as it goes.

1.3 Key Concepts

Some of the important issues in system administration which cfengine can help with.

1.3.1 Control files

One of the endearing characteristics of BSD and system 5 systems is that they are configured through human-readable text files. To add a new user to the system you edit '`/etc/passwd`', to add a new disk you must edit '`/etc/fstab`' etc. Many applications are also configured with the help of text files. When installing a new system for the first time, or when changing updating the setup of an old system you are faced with having to edit lots of files. In some cases you will have to add precisely the same line to the same file on every system in your network as a change is made, so it is handy to have a way of automating this procedure so that you don't have to load every file into an editor by hand and make the changes yourself. This is one of the tasks which cfengine will automate for you.

1.3.2 Network interface

Each host which you connect to an ethernet-based network running TCP/IP protocols must have a so-called 'net interface'. This network interface must be configured before it will work. Normally one does this with the help of the `ifconfig` command. This can also be checked and configured automatically by cfengine.

Network configuration involves telling the interface hardware what the internet (IP) address of your system is, so that it knows which incoming 'packets' of data to pay attention to. It involves telling the interface how to interpret the addresses it receives by setting the 'netmask' for your network (see below). Finally you must tell it which dummy address is to be used for messages which are broadcast to all hosts on your network simultaneously See ⟨undefined⟩ [netmask], page ⟨undefined⟩.

1.3.3 Network File System (NFS) or distribution?

Probably the first thing you are interested in doing with a network (after you've had your fill of the world wide web) is to make your files available to some or all hosts on the network, no matter where in your corporate empire (or university dungeon) you might be sitting. In other words, if you have a disk which is physically connected to host A, you would like to make the contents of that disk available to hosts B, C, D... etc. NFS (the network filesystem) does this for you. The process works by 'filesystems'.

A filesystem is one partition of a disk drive – or one unit of disk space which can be accessed by a single 'logical device' '`/dev/something`'. To make a filesystem available to other hosts you have to do three things.

- On the host the disk is physically connected to you must *export* the filesystem by adding something to the file '`/etc/exports`'. This tells NFS who is allowed to access the disk and who isn't.
- On the host which is to access the filesystem you must create a mount point. This is a name in the directory tree at which you want to add the files to your local filesystem.
- On the host which is to access the files you must mount the filesystem onto the mount point. The mount operation is the jargon for telling the system to access the device on which the data are stored. Mounting is analogous to opening a file: files are opened, filesystems are mounted.

Only after all three of these have been done will a filesystem become available across the network. Cfengine will help you with the last two in a very transparent way. You could also use the text-editing facility in cfengine to edit the exports file, but there are other ways update the exports file using *netgroups* which we shall not go into here. If you are in doubt, look up the manual page on exports.

Some sites prefer to minimize the use of NFS filesystems, to avoid one machine being dependent on another. They prefer to make a local copy of the files on a remote machine instead. Traditionally programs like `rdist` have been used for this purpose. You may also use cfengine to copy files in this way, See ⟨undefined⟩ [Emulating rdist], page ⟨undefined⟩.

1.3.4 Name servers (DNS)

There are two ways to specify addresses on the internet (called IP addresses). One is to use the textual address like '`ftp.uu.net`' and the other is to use the numerical form '`192.48.96.9`'. Alas, there is no one-to-one correspondence between the numerical addresses and the textual ones, thus a service is required to map one to the other.

The service is performed by one or more special hosts on the network called *nameservers*. Each host must know how to contact a nameserver or it will probably hang the first time you give it an IP address. You tell it how to contact a nameserver by editing the text-file '`/etc/resolv.conf`'. This file must contain the domain name for your domain and a list of possible nameservers which can be contacted, in order of priority. Because this is a special file which every host must have, you don't have to use the editing facilities in cfengine explicitly. You can just define the nameservers for each host in the cfengine file and cfengine will do the editing automatically. If you want to change the priority of nameservers later, or even change the list then a simple change of one or two lines in the configuration file will enable you to reconfigure every host on your network automatically without having to do any editing yourself!

1.3.5 Monitoring important files

Security is an important issue on any system. In the busy life of a system administrator it is not always easy to remember to set the correct access rights on every file and this can result in either a security breach or problems in accessing files.

A common scenario is that you, as administrator, fetch a new package using ftp, compile it and install it without thinking too carefully. Since the owner and permissions of the files in an ftp archive remains those of the program author, it often happens that the software is left lying around with the owner and permissions as set by the author of the program rather than any user-name on *your* system. The user-id of the author might be anybody on your system — or perhaps nobody at all! The files should clearly be owned by root and made readable and unwritable to normal users.

Simple accidents and careless actions under stress could result in, say, the password file being writable to ordinary users. If this were the case, the security of the entire system would be compromised. Cfengine therefore allows you to monitor the permissions, ownership and general existence of files and directories and, if you wish, correct them or warn about them automatically.

1.3.6 Making links

One of the difficulties with having so many different variations on the theme of BSD and system 5 based operating systems is that similar files are not always where you expect to find them. They have different names or lie in different directories. The usual solution to the problem is to make an alias for these files, or a pointer from one filename to another. The name for such an alias is a *symbolic link*.

It is often very convenient to make symbolic links. For example, you might want the sendmail configuration file '`/etc/sendmail.cf`' to be a link to a global configuration file, say,

'`/usr/local/mail/etc/sendmail.cf`'

on every single host on your network so that there is only one file to edit. If you had to make all of these links yourself, it would take a lifetime. Cfengine will make such a link automatically and check it each time time is run. You can also ask it to tidy up old links which have been left around and no longer point to existing files. If you reinstall your operating system later it doesn't matter because all your links are defined in your cfengine configuration file, recorded for all time. Cfengine won't forget it, and you won't forget it because the setup is defined in one central place.

Cfengine will also allow you to make hard links to regular files, but not other kinds of file. A hard link to a symbolic link, is the same as a hard link to the file the symbolic link points to.

1.4 Functionality

The notes above give you a rough idea of what cfengine can be used for. Here is a summary of cfengine's capabilities.

- Check and configure the network interface.
- Edit textfiles for the system and for all users.
- Make and maintain symbolic links, including multiple links from a single command.
- Check and set the permissions and ownership of files.
- Tidy (delete) junk files which clutter the system.
- Systematic, automated mounting of NFS filesystems.
- Checking for the presence of important files and filesystems.
- Controlled execution of user scripts and shell commands.
- Cfengine follows a class-based decision structure.
- Process management.

How do you run cfengine? You can run it as a cron job, or you can run it manually. You may run cfengine scripts/programs as often as you like. Each time you run a script, the engine determines whether anything needs to be done — if nothing needs to be done, nothing is done! If you use it to monitor and configure your entire network from a central file-base, then the natural thing is to run cfengine daily with the help of `cron`. See ⟨undefined⟩ [cfwrap], page ⟨undefined⟩, but also, See ⟨undefined⟩ [cfrun], page ⟨undefined⟩.

2 Getting started

2.1 What you must have in a cfengine program

A cfengine configuration file for a large network can become long and complex so, before we get down to details, let's try to strip away the complexity and look only to the essentials.

Each cfengine program or configuration file is a list of declarations of items to be checked and perhaps fixed. You begin by creating a file called '`cfengine.conf`'. The simplest meaningful file you can create is something like this:

```
# Comment...

control:

  actionsequence = ( links )

links:

  /bin -> /usr/bin
```

The example above checks and makes (if necessary) a link from '`/bin`' to '`/usr/bin`'. Let's examine this example more closely. In a cfengine program:

- Use of space is unrestricted. You can start new lines wherever you like. You should generally have a space before and after parentheses to avoid confusing the parser.
- A comment is some text which is ignored by cfengine. The '`#`' symbol designates a comment and means: ignore the remaining text on this line. A comment symbol must have a space in front of it, or start a new line so that cfengine knows you don't mean the symbol as part of another word.
- Words which end in a single colon define *sections* in a program. Under a given section you group together all declarations of a given type. Section names must all be taken from a list defined by the language. You cannot define your own sections.
- Words which end in two colons are so-called *class* names. They are used for making decisions in cfengine.
- Statements which are of the form *name*`=(` *list* `)` are used to assign the value on the right hand side to the name on the left hand side of the equals sign.

In simple example above has three of the four types of object described above. The `control:` section of any program tells cfengine how to behave. In this example it adds the action *links* to the actionsequence. For *links* you could replace some other action. The essential point is that, if you don't have an action sequence, your cfengine program will do absolutely nothing! The action sequence is a list which tells cfengine what do to and in which order.

The `links:` section of the file tells cfengine that what follows is a number of links to be made. If you write this part of the file, but forget to add links to the actionsequence, then nothing will be done! You can add any number of links in this part of the file and they will all be dealt with in order when—and only when—you write *links* in the action sequence.

To summarize, you *must* have:

- Some declarations which specify things to be done.
- An action sequence which tells cfengine which sections to process, how many times and in which order they should be processed.

Now let's think a bit about how useful this short example program is. On a SunOS system, where the directory '`/bin`' is in fact supposed to be a link, such a check could be useful, but on some other system where '`/bin`' is a not a link but a separate directory, this would result in an error message from cfengine, telling you that '`/bin`' exists and is not a link. The lesson is that, if we want to use cfengine to make *one single* program which can be run on any host of any type, then we need some way of restricting the above link so that it only gets checked on SunOS systems. We can write the following:

```
# Comment...

control:

  actionsequence = ( links  )

links:

  sun4::

       /bin -> /usr/bin
       # other links

   osf::

       # other links
```

The names which have double colons after them are called *classes* and they are used to restrict a particular action so that it only gets performed if the host running the program is a member of that class. If you are familiar with C++, this syntax should make you think of classes definitions in C++. Classes works like this: the names above `sun4`, `sun3`, `osf` etc. are all internally defined by cfengine. If a host running, say, the OSF operating system executes the file it automatically becomes a member of the class `osf`. Since it cannot be a member more than one of the above, this distinguishes between different types of operating system and creates a hidden `if`..`then`...`else` test.

This is the way in which cfengine makes decisions. The key idea is that actions are only carried out if they are in the same class as the host running the program. Classes are dealt with in detail in the next chapter.

Now let's see how to add another kind of action to the action sequence.

```
# Comment...

control:

  actionsequence = ( tidy links )
```

```
links:

  /bin -> /usr/bin

tidy:

   /tmp  pattern=* age=7 recurse=inf
```

We have now added a new kind of declaration called `tidy:` which deletes files. In the example above, we are looking at files in the directory '`/tmp`' which match the pattern '`*`' and have not been accessed for more than seven days. The search for these files descends recursively down any number of subdirectories.

To make any of this happen we must add the word *tidy* to the action sequence. If we don't, the declaration will be ignored. Notice also that, regardless of the fact that `links:` comes before `tidy:`, the order in the action sequence tells us that all `tidy` actions will be performed before `links:`.

The above structure can be repeated to build up a configuration file or script.

2.2 Program structure

To summarize the previous section, here is a sketch of a typical cfengine configuration program showing a sensible structure. The various sections are listed in a sensible order which you would probably use in the action sequence.

An individual section-declaration in the program looks something like this:

action-type:

   class1::

      list of things to do...

   class2::

      list of things to do...

`action-type` is one of the following reserved words:

```
groups, control, homeservers, binservers, mailserver, mountables,
import, broadcast, resolve, defaultroute, directories, miscmounts,
files, ignore, tidy, required, links, disable, shellcommands,
editfiles, processes
```

The order in which declarations occur is not important to cfengine from a syntactical point of view, but some of the above actions define information which you will want to refer to later. All variables, classes, groups etc. must be defined before they are used. That means that it is smart to follow the order above for the sections in the first line of the above list.

The order in which items are declared is not to be confused with the order in which they are executed. This is determined by the `actionsequence`, See ⟨undefined⟩ [control], page ⟨undefined⟩. Probably you will want to coordinate the two so that they match as far as possible.

For completeness, here is a complete summary of the structure of a very general cfengine configuration program. The format is free and use of space is unrestricted, though it is always a good idea to put a space in front before and after parentheses when defining variables.

```
######################################################################
#
# Example of structure
#
######################################################################

groups:

   group1 = ( host host ...  )
   group2 = ( host host ...  )
   ...

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

control:

   class::

   site      =  ( mysite )
   domain    =  ( mydomain )
   ...

    actionsequence =
      (
      action name
      ....
      )

   mountpattern = ( mountpoint )
   homepattern = ( wildcards matching home directories )

   addclasses = ( foo bar )

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

homeservers:

   class::
           home servers

binservers:
```

```
    class::
                binary servers

mailserver:

    class::
                mail server

mountables:

    class::

                list of resources

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

import:

    class::     include file

    class::     include file

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

broadcast:

  class::  ones   # or zeros / zeroes

defaultroute:

    class::  my-gw

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

resolve:

    any::

         list of nameservers

    ...
```

2.3 Optional features in cfengine

Cfengine doesn't do anything unless you ask it to. When you run a cfengine program it generates no output unless it finds something it believes to be wrong. It does not carry out any actions unless they are declared in the action sequence. In fact it's just like one of those people you try to avoid at the office because they only complain about what's wrong and never ever say anything positive. But all this can change.

If you like, you can make cfengine positively chatty. Cfengine can be run with a number of command line options See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩. If you run the program with the '-v' or '--verbose' options, it will supply you cheerily with a resume of what it is doing. Certain warning messages also get printed in verbose mode.

You can ask cfengine to check lots of things – the timezone for instance, or the domain name. In order for it to check these things, it needs some information from you. All of the switches and options which change the way in which cfengine behaves get specified either on the command line or in the `control:` section of the control file. Some special control variables are used for this purpose. Here is a short example:

```
control:

  domain   = ( mydomain.no )
  netmask  = ( 255.255.255.0 )
  timezone = ( MET )

  mountpattern = ( /mydomain/mountpoint )

  actionsequence =
     (
     checktimezone     # check time zone
     netconfig         # includes check netmask
     resolve           # includes domain
     mountinfo         # look for mounted disks under mountpattern
     )
```

To get verbose output you must run cfengine with the appropriate command line option '--verbose' or '-v'.

Notice that setting values has a special kind of syntax: a variable name, an equals sign and a value in parentheses. This tells you that the quantity of the left hand side assumes the value on the right hand side. There are lots of questions you might ask at this point. The answers to these will be covered as we go along and in the next chapter.

Before leaving this brief advertisement for control parameters, it is worth noting the definition of `mountpattern` above. This declares a directory in which cfengine expects to find mounted disks. It will be explained in detail later, for now notice that this definition looks rather stupid and inflexible. It would be much better if we could use some kind of variables to define where to look for mounted filesystems. And of course you can...

Having briefly scraped the surface of what cfengine can do, turn to the example See ⟨undefined⟩ [Example configuration file], page ⟨undefined⟩ and take a look at what a complete program can look like. If you understand it, you might like to skip through the rest of the manual until you find what you are looking for. If it looks mysterious, then the next chapter should answer some questions in more depth.

2.4 Invoking cfengine

Cfengine may be invoked in a number of ways. Here are some examples:

```
host% cfengine

host% cfengine --file myfile

host% cfengine -f myfile -v -n

host% cfengine --help
```

The first of these (the default command, with no arguments) causes cfengine to look for a file called '`cfengine.conf`' in the current directory and execute it silently. The second command reads the file '`myfile`' and works silently. The third works in verbose mode and the `-n` option means that no actions should actually be carried out, only warnings should be printed. The final example causes cfengine to print out a list of its command line options.

The complete list of options is listed in the summary at the beginning of this manual, or you can see it by giving the `-h` option. See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩

In addition to running cfengine with a filename, you can also treat cfengine files as scripts by starting your cfengine program with the standard shell line:

```
#!/local/gnu/bin/cfengine -f
#
# My config script
#
```

Here we assume that you have installed cfengine under the directory '`/local/gnu/bin`'. By adding a header like this to the first line of your program and making the file executable with the `chmod` shell command, you can execute the program just by typing its name—i.e. without mentioning cfengine explicitly at all.

As a novice to cfengine, it is advisable to check all programs with the `-n` option before trusting them to your system, at least until you are familiar with the behaviour of cfengine. This 'safe' option allows you to see what cfengine wants to do, without actually committing yourself to doing it.

2.5 CFINPUTS environment variable

Whenever cfengine looks for a file it asks a question: is the filename an absolute name (that is a name which begins from '`/`' like `/usr/file`), is it a file in the directory in which you invoke cfengine or is it a file which should be searched for in a special place?

If you use an absolute filename either on the command line using `-f` or in the `import` section of your program (a name which begins with a slash '/'), then cfengine trusts the name of the file you have given and treats it literally. If you specify the name of the file as simple '`.`' then cfengine reads its input from the standard input.

If you run cfengine without arguments (so that the default filename is '`cfengine.conf`') or you specify a file without a leading slash in the `import` section, then the value of the environment variable `CFINPUTS` is prepended to the start of the file name. This allows you to keep your configuration in a standard place, pointed to by `CFINPUTS`. For example:

```
host# setenv CFINPUTS /usr/local/gnu/lib/cfengine/inputs

host# cfengine -f myfile
```

In this example, cfengine tries to open

'`/usr/local/gnu/lib/cfengine/inputs/myfile`'.

2.6 What to aim for

If you are a beginner to cfengine, you might not be certain exactly how you want to use it. Here are some hints from Dr. Daystrom about how to get things working quickly.

- Run cfengine from cron every hour on all your systems. Be sure to label long tasks, or tasks which do not need to be performed often by a *time class* which prevents it from being executed all the time, See ⟨undefined⟩ [Using cfengine as a front-end for cron], page ⟨undefined⟩.

  Running cfengine from cron means that it will be run in parallel on your systems. Cfengine on one host does not have to wait for cfengine on another host to complete.

- Set up `cfd` on all your systems so that cfengine can be executed remotely, so that you can immediately "push" changes to all your hosts with `cfrun`. Think carefully about whom you wish to give permission to run cfengine from the net, See ⟨undefined⟩ [Configuring cfd], page ⟨undefined⟩. Set up you '`cfd.conf`' file accordingly. You can also use this daemon to grant access rights for remote file copying.

  Cfrun polls all your hosts serially and gives you a concatenated indexed list of problems on all hosts. The disadvantage with cfrun is that each host has to wait its turn.

- Don't forget to add `cfd` to the system startup scripts, or to '`inittab`' so that it starts when you boot your system.
- Add *all* your hosts to the '`cfrun.hosts`' file. It does not matter that some may be master servers and others clients. The locking mechanisms will protect you from silliness, See ⟨undefined⟩ [Deadlocks and runaway loops], page ⟨undefined⟩. Cfengine will work it out. Cfrun allows you to remotely execute cfengine on groups of hosts which satisfy a list of cfengine classes.

When you have set up these components, you can sit back and edit the configuration files and watch things being done.

3 More advanced concepts

3.1 Classes

The idea of classes is central to the operation of cfengine. Saying that cfengine is 'class orientated' means that it doesn't make decisions using `if`...`then`...`else` constructions the way other languages do, but only carries out an action if the host running the program is in the same class as the action itself. To understand what this means, imagine sorting through a list of all the hosts at your site. Imagine also that you are looking for the *class* of hosts which belong to the computing department, which run GNU/Linux operating system and which have yellow spots! To figure out whether a particular host satisfies all of these criteria you first delete all of the hosts which are not GNU/Linux, then you delete all of the remaining ones which don't belong to the computing department, then you delete all the remaining ones which don't have yellow spots. If you are on the remaining list, then you are in the class of all computer-science-Linux-yellow-spotted hosts and you can carry out the action.

Cfengine works in this way, narrowing things down by asking if a host is in several classes at the same time. Although some information (like the kind of operating system you are running) can be obtained directly, clearly, to make this work we need to have lists of which hosts belong to the computer department and which ones have yellow spots.

So how does this work in a cfengine program? A program or configuration script consists of a set of declarations for what we refer to as *actions* which are to be carried out only for certain classes of host. Any host can execute a particular program, but only certain action are extracted — namely those which refer to that particular host. This happens automatically because cfengine builds up a list of the classes to which it belongs as it goes along, so it avoids having to make many decisions over and over again.

By defining classes which classify the hosts on your network in some easy to understand way, you can make a single action apply to many hosts in one go – i.e. just the hosts you need. You can make generic rules for specific type of operating system, you can group together clusters of workstations according to who will be using them and you can paint yellow spots on them – what ever works for you.

A *cfengine action* looks like this:

action-type:

    compound-class::

        declaration

A single class can be one of several things:

- The name of an operating system architecture e.g. `ultrix`, `sun4` etc. This is referred to henceforth as a *hard class*.
- The (unqualified) name of a particular host. If your system returns a fully qualified domain name for your host, cfengine truncates it so as to unqualify the name.
- 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 ... Day31).
- A month (in the form January, February, ... December).
- A year (in the form Yr1997, Yr2001).
- An arbitrary user-defined string. See ⟨undefined⟩ [addclasses], page ⟨undefined⟩.

A compound class is a sequence of simple classes connected by dots or 'pipe' symbols (vertical bars). For example:

```
myclass.sun4.Monday::

sun4|ultrix|osf::
```

A compound class evaluates to 'true' if all of the individual classes are separately true, thus in the above example the actions which follow `compound_class::` are only carried out if the host concerned is in `myclass`, is of type `sun4` and the day is Monday! In the second example, the host parsing the file must be either of type `sun4` *or* `ultrix` *or* `osf`. In other words, compound classes support two operators: AND and OR, written '`.`' and '`|`' respectively. Cfengine doesn't care how many of these operators you use (since it skips over blank class names), so you could write either

```
solaris|irix::
```

or

```
solaris||irix::
```

depending on your taste. On the other hand, the order in which cfengine evaluates AND and OR operations *does* matter, and the rule is that AND takes priority over OR, so that '`.`' binds classes together tightly and all AND operations are evaluated before ORing the final results together. This is the usual behaviour in programming languages. You can use round parentheses in cfengine classes to override these preferences.

Cfengine allows you to define switch on and off dummy classes so that you can use them to select certain subsets of action. In particular, note that by defining your own classes, using them to make compound rules of this type, and then switching them on and off, you can also switch on and off the corresponding actions in a controlled way. The command line options `-D` and `-N` can be used for this purpose. See also ⟨undefined⟩ [addclasses], page ⟨undefined⟩.

A logical NOT operator has been added to allow you to exclude certain specific hosts in a more flexible way. The logical NOT operator is (as in C and C++) '`!`'. For instance, the following example would allow all hosts except for `myhost`:

```
action:

 !myhost::
```

command

and similarly, so allow all hosts in a user-defined group `mygroup`, *except* for `specialhost`, you would write

```
action:

 mygroup.!myhost::

        command
```

which reads 'mygroup AND NOT myhost'. The NOT operator can also be combined with OR. For instance

```
class1|!class2
```

would select hosts which were either in class 1, or those which were not in class 2.

Finally, there is a number of reserved classes. The following are hard classes for various operating system architectures. They do not need to be defined because each host knows what operating system it is running. Thus the appropriate one of these will always be defined on each host. Similarly the day of the week is clearly not open to definition, unless you are running cfengine from outer space. The reserved classes are:

```
ultrix, sun4, sun3, hpux, hpux10, aix, solaris, osf, irix4, irix, irix64
        freebsd, netbsd, bsd4_3, newsos, solarisx86, aos,
            nextstep, bsdos, linux, debian, cray
```

If these classes are not sufficient to distinguish the hosts on your network, cfengine provides more specific classes which contain the name and release of the operating system. To find out what these look like for your systems you can run cfengine in 'parse-only-verbose' mode:

```
cfengine -p -v
```

and these will be displayed. For example, solaris 2.4 systems generate the additional classes `sunos_5_4` and `sunos_sun4m`, `sunos_sun4m_5_4`.

Cfengine uses both the unqualified and fully host names as classes. Some sites and operating systems use fully qualified names for their hosts. i.e. `uname -n` returns to full domain qualified hostname. This spoils the class matching algorithms for cfengine, so cfengine automatically truncates names which contain a dot '.' at the first '.' it encounters. If your hostnames contain dots (which do not refer to a domain name, then cfengine will be confused. The moral is: don't have dots in your host names! *NOTE: in order to ensure that the fully qualified name of the host becomes a class you must define the domain variable.* The dots in this string will be replaced by underscores.

In summary, the operator ordering in cfengine classes is as follows:

'`()`' Parentheses override everything.

'`!`' The NOT operator binds tightest.

'`.`' The AND operator binds more tightly than OR.

'`|`' OR is the weakest operator.

3.2 Variable substitution

When you are building up a configuration file it is very useful to be able to use variables. If you can define your configuration in terms of some key variables, it can be changed more easily later, it is more transparent to the reader of the program and you can also choose to define the variables differently on different types of system. Another way of saying this is that cfengine variables also belong to classes. Cfengine makes use of variables in three ways.

- Environment variables from the shell
- Special variables used in cfengine features
- General macro-string substitution.

Environment variables are fetched directly from the shell on whatever system is running the program. An example of a special variable is the `domain` variable from the previous section. Straightforward macro substitution allows you to define a symbol name to be replaced by an arbitrary text string. All these definitions (apart from shell environment variables, of course) are made in the control part of the cfengine program:

```
control:

  myvar = ( /usr/local/mydir/lib/very/long/path )   # define macro

...

links:

  $(myvar) -> /another/directory
```

Here we define a macro called `myvar`, which is later used to define the creation of a link. As promised we can also define class-dependent variables:

```
control:

  sun4:: myvar = ( sun )
  hpux:: myvar = ( HP )
```

Cfengine gives you access to the shell environment variables and allows you to define variables of your own. It also keeps a few special variables which affect the way in which cfengine works. When cfengine expands a variable it looks first at the name in its list of special variables, then in the list of user-defined macros and finally in the shell environment for a match. If none of these are found it expands to the empty string.

Variables are referred to in either of two different ways, depending on your taste. You can use the forms `$(variable)` or `${variable}`. The variable in braces or parentheses can be the name of any user defined macro, environment variable or one of the following special internal variables.

`AllClasses`
: A long string in the form '`CFALLCLASSES=class1:class2...`'. This variable is a summary of all the defined classes at any given time. It is always kept up to date so that scripts can make use of cfengine's class data.

`arch` The current detailed architecture string—an amalgamation of the information from *uname*. Non-definable.

`binserver`
The default server for binary data. See ⟨undefined⟩ [NFS resources], page ⟨undefined⟩. Non definable.

`class` The currently defined system hard-class (e.g. `sun4`, `hpux`). Non-definable.

`domain` The currently defined domain.

`faculty` The faculty or site as defined in control (see site).

`fqhost` The fully qualified (DNS/BIND) hostname of the system, which includes the domain name as well.

`host` The hostname of the machine running the program.

`MaxCfengines`
The maximum number of cfengines which should be allowed to co-exist concurrently on the system. This can prevent excessive load due to unintentional spamming in situations where several cfengines are started independently. The default value is unlimited.

`OutputPrefix`
This quoted string can be used to change the default 'cfengine:' prefix on output lines to something else. You might wish to shorten the string, or have a different prefix for different hosts. The value in this variable is appended with the name of the host. The default is equivalent to,

```
OutputPrefix = ( "cfengine:$(host):")
```

`RepChar` The character value of the string used by the file repository in constructing unique filenames from path names. This is the character which replaces '`/`' See ⟨undefined⟩ [repchar], page ⟨undefined⟩.

`site` This variable is identical to `$(faculty)` and may be used interchangeably.

`split` The character on which list variables are split See ⟨undefined⟩ [split], page ⟨undefined⟩.

`sysadm` The name or mail address of the system administrator.

`timezone` The current timezone as defined in `control`.

`UnderscoreClasses`
If this is set to 'on' cfengine uses hard-classes which begin with an underscore, so as to avoid name collisions. See also See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩.

These variables are kept special because they play a special role in setting up a system configuration. See ⟨undefined⟩ [Global configurations], page ⟨undefined⟩. You are encouraged to use them to define fully generalized rules in your programs. Variables can be used to advantage in defining filenames, directory names and in passing arguments to shell commands. The judicious use of variables can reduce many definitions to a single one if you plan carefully.

NOTE: the above control variables are not case sensitive, unlike user macros, so you should not define your own macros with these names.

The following variables are also reserved and may be used to produce troublesome special characters in strings.

`cr` Expands to the carriage-return character.

`dblquote` Expands to a double quote "

`dollar` Expands to '`$`'.

`lf` Expands to a line-feed character (unix end of line).

`n` Expands to a newline character.

`quote` Expands to a single quote `'`.

`spc` Expands simply to a single space. This can be used to place spaces in filenames etc.

`tab` Expands to a single tab character.

You can use variables in the following places:

- In any directory name. The `$(binserver)` variable is not always appropriate in this context. For instance

```
links:

  osf::
      /$(site)/${host}/directory -> somefile
```

- In any quoted string. See ⟨undefined⟩ [shellcommands], page ⟨undefined⟩.

```
shellcommands:

  any::

   "/bin/echo $(timezone) | /bin/mail $(sysadm)"
   '/bin/echo "double quotes!"'
```

The latter possibility enables cfengine's variables to be passed on to user-defined scripts.

- To define the values of options passed to various actions, in the form *option*`=$(variable)`.

Variables can be defined differently under different classes by preceding the definition with a class name. For example:

```
control:

   sun4::  my_macro = ( User_string_1 )
   irix::  my_macro = ( User_string_2 )
```

Here the value assigned to `$(my_macro)` depends on which of the classes evaluates to true. This feature can be used to good effect to define the mail address of a suitable system administrator for different groups of host.

```
control:

 physics::   sysadm = ( mark,fred )
 chemistry:: sysadm = ( localsys@domain )
```

Note, incidentally, that the '`-a`' option can be used to print out the mail address of the system administrator for any wrapper scripts.

3.3 Defining classes and making exceptions

Because cfengine works at a very high level, doing very many things for very few lines of code it might seem that some flexibility is lost. When we restrict certain actions to special classes it is occasionally useful to be able to switch off classes temporarily so as to cancel the special actions.

You can define classes of your own which can be switched on and off, either on the command line or from the action sequence. For example, suppose we define a class *include*. We use `addclasses` to do this.

```
addclasses = ( include othersymbols )
```

The purpose of this would be to allow certain 'excludable actions' to be defined. Actions defined by

```
any.include::
                actions
```

will normally be carried out, because we have defined `include` to be true using `addclasses`. But if cfengine is run in a restricted mode, in which `include` is set to false, we can exclude these actions.

So, by defining the symbol `include` to be false, you can exclude all of the actions which have `include` as a member. There are two ways in which this can be done, one is to negate a class globally using

```
cfengine -N include
```

This undefines the class `include` for the entire duration of the program.

Another way to specify actions is to use a class to select only a subset of all the actions defined in the actionsequence. You do this by adding a class name to one on the actions in action sequence by using a dot '`.`' to separate the words. In this case the symbol only evaluates to 'true' for the duration of the action to which it it attached. Here is an example:

```
links.onlysome
shellcommands.othersymbols.onlysome
```

In the first case `onlysome` *is defined to be true* while this instance of `links` is executed. That means that only actions labelled with the class `onlysome` will be executed as a result of that statement.

In the latter case, both `onlysome` and `othersymbols` are defined to be true for the duration of `shellcommands`.

This syntax would normally be used to omit certain time-consuming actions, such as tidying all home directories. Or perhaps to synchronize certain actions which have to happen in a certain order.

For more advanced uses of cfengine you might want to be able to define a class on the basis of the success or failure of a user-program, a shell command or user script. Consider the following example

```
groups:

   have_cc = ( "/bin/test -f /usr/ucb/cc" "/bin/test -f /local/gnu/cc"  )
```

Note that as of version 1.4.0 of cfengine, you may use the word `classes` as an alias for `groups`. Whenever cfengine meets an object in a class list or variable, which is surrounded by either single, double quotes or reversed quotes, it attempts to execute the string as a command passed to the Bourne shell. If the resulting command has return code zero (proper exit) then the class on the left hand side of the assignment (in this case '`have_cc`') will be true. If the command returns any other value (an error number) the result is false. Since groups are the logical OR of their members (it is sufficient that one of the members matches the current system), the class '`have_cc`' will be defined above if either '`/usr/ucb/cc`' or '`/local/gnu/cc`' exist, or both.

3.4 The generic class any

The generic wildcard `any` may be used to stand for any class. Thus instead of assigning actions for the class `sun4` only you might define actions for any architecture by specifying:

```
any::
      actions
```

If you don't specify any class at all then cfengine assumes a default value of `any` for the class.

3.5 Debugging tips

A useful trick when debugging is to eliminate unwanted actions by changing their class name. Since cfengine assumes that any class it does not understand is the name of some host, it will simply ignore entries it does not recognize. For example:

```
myclass::
```

can be changed to

```
Xmyclass::
```

Since `Xmyclass` no longer matches any defined classes, and is not the name of any host it will simply be ignored. The `-N` option can also be used to the same effect. See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩.

3.6 Access control

It is sometimes convenient to be able to restrict the access of a program to a handful of users. This can be done by adding an access list to the `control:` section of your program. For example,

```
control:
    ...
    access = ( mark root )
```

would cause cfengine to refuse to run the program for any other users except mark and root. Such a restriction would be useful, for instance, if you intended to make set-user-id scripts but only wished certain users to be able to run them. If the access list is absent, all users can execute the program.

Note: if you are running cfengine via the `cfrun` program then cfengine is always started with the same user identity as the cfd process on the remote host. Normally this is the root user identity. This means that the access keyword will have no effect on the use of the command `cfrun`.

3.7 Wildcards in directory names

In the two actions `files` and `tidy` you define directory names at which file checking or tidying searches should start. One economical feature is that you can define a whole group of directories at which identical searches should start in one fell swoop by making use of *wildcards*. For example, the directory names

```
/usr/*/*
/bla/*/ab?/bla
```

represent all of the *directories* (and only directories) which match the above wildcard strings. Cfengine opens each matching directory and iterates the action over all directories which match.

The symbol '`?`' matches any single character, whereas '`*`' matches any number of characters, in accordance with shell file-substitution wildcards.

When this notation is used in directory names, it always defines the starting point for a search. It does not tell the command how to search, only where to begin. The `pattern` directive in `tidy` can be used to specify patterns when tidying files and under `files` all files are considered, See ⟨undefined⟩ [tidy], page ⟨undefined⟩, and ⟨undefined⟩ [files], page ⟨undefined⟩, ⟨undefined⟩ [Recursion], page ⟨undefined⟩.

3.8 File sweeps

File sweeps are searches through a directory tree in which many files are examined and considered for processing in some way. There are many instances where one uses cfengine to perform a file sweep.

- As part of a `files` action, for checking access rights and ownership of files.
- As part of a `tidy` action, for checking files for deletion.
- As part of a `copy` action, while recursively checking whether to copy a file tree.

The problem with file sweeps is that they can be too sweeping! Often you are not interested in examining every single file in a file tree. You might wish to perform a search

* excluding certain named directories and their subdirectories, See ⟨undefined⟩ [ignore], page ⟨undefined⟩.
* excluding certain files and directories matching a specific pattern.
* including only a subset of files matching specific patterns.

The tidy action is slightly different in this respect, since it always expects to match a specific pattern. One is generally not interested in a search which deletes everything except for a named pattern: this would be too dangerous. For this reason, the syntax of `tidy` is different and is documented in the section on tidying, See ⟨undefined⟩ [tidy], page ⟨undefined⟩.

For file sweeps within `files` and `copy` you can specify specific search parameters using the keywords `include=` and `exclude=`. For example,

```
files:

   /usr/local/bin m=0755 exclude=*.ps action=fixall
```

In this example cfengine searches the entire file tree (omitting any directories listed in the ignore-list, See ⟨undefined⟩ [ignore], page ⟨undefined⟩) and omitting any files ending in the extension '`.ps`'.

Specifying the `include=` keyword is slightly different since it automatically restricts the search to only named patterns, whenever you have one or more instances of it. If you include patterns in this way, cfengine ignores any files which do not match the given patterns. It also ignores any patterns which you have specified in the global ignore-list as well as patterns excluded with `exclude=`*pattern*. In other words, exclusions always override inclusions.

If you exclude a pattern or a directory and wish to treat it in some special way, you need to code an explicit check for that pattern as a separate entity. For example, to handle the exluded '`.ps`' files above, you would need to code something like this:

```
files:

   /usr/local/bin m=0644 include=*.ps action=fixall
```

Note: don't be tempted to enclose your wildcards in quotes. The quotes will be treated literally and the pattern might not match the way you would expect.

3.9 Log files written by cfengine

Cfengine keeps two kinds of log-file privately and it allows you to log its activity to syslog. Syslog logging may be switched on with the `Syslog` variable, See ⟨undefined⟩ [Syslog], page ⟨undefined⟩.

The first log cfengine keeps is for every user (every subdirectory of a home directory filesystem). A file `~/.cfengine.rm` keeps a list of all the files which were deleted during the last pass of the `tidy` function. This is useful for users who want to know files have been removed without their blessing. This helps to identify what is happening on the system in case of accidents.

Another file is built when cfengine searches through file trees in the `files` action. This is a list of all programs which are setuid root, or setgid root. Since such files are a potential security risk, cfengine always prints a warning when it encounters a new one (one which is not already in its list). This allows the system administrator to keep a watchful eye over new programs which appear and give users root access. The cfengine log is called `/etc/cfengine/cfengine.log`. The file is not readable for general users.

3.10 Quoted strings

In several cfengine commands, you use quoted strings to define a quantity of text which may contain spaces. For example

```
control:

  macro = ( "mycommand" )

editfiles:

  { $(HOME)/myfile

   AppendIfNoSuchLine 'This text contains space'
  }
```

In each case you may use any one of the three types of quote marks in order to delimit strings,

' *or* " *or* `

If you choose, say ", then you may not use this symbol within the string itself. The same goes for the other types of string delimiters. Unlike the shell, cfengine treats these three delimiters in precisely the same way. There is no difference between them. If you need to quote a quoted string, then you should choose a delimiter which does not conflict with the substring.

Note that you can use special variables for certain symbols in a string See ⟨undefined⟩ [Variable substitution], page ⟨undefined⟩.

3.11 Regular expressions

Regular expressions can be used in cfengine in connection with `editfiles` and `processes` to search for lines matching certain expressions. A regular expression is a generalized wildcard. In cfengine wildcards, you can use the characters '*' and '?' to match any character or number of characters. Regular expressions are more complicated than wildcards, but have far more flexibility.

NOTE: the special characters '`*`' and '`?`' used in wildcards do not have the same meanings as regular expressions!.

Some regular expressions match only a single string. For example, every string which contains no special characters is a regular expression which matches only a string identical to itself. Thus the regular expression '`cfengine`' would match only the string "cfengine", not "Cfengine" or "cfengin" etc. Other regular expressions could match more general strings. For instance, the regular expression '`c*`' matches any number of c's (including none). Thus this expression would match the empty string, "c", "cccc", "ccccccccc", but not "cccx".

Here is a list of regular expression special characters and operators.

'\' The backslash character normally has a special purpose: either to introduce a special command, or to tell the expression interpreter that the next character is not to be treated as a special character. The backslash character stands for itself only when protected by square brackets `[\]` or quoted with a backslash itself '\\'.

'\b' Matches word boundary operator.

'\B' Match within a word (operator).

'\<' Match beginning of word.

'\>' Match end of word.

'\w' Match a character which can be part of a word.

'\W' Match a character which cannot be part of a word.

'*any character*'
Matches itself.

'.' Matches any character

'*' Match zero or more instances of the previous object. e.g. 'c*'. If no object precedes it, it represents a literal asterisk.

'+' Match one or more instances of the preceding object.

'?' Match zero or one instance of the preceding object.

'{ }' Number of matches operator. '{5}' would match exactly 5 instances of the previous object. '{6,}' would match at least 6 instances of the previous object. '{7,12}' would match at least 7 instances of, but no more than 12 instances of the preceding object. Clearly the first number must be less than the second to make a valid search expression.

'|' The logical OR operator, OR's any two regular expressions.

'[*list*]' Defines a list of characters which are to be considered as a single object (ORed). e.g. '[a-z]' matches any character in the range a to z, 'abcd' matches either a, b, c or d. Most characters are ordinary inside a list, but there are some exceptions: ']' ends the list unless it is the first item, '\' quotes the next character, '[:' and ':]' define a character class operator (see below), and '-' represents a range of characters unless it is the first or last character in the list.

'[^*list*]' Defines a list of characters which are NOT to be matched. i.e. match any character except those in the list.

''[:*class*:]''
Defines a class of characters, using the ctype-library.

alnum Alpha numeric character

alpha An alphabetic character

blank A space or a TAB

cntrl A control character.

digit 0-9

graph same as print, without space

lower a lower case letter

print printable characters (non control characters) `punct` neither control nor alphanumeric symbols

`space` space, carriage return, line-feed, vertical tab and form-feed.

`upper` upper case letter

`xdigit` a hexadecimal digit 0-9, a-f

'`(..)`' Groups together any number of operators.

'`\`*digit*' Back-reference operator (refer to the GNU regex documentation).

'`^`' Match start of a line.

'`$`' Match the end of a line.

Here is a few examples. Remember that some commands look for a regular expression match of part of a string, while others require a match of the entire string See ⟨undefined⟩ [editfiles], page ⟨undefined⟩.

```
^#       match string beginning with the # symbol
^[^#]     match string not beginning with the # symbol
^[A-Z].+  match a string beginning with an uppercase letter
          followed by at least one other character
```

3.12 Iterating over lists

Shell list variables are normally defined by joining together a list of directories using a concatenation character such as '`:`'. A typical example of this is the PATH variable:

```
PATH=/usr/bin:/usr/local/bin:/usr/sbin
```

It is convenient to be able to use such variables to force cfengine to iterative over a list. This gives us a compact way of writing repeated operations and it allows a simple method of communication with the shell environment. For security reasons, iteration is supported only in the following contexts:

- in the 'to' field of a multiple link action,
- in the 'from' field of a copy action,
- in the directory field of a tidy action,
- in the directory field of the files action,
- in the ignore action,
- in a shell command.

This typically allows communication with PATH-like environment variables in the shell.

In these contexts, any variable which has the form of a list joined together by colons will be iterated over at compilation time. Note that you can change the value of the list separator using the `split` variable in the control section of the program See ⟨undefined⟩ [split], page ⟨undefined⟩.

For example, to link all of the binary files in the PATH environment variable to a single directory, tidying dead links in the process, you would write

```
control:

  actionsequence = ( links tidy )

links:

  /allbin +> $(PATH)

tidy:

  # Hopefully no-match matches nothing

  /allbin pattern=no-match age=0 links=tidy
```

no-match is not a reserved word in cfengine, this is just a string you do not expect to match any file.

Alternatively, you might want to define an internal list using a space as a separator:

```
control:

   split = ( " " )

   mylist = ( "mark ricky bad-dude" )

tidy:

   /mnt/home1/$(mylist) pattern=*.cfsaved age=1
```

This example iterates the tidy action over the directories '`/mnt/home1/mark`', '`/mnt/home1/ricky`' and '`/mnt/home1/bad-dude`'.

The number of list variables in any path or filename should normally be restricted to one or two, since the haphazard combination of two lists will seldom lead to any meaningful pattern. The only obvious exception is perhaps to iterate over a common set of child-directories like '`bin`', '`lib`' etc in several different package directories.

4 Designing a global system configuration

This chapter is about building strategies for putting together a site configuration for your entire network.

4.1 General considerations

In order to use any system administration tool successfully, you have to make peace with your system by deciding exactly what you expect and what you are willing to do to achieve the results. You need to decide what you will consider to be acceptable and what is to be considered completely untenable. You need to make these decisions because otherwise you will only be confused later when things don't go the way you expected.

Experience shows that the most successful policies for automation involve keeping everything as simple as possible. The more uniform or alike your machines are, the easier they are to run and the happier users are. Sometimes people claim that they need such great flexibility that all their machines should be different. This belief tends to be inversely proportional to the number of machines they run and generally only applies to very special development environments! Usually you will only need one or to machines to be special and most can be made very similar.

Site configuration is about sharing and controlling resources. The resources include disks (filespace), files, data, programs, passwords and physical machines. Before planning your sitewide configuration you should spend some time deciding how you would like things to work.

In the remaining parts of this chapter, you will find some hints and tips about how to proceed, but remember that when push comes to shove, you must make your own choices.

4.2 Using netgroups

If you use the network information service (NIS) on your local network then you may already have defined *netgroups* consisting of lists of hosts which belong to specific owners at your site. If you have, then you can use these groups within cfengine. This means that you can use the same groups in the `/etc/exports` file as you use to define the mount groups and classes. See ⟨undefined⟩ [groups], page ⟨undefined⟩.

A netgroup is a list of hostnames or user names which are registered in the network information service (NIS) database under a specific name. In our case we shall only be interested in lists of hostnames.

To make a netgroup you need to define a list in the file `/etc/netgroup` on your NIS server. If you are not the NIS administrator, you will have to ask to have a netgroup installed. The form of a netgroup list of hosts is:

```
mylist-name      (host1,,) (host2,,) (host3,,) (host4,,)

norway-sun4-host (saga,,) (tor,,) (odin,,)
foes-linux-hosts (borg,,)
```

Each list item has three entries, but only the first is relevant for a host list. See the manual pages on netgroups for a full explanation of the meaning of these fields.

The usefulness of netgroups is that they can be used to stand for a list of hostnames in system files like '/etc/exports'. This compresses the amount of text in this file from a long list to a single name. It also means that if you use the same list of hosts from a netgroup inside cfengine when defining groups and classes, you can be sure that you are always using the same list. In particular it means that you don't have to update multiple copies of a list of hosts.

The netgroups can now be used in cfengine programs by using the `+` or `@+` symbols in the `groups` section. See ⟨undefined⟩ [groups], page ⟨undefined⟩.

4.3 Files and links

File and link management takes several forms. Actions are divided into three categories called `files`, `tidy` and `links`. The first of these is used to check the existence of, the ownership and permissions of files. The second concerns the systematic deletion of garbage files. The third is a link manager which tests, makes and destroys links. The monitoring of file access bits and ownership can be set up for individual files and for directory trees, with controlled recursion. Files which do not meet the specified criteria can be 'fixed' —i.e. automatically set to the correct permissions, or can simply be brought to the attention of the system administrator by a warning. The syntax of such a command is as follows:

```
files:

  class::

    /path mode=mode owner=owner group=group

         recurse=no-of-levels action=action
```

The directory or file name is the point at which cfengine begins looking for files. From this point the search for files proceeds recursively into subdirectories with a maximum limit set by the `recurse` directive, and various options for dealing with symbolic links and device boundaries. The mode-string defines the allowed file-mode (by analogy with '`chmod`') and the owner and group may specify lists of acceptable user-ids and group-ids. The action taken in response to a file which does not meet acceptable criteria is specified in the action directive. It includes warning about or directly fixing all files, or plain files or directories only. Safe defaults exist for these directives so that in practice they may be treated as options.

For example,

```
files:

  any::
       /usr/*/bin mode=a+rx,o-w own=root r=inf act=fixall
```

which (in abbreviated form) would check recursively all files and directories starting from directories matching the wildcard (e.g. '`/usr/local/bin`', '`/usr/ucb/bin`'). By default, `fixall` causes the permissions and ownership of the files to be fixed without further warning.

One problem with symbolic links is that the files they point to can get deleted leaving a 'hanging pointer'. Since cfengine can make many hundreds of links without any effort, there is the danger that, in time, the system could become full of links which don't point anywhere. To combat this problem, you can set the option `links=tidy` in the files section. If this is set, cfengine will remove any symbolic links which do not point to existing files See ⟨undefined⟩ [files], page ⟨undefined⟩.

The creation of symbolic links is illustrated in figure 1 and the checking algorithm was discussed in section 2. In addition to the creation of single links, one may also specify the creation of multiple links with a single command. The command

```
links:

   binaryhost::

      /local/elm/bin +> /local/bin
```

links all of the files in '`/local/elm/bin`' to corresponding files in '`/local/bin`'. This provides, amongst other things, one simple way of installing software packages in regular 'bin' directories without controlling users' PATH variable. A further facility makes use of cfengine's knowledge of available (mounted) binary resources to search for matches to specific links. Readers are referred to the full documentation concerning this feature.

The need to tidy junk files has become increasingly evident during the history of cfengine. Files build up quickly in areas like '`/tmp`', '`/var/tmp`'. Many users use these areas for receiving large ftp-files so that their disk usage will not be noticed! To give another example, just in the last few months the arrival of netscape World Wide Web client, with its caching facilities, has flooded hard-disks at Oslo with hundreds of megabytes of WWW files. In addition the regular appearance of '`core`' files[1] and compilation by-products ('`.o`' files and '`.log`' files etc.) fills disks with large files which many users do not understand. The problem is easily remedied by a few lines in the cfengine configuration. Files can be deleted if they have not been accessed for n-days. Recursive searches are both possible and highly practical here. In following example:

```
tidy:

   AllHomeServers::

      home                 pattern=core       r=inf age=0
      home/.wastebacket    pattern=*          r=inf age=14
      home/.netscape-cache pattern=cache????* r=inf age=2
      home/.MCOM-cache     pattern=cache????* r=inf age=2
      home/.netscape       pattern=cache????*  r=inf age=2
```

all hosts in the group '`AllHomeServers`' are instructed to iterate over all users' home directories (using the wildcard `home`) and look for files matching special patterns. Cfengine tests the *access time* of files and deletes only files older than the specified limits. Hence all core files, in this example, are deleted immediately, whereas files in the subdirectory '`.wastebasket`' are deleted only after they have lain there untouched for 14 days, and so on.

As a system administrator you should, of course, exercise great caution when making rules which can delete users' files. A single slip of the hand can result in a rule which will irretrievably delete files.

[1] On some systems, core dumps cannot be switched off!

When making a 'tidy' strategy you should probably coordinate with your backup policy. You should not delete files until after you have taken a backup, so that — if the worst should happen — you are covered against possible accidents.

Cfengine helps to some extent to keep track of what files it deletes. When tidying users' home directories it creates a log file of all files which were deleted on the last tidy operation. This log is called `~/.cfengine.rm`.

You might consider tidying certain files only once a week, in which case a command such as

```
tidy:

   AllHomeServers.Sunday::

         files to tidy
```

could be useful. Nonsense files, such as 'core' files could be tidied every night.

NOTE! Be careful when telling cfengine to delete core files. If you write a wildcard like `core*`*, then you could risk deleting important system files such as* `core.h`*.*

4.4 Copying files

The administration of a system often requires the copying of files. The reason for this is usually that we would like to distribute a copy of a particular file, from some master location and ensure that all of the copies are up to date. Another use for this is to install software from one directory (perhaps on a CD ROM) to another.

Cfengine helps this process by allowing you to copy a single file or a file tree, from one directory to another, perhaps checking the permissions and owners of a file to adjust the copies in some special way. The files are checked by cfengine using one of two methods.

- A date-stamp comparison with a master file, using last-change times, can be used to tell cfengine to recopy a file from the master if the master file is newer than the copy.
- A checksum can be computed for each file and compared with one for the master file. If the contents of the copy file differs in any way from the master, the file will be re-copied.

Cfengine allows you to do the following

- Copy a single file to another file in a different location, perhaps with a new name, new permissions and a different owner.
- Copy a single file to all users on the system, changing the owner of the file for each user automatically. (This could be used to distribute and update standard setup files.)
- Recursively copy an entire file tree, omitting files which match a list of wildcard-patterns, or linking certain files instead of copying.

You can find out more about copying in the reference section See ⟨undefined⟩ [copy], page ⟨undefined⟩.

4.5 Managing processes

Cfengine allows you to check for the existence of processes on your system, send those processes signals (such as kill) and perhaps restart those processes. Typical applications for this are sending '`cron`' and '`inetd`' the HUP signal, after editing their configuration files, or killing unwanted processes (such as user programs which hog the system at peak usage times).

You can read more about this in the reference section See ⟨undefined⟩ [processes], page ⟨undefined⟩.

4.6 Cfengine's model for NFS-mounted filesystems

Most of the filesystems that you will want to make available across the network are going to fall into one of two categories. In cfengine parlance these are called *home directories* and *binary directories*. A home directory is a place where users' login directories are kept. This is traditionally a directory called '`/home`' or '`/users`' or some subdirectory of these. A binary directory is a place where compiled software is kept. Such files (which do not belong to the pure operating system release) are often placed in a directory called '`/usr/local`' or simply '`/local`'.

In this chapter we shall consider a scheme for using cfengine to make NFS filesystem management quite painless.

4.6.1 NFS filesystem resources

Using the Network File System (NFS) in a large workstation environment requires a bit of planning. The idea of NFS is to share files on one host with other hosts. In most cases, filesystems to be shared across the network fall into two categories: *binary* filesystems (those which contain compiled software) and *user* or *home* filesystems (which contain users' login areas).

The most simple minded way to share resources would be to mount every resource (each available NFS filesystem) onto every host. To avoid collisions, each filesystem would have to have a unique name. This is one possibility, but not a very intelligent one. As experienced users will realize, cross-mounting too many NFS filesystems is a recipe for all kinds of trouble.

Cfengine offers a simple model which can help you pick out only the resources you need from the list of NFS filesystems. It will then mount them automatically and edit the appropriate filesystem tables. It does this by defining classes of hosts. For instance — you really don't need to mount a binary filesystem for an `ultrix` system onto an `HPUX` system. There would be no point — binary resources are *architecture* or *hard-class dependent*. But home directories are architecture independent.

Cfengine lets you to define a list of allowed servers for various hosts so that only filesystems from the servers will be considered for mounting!

4.6.2 Unique filesystem mountpoints

The first step towards treating NFS filesystems as network resources is to invent a naming scheme so that every filesystem has a unique name on which it can be mounted. If we don't sort this out now, we could find two or more hosts with a filesystem called `/usr/local`, both of which we might like to mount since they contain different software.

A simple but extremely useful naming scheme is the following.[2] If you don't like this scheme you can invent your own, but the remainder of the text will encourage you to use this one. If you follow this scheme, exactly as described here, you will never have any problems with mount points. We shall describe the scheme in detail below. Here are some points to digest:

- When mounting a remote filesystem on your local system, the local and remote directories should always have exactly the same name.
- The name of every filesystem mountpoint should be unique and tell you something meaningful about where it is located and what its function is.
- You can always make links from special unique names to more general names like '`/usr/local`'. If you this involves compiled software and you do this on one host, you should do it on others which are of the same type.
- It doesn't matter whether software compiles in the path names of special directories into software as long as you follow the points above.

Each filesystem is given a directory name composed of three parts:

```
/site/host/contents
```

The first directory (which only exists to create a suitable mountpoint) is the name of your local site. If you are a physics department at a university (with a separate setup) you could call this 'physics'. It could be your company name or whatever. The second piece is the name of the host to which the disk space is physically attached. The final piece is the name of the filesystem. Here are some typical examples:

```
/physics/einstein/local    # /usr/local for einstein@physics
/physics/newton/u1         # user partition 1 for newton@physics
```

On the machines which are home to the 'local' partition, it is better to make a link to `/usr/local` than call the filesystem `/usr/local` directly. This is because it makes the procedure of organizing the entire network much clearer.

It is worth noting that, when you ask cfengine to mount such a resource, it will automatically make the mount directory and can easily be asked to make a link to `/usr/local`, so this small amount of extra work is really no work at all.

The whole naming convention is compactly summarized by defining a mount point variable. See ⟨undefined⟩ [mountpattern], page ⟨undefined⟩. With the present scheme, this can be defined as

```
mountpattern = ( /$(site)/$(host) )
```

so that it evaluates to the name of the host executing the file regardless of who that may be. This variable is used together with the `homepattern` pattern variable, which is used to distinguish between home directories and binary resources. See ⟨undefined⟩ [homepattern], page ⟨undefined⟩. You can think of this as being part of the naming convention. In this text, we use the convention `u1 u2 u3...` for home disks. You could equally well use `home1 home2...` etc. As long as the name is unique, it doesn't matter.

The full list of named resources should now be listed in the `mountables` list, which is simply a list of all the resources available for mounting on the network. See ⟨undefined⟩ [mountables], page ⟨undefined⟩.

[2] This unique naming scheme was suggested to me originally by Knut Borge at USIT of the University of Oslo.

4.6.3 How does it work?

Once you have defined your unique names, how does cfengine know what to mount? The idea is now to define a list of servers for each class of hosts. See ⟨undefined⟩ [binservers], page ⟨undefined⟩.

Suppose we make a `binserver` declaration:

```
binservers:

  mygroup.sun4::

     einstein
     newton
```

This would tell cfengine that it should mount all binary resources from hosts `einstein` or `newton` onto any host of type `sun4` in the group `mygroup`. Every filesystem which is listed in `mountables` and is not a home directory will be mounted. See ⟨undefined⟩ [mountables], page ⟨undefined⟩.

Home directories and binary resources are kept separate automatically by cfengine, because a home directory is one whose contents-name matches the `homepattern` pattern variable. See ⟨undefined⟩ [Unique filesystem mountpoints], page ⟨undefined⟩.

A `homeserver` declaration:

```
homeservers:

  mygroup::

     einstein
     newton
     schwinger
     feynman
```

would correspondingly mean mount all the home directory resources on the hosts in the list on all hosts in the group `mygroup`. Clearly it is unnecessary to distinguish between the architecture platform types of the actual servers for user directories.

In each case, cfengine will mount filesystems, make the appropriate directories for the mount point and edit the filesystem table. See ⟨undefined⟩ [actionsequence], page ⟨undefined⟩.

4.6.4 Special variables

Once you have mounted a resource on a unique directory, you have access to all of the relevant filesystems on your network — but you really wanted the 'local' filesystem to be mounted on `/usr/local`. All you need do now is to make a link:

```
links:

  any::
```

```
/usr/local  -> /$(site)/$(binserver)/local
```

The meaning of this is that, on any host, the directory `/usr/local` should be a link to the 'nearest' binary server's 'local' resource. The `$(binserver)` variable can in principle expand to any binary server in the list. In practice, cfengine goes through the list in order and picks the first filesystem resource which matches.

Could this lead to a collision? Suppose we are on the host 'einstein' and we execute the above command. The host 'einstein' has a filesystem `/physics/einstein/local` on its local disk — it is in fact the binary server for the network, so it certainly doesn't need to mount any NFS filesystems. But this is no problem because cfengine automatically treats `$(host)` as the highest priority binary server for any host. That means that if you have a local filesystem, it will always have priority.

In contrast, if the host 'schwinger' ran the command above, it would find no local filesystem called `/physics/schwinger/local`, so it would go along the list of defined binary servers, find 'einstein' and try again. It will succeed in finding 'einstein' *provided all the binary servers were mounted before the link command is executed*. This means that you should structure the `actionsequence` so that all filesystems are mounted before any links are made.

With a little practice, the cfengine model can lead to an enormous simplification of the issue of NFS-mountable resources.

NOTE: cfengine does not try to export filesystems, only mount already exported filesystems. If you want to automate this procedure also, you can use the `editfiles` facility to add entries to '`/etc/exports`' See ⟨undefined⟩ [editfiles], page ⟨undefined⟩. In practice this is very difficult to do and perhaps not desirable.

4.6.5 Example programs for mounting resources

Let's write a very simple configuration for a network with only one server called hal, where all the hosts are of the same operating system type. In such an example we can avoid using classes altogether.

```
control:

  site   = ( univ )
  domain = ( univ.edu )

  actionsequence =
     (
     mountall
     mountinfo
     addmounts
     mountall
     links
     )

binservers:

   hal
```

```
homeservers:

   hal

mailserver:

   hal:/var/spool/mail

mountables:

   hal:/univ/home1
   hal:/univ/home2
   hal:/univ/local

links:

   /usr/local -> /univ/local
```

In this example, we have only one type of host so the configuration is the same for each of them: no class references are required. If we look through the action sequence we see that the program first mounts all the filesystems which are already defined on each host. It does this to be sure that everything which is already set up to be mounted is mounted. Let's assume that there are no problems with this.

The next thing that happens is that `mountinfo` builds a list of the filesystems which each host has successfully mounted. Then by calling `addmounts` we ask cfengine to check whether the host is missing any filesystems. What happens is that cfengine first looks to see what servers are defined for each host. In this case all hosts on the network have only one server: hal. Hal is defined as a server for both binary data and 'home' data — i.e. users' home directories. The list `mountables` tells cfengine what filesystems are available over the network for the server hal. There are three filesystems which can be mounted, called '`/univ/home1`', '`/univ/home2`' and '`/univ/local`'. Cfengine checks to see whether each of these filesystems is mounted and, if not, it builds the necessary directories, edits the necessary files and mounts the filesystems.

Finally we come to `links` in the action sequence. This tells cfengine to look at the defined links. There is one link defined: a link from '`/usr/local`' to the mounted filesystem '`/univ/local`'. Cfengine checks and tries to make the link if necessary. If all goes well, each host on the network should now have at least three filesystems mounted and a link from '`/usr/local`' to '`/univ/local`'.

Here is another simple example program for checking and automatically mounting an NFS based `/usr/local` and all home directories onto all hosts on a small network. Here we have several servers and must therefore use some classes.

```
#
#  Mounts
#


control:
```

```
   site      = ( mysite )
   domain    = ( mysite.country )
   sysadm    = ( mark )
   netmask   = ( 255.255.255.0 )

   actionsequence =
      (
      mountall
      mountinfo
      addmounts
      mountall
      links
      )

   mountpattern = ( /$(site)/$(host) )
   homepattern   = ( u? )               # u1 u2 u3 etc..

groups:

   MyGroup =
      (
      host1
      host2
      binserver1
      binserver2
      )

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

homeservers:

   MyGroup:: host1


binservers:

   MyGroup.sun4::   server1
   MyGroup.ultrix:: server2

mailserver:

   host1:/usr/spool/mail

mountables:

   host1:/mysite/host1/u1
   host1:/mysite/host1/u2
   server1:/mysite/server1/local
```

```
    server2:/mysite/server2/local


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

links:

      /usr/local  -> /${site}/${binserver}/local
```

Let's suppose we run this program on host2 which is an ultrix machine. This host belongs to the class `mygroup` and the hard-class `ultrix`. This tells us that its homeserver is host1, its binary server is server2 and its mailserver is host1. Moreover, since the homepattern matches any filesystem ending in u-something, it recognizes the two home directories in the mountables list — and therefore the two binary directories also.

The action sequence starts by mounting all of the filesystems currently in the filesystem table '`/etc/fstab`'. It then scans the list of mounted filesystems to find out what is actually mounted. Since the homeserver is host1, we know that our host has to mount all home-filesystems from this server, so it checks for '`host1:/mysite/host1/u1`' and '`host1:/mysite/host1/u2`'. If they are not present they are added to '`/etc/fstab`'[3]. Next, we know that the binary server is server1, so we should check for '`server1:/mysite/server1/local`'. The mail server is also checked for and added if necessary. Cfengine then tries to mount all filesystems once again, so that the new filesystems should be added.

Note that, in the process of adding the filesystems to '`/etc/fstab`', cfengine creates the directories up to and including the point at which the filesystems should be mounted. If something prevents this — if we try to mount on top of a plain file for instance — then this will result in an error.

Finally, we reach the link section and we try to expand the variables. `$(site)` expands to '`mysite`'. `$(binserver)` expands first to the hostname (host2), but '`/mysite/host2/local`' does not exist, so it then goes to the binserver list, which substitutes server1 for the value of `$(binserver)`. Since '`/mysite/server1/local`' does exist and is now mounted, cfengine makes a link to this directory from '`/usr/local`'. The script is then completed.

If the script is run again, everything should now be in place so nothing happens. If for some reason it failed the first time, it will fail again. At any rate it will either do the job once and for all or signal an error which must be corrected by human intervention[4].

4.7 Using the automounter

The automounter is a daemon based service which replaces static mounting of NFS filesystems with a dynamical model. When the automounter is running, filesystems are mounted only when a user tries to access a file which resides on one of those filesystem. After a given period (usually five

---

[3] Note: if the filesystem was in the fstab but not actually mounted a warning is issued telling you that the filesystem was probably not exported correctly on host1.

[4] One possibility is that an NFS filesystem cannot be mounted because the host serving the filesystem is out of service. If this is the case then a subsequent re-run when the server resumes normal service will succeed.

minutes) any filesystem which has not been accessed is unmounted. The advantage of this scenario is that hanging servers do not affect the behaviour of hosts which mount their filesystems, unless a specific file is being accessed. In both cases, filesystems must be exported in order to be mountable.

It is not the purpose of this section to explain the use of the automounter in detail, only to offer hints as to how cfengine can be used to simplify and rationalize automount configuration for the already initiated. Let us begin by comparing the behaviour of the automounter with the cfengine model for mounted filesystems.

The automounter is designed to be used together with a global configuration file, distributed by NIS (the network information service). As such, all hosts read the same configuration file. This makes it appear as though all hosts end up mounting every filesystem in the automount configuration database, but this is not so in practice because filesystems are only mounted if required. Thus a system which does not require a filesystem will not attempt to mount it. Moreover, the existence of a global configuration file does not affect which hosts have the right to mount certain filesystems (which is specified by exports or share on the relevant server), thus a request to mount a non-exported filesystem will result in an access denial. The automounter is configured locally on each host in files named '`/etc/auto_master`', '`auto_direct`' etc.

In the cfengine static mounting scheme, you define a list of binary and home servers. The filesystem table is modified on the basis of these decisions, and filesystems are only added if cfengine deems it appropriate to mount them on a given host. The idea here is to minimize the number of filesystems mounted to those which are known to be required. Again the issue of access permissions must be arranged separately. These filesystems are placed directly in '`/etc/fstab`', or the equivalent for your system.

From cfengine, you can use the automounter instead of the static mount model by

- omitting `addmounts`, `mountinfo`, `mountall` from the actionsequence, in the control part of your cfengine program,
- using `editfiles` to edit the relevant configuration files such as '`/etc/auto_master`', or '`auto_direct`' etc,
- using the `AutomountDirectResources` command in editfiles to dump the list of cfengine class-based list of mountables into a file of your choice in the correct format for autmount's direct maps,
- using `processes` See ⟨undefined⟩ [processes], page ⟨undefined⟩ to restart the automounter (send the hangup signal `hup`), or perhaps stop and restart the daemon by sending the `term` signal (you should never send the `kill` signal),
- using the multiple link facilities to link in indirect mounted filesystems as required, and `files` or `tidy` to clean up stale links afterwards,
- perhaps using `copy` to distribute basic automount configuration files to multiple systems.

The automounter was created to solve certain problems which cfengine now solves (in the author's opinion) better. For example, the use of the 'hosts' map in the automounter mounts filesystems like '`/usr/local`' on different (uniquely named) mountpoints for each host in order to avoid name space collisions. Using cfengine and a unique naming scheme, you can achieve the same thing more cleanly, without all of the gratuitous linking and unlinking which the automounter performs by itself. Moreover, the idea of a unique name-space is better practice and more in keeping with new global filesystem ideas such as AFS and DFS. The only advantage of the automounter is that one avoids the annoying error messages from hung servers about "NFS server not responding". In that respect, it seems sensible to use only direct mounts and a unique name space.

Some systems advocate grouping all users' login (home) directories under a common directory called '/home' or 'users'. The automounter goes through all manner of contortions to achieve this task. If you use a unique naming scheme like the one advocated here, this is a trivial task. You simply arrange to mount or automount all user directories, such as

```
/site/host/home1
/site/host/home2
...
```

and then link them as follows:

```
/home +> /site/host/home1
/home +> /site/host/home2
...
```

Finally, you should be aware that the automounter does not like to be mixed with static mount and unmount operations. Automounted filesystems take priority over statically mounted filesystems, but the automounter can be confused by manually mounting or unmounting filesystems while it is running.

4.8 Editing Files

A very convenient characteristic of BSD/System 5 systems is that they are configured primarily by human-readable textfiles. This makes it easy for humans to configure the system and it also simplifies the automation of the procedure. Most configuration files are line-based text files, a fact which explains the popularity of, for example, the Perl programming language. Cfengine does not attempt to compete with Perl or its peers. Its internal editing functions operate at a higher level which are designed for transparency rather than flexibility. Fortunately most editing operations involve appending a few lines to a file, commenting out certain lines or deleting lines.

For example, some administrators consider the finger service to be a threat to security and want to disable it. This could be done as follows.

```
editfiles:

      { /etc/inetd.conf

      HashCommentLinesContaining "finger"
      }
```

Commands containing the word 'Comment' are used to 'comment out' certain lines from a text-file—i.e. render a line impotent without actually deleting it. Three types of comment were supported originally: shell style (hash) '#', '%' as used in TeX and on AIX systems, and C++-style '//'.

A more flexible way of commenting is also possible, using directives which first define strings which signify the start of a comment and the end of a comment. A single command can then be used to render a comment. The default values of the comment-start string is '# ' and the default comment-end string is the empty string. For instance, to define C style comments you could write:

```
{ file

SetCommentStart "/* "
SetCommentEnd   " */"

# Comment out all lines containing printf!

CommentLinesMatching ".*printf.*"
}
```

Other applications for these editing commands include monitoring and controlling root-access to hosts by editing files such as '`.rhosts`' and setting up standard environment variables in global shell resource files— for example, to set the timezone. You can use the editing feature to update and distribute the message of the day file, or to configure sendmail, See ⟨undefined⟩ [FAQS and Tips], page ⟨undefined⟩.

An extremely powerful feature of cfengine is the ability to edit a similar file belonging to every user in the system. For example, as a system administrator, you sometimes need to ensure that users have a sensible login environment. Changes in the system might require all users to define a new environment variable, for instance. This is achieved the with `home` pseudo-wildcard. If one writes

```
{ home/.cshrc

AppendIfNoSuchLine "# Sys admin/cfengine: put next line here"
AppendIfNoSuchLine "setenv PRINTER newprinter"
}
```

then the users' files are checked one-by-one for the given lines of text, and edited if necessary.

Files are loaded into cfengine and edited in memory. They are only saved again if modifications to the file are carried out, in which case the old file is preserved by adding a suffix to the filename. When files are edited, cfengine generates a warning for the administrator's inspection so that the reason for the change can be investigated.

The behaviour of cfengine should not be confused with that of *sed* or *perl*. Some functionality is reproduced for convenience, but the specific functions have been chosen on the basis of (i) their readability and (ii) the fact that they are 'frequently-required-functions'. A typical file editing session involves the following points:

- Load file into memory.
- Is the size of the file within sensible user-definable limits? If not, file could be binary, refuse to edit.
- Check each editing command and count the number of edits made.
- If number of edits is greater than zero, rename the old file and save the edited version in its place. Inform about the edit.
- If no edits are made, do nothing, say nothing.

Equivalent one-line sed operations involve editing the same file perhaps many times to achieve the same results—without the safety checks in addition.

4.9 Disabling and the file repository

The existence of certain files can compromise the integrity of your system and you may wish to ensure that they do not exist. For example, some manufacturers sell their workstations with a '+' symbol in the file '/etc/hosts.equiv'. This means that anyone in your NIS domain has password free access to the system!! Since this is probably not a good idea, you will want to disable this file by renaming it, or simply deleting it.

```
disable:

   /etc/hosts.equiv
```

Other files compromise the system because they grow so large that they fill an entire disk partition. This is typically true of log files such as the system 5 files '/var/adm/wtmpx' and '/var/lp/logs/lpsched'. Other files like */var/adm/messages* get "rotated" by the system so that they do not grow so large as to fill the disk. You can make cfengine rotate these files too, by writing

```
disable:

    Sunday::

    /var/lp/logs/lpsched  rotate=3
```

Now, when cfengine is run, it renamed the file 'lpsched' to a file called 'lpsched.1'. It also renames 'lpsched.1' as 'lpsched.2' and so on, until a maximum of 3 files are kept. After passing 3, the files 'fall off the end' and are deleted permanently. This procedure prevents any log files from growing too large. If you are not interested in keeping back-logs, then you may write `rotate=empty` and cfengine will simply empty the log file.

When ever cfengine disables a file (`disable` or `links` with the '!' operator), or saves a new file on top of an old one (`copy` or `editfiles`), it makes a backup of the original. Usually disabled files are renamed by appending the string '.cfdisabled' the the filename; copied files are saved by appending the string '.cfsaved'. It is possible to switch off backup file generation in the copy feature by setting the variable `backup=false`, but a better way of managing disabled and backed-up files is to use a directory in which you collect all such files for the whole system. This directory is called the file repository and is set in the control part of the program, as follows:

```
control:

   repository = ( directory-name )
```

If this variable is defined, cfengine collects all backup and disabled files (except for rotated files) in this directory, using a unique pathname. You can then inspect these files in the repository and arrange to tidy the repository for old files which are no longer interesting.

4.10 Running user scripts

Above all, the aim of cfengine is to present a simple interface to system administrators. The actions which are built into the engine are aimed at solving the most pressing problems, not at solving every problem. In many cases administrators will still need to write scripts to carry out more specific tasks. These scripts can still be profitably run from cfengine. Variables and macros defined in cfengine can be passed to scripts so that scripts can make maximal advantage of the class based decisions. Also note that, since the days of the week are also classes in cfengine, it is straightforward to run weekly scripts from the cfengine environment (assuming that the configuration program is executed daily). An obvious use for this is to update databases, like the fast-find database one day of the week, or to run quota checks on disks.

```
shellcommands:

   myhost.Sunday::

      "/usr/bin/find/updatedb"
```

Cfengine scripts can be passed variables using normal variable substitution:

```
control:

   cfbin     = ( /local/gnu/lib/cfengine/bin )
   backupdir = ( /iu/dax/backup )

shellcommands:

  "$(cfbin)/cfbackup -p -f $(backupdir) -s /iu/nexus/u1"
```

If you need to write a particularly complex script to expand cfengine's capabilities, it might be useful to have full access to the defined classes. You can do this in one of two ways:

- Pass the variable `$(allclasses)` to the script. This contains a list of all classes in the form of a string

  `CFALLCLASSES=`*class1*`:`*class2*`:...`

  This variable always contains an up to date list of the defined classes.
- Use the command line option '`-u`' or '`--use-env`'. When this is defined, cfengine defines an internal environment variable called '`CFALLCLASSES`' which contains the same list as above. Unfortunately, system 5 boxes don't seem to like having to update an environment variable continuously and tend to dump core, so this is not the default behaviour!

4.11 Compressing old log files

In the previous two sections we have looked at how to rotate old log files and how to execute shell commands. If you keep a lot of old log files around on your system, you might want to compress them so that they don't take up so much space. You can do this with a shell command. The example below looks for files matching a shell wildcard. Names of the form '`file.1`', '`file.2`'...'`file.10`' will match this wildcard and the compression program sees that they get compressed. The output is dumped to avoid spurious messages.

```
shellcommands:

  "$(gnu)/gzip /var/log/*.[0-9] /var/log/*.[0-9][0-9]  > /dev/null 2>&1"
```

Cfengine will also recognize rotated files if they have been compressed, with suffixes '.Z', '.gz', '.rbz' or '.rbz'.

4.12 Managing ACLs

Access control lists are extended file permissions. They allow you to open or close a file to a named list of users (without having to create a special group for those users). They also allow you to open or close a file for a named list of groups. Several unix-like operating systems have had access control lists for some time; but they do not seem to have caught on.

There is a number of reasons for this dawdling in the past. The tools for setting ACLs are generally interactive and awkward to use. Because a named list of users would lead to excessive verbosity in an `ls -l` listing, one does not normally see them. There is therefore the danger that the hidden information would lead to undetected blunders in opening files to the wrong users. ACLs are also different on every vendor's filesystems and they don't work over intersystem NFS. In spite of these reservations, ACLs are a great idea. Here at Oslo College, it seems that users are continually asking how they can open a file just for the one or two persons they wish to collaborate with. They have grown used to Novell/PC networks which embraced the technology from Apollo/NCS much earlier. Previously the Unix answer to users has always been: go ask the system administrator to make a special group for you. Then do the '`chmod`' thing. And then they would say: so what's so great about this Unix then?

Addressing this lack of standardization has been the job of a POSIX draft committee. Some vendors have made their implementations in the image of this draft. Solaris 2.6 has a good implementation. In spite of this, even these systems have only awkard tools for manipulating ACLs. Not the kind of thing you want to be around much, if you have better things to do. But the incompatibility argument applies only to multiple vendor headbutting. Some institutions who share data on a global basis opt for advanced solutions to network filesystems, such as AFS and DFS. Filesystems such as DCE's DFS make extensive use of file ACLs, and they are not operating system specific. Even so, DFS provides only interactive tools for examining and setting file permissions, and this is of little use to system administrators who would rather relegate that sort of thing to a script.

The need for this kind of thing is clear. Systems which make use of ACLs for security can be brought to their knees by changing a few ACLs. Take the Apollo/Domain OS as an example. All one needs to do to kill the system is to change a few ACLs and forget what they were supposed to be. Suddenly the system is crippled, nothing works. The only solution, if you don't have a backup, is to remove all of the security. Unix has a simpler security philosophy when it comes to the operating system files, but ACLs would be a valuable addition to the security of our data.

A cfengine bare-bones file-checking program looks like this:

```
#
# Free format cfengine program
```

```
#

 control:

   ActionSequence = ( files )

 files:

   classes::

     /directory/file  mode=644
                      owner=mark,ds
                      group=users,adm
                      acl=zap
                      action=fixplain

  # ... more below
```

This program simply checks the permissions and ownership of the named file. The regular file mode, owner and group are specified straightforwardly. The new feature here is the `acl` directive. It is a deceptively simply looking animal, but it hides a wealth of complexity. The `zap` is, of course, not an access control list. Rather, cfengine uses a system of aliases to refer to ACLs, so that the clutter of the complex ACL definitions does not impair the clarity of a file command. An ACL alias is defined in a separate part of the program which looks like this:

```
# ...contd


acl:

  { zap

  method:append
  fstype:solaris
  user:rmz:rwx
  user:len:r
  }
```

As you can see, an ACL is a compound object—a bundle of information which specifies which users have which permissions. Because ACLs are *lists* the alias objects must also know whether the items are to be appended to an existing list or whether they are to replace an existing list. Also, since the permission bits, general options and programming interfaces are all different for each type of filesystem, we have to tell cfengine what the filesystem type is.

It is possible to associate several ACL aliases with a file. When cfengine checks a files with ACLs, it reads the existing ACL and compares it to the new one. Files are only modified if they do not conform to the specification in the cfengine program. Let's look at a complete example:

```
files:
```

```
  $(HOME)/myfile acl=acl_alias1 action=fixall

acl:

  { acl_alias1

  method:append
  fstype:solaris
  user:len:rwx
  }
```

ACLs are viewed in Solaris with the command '`getfacl`'. Suppose that, before running this program, our test-file had permissions

```
user:*:rwx
user:mark:rwx           #effective:r-x
group:*:r-x              #effective:r-x
mask:r-x
other:r-x
default_user:rw-
default_group:r--
default_mask:-w-
default_other:rwx
```

After the cfengine run, the ACL would become:

```
user:*:rwx
user:mark:rwx           #effective:r-x
user:len:rwx            #effective:r-x
group:*:r-x              #effective:r-x
mask:r-x
other:r-x
default_user:rw-
default_group:r--
default_mask:-w-
default_other:rwx
```

Suppose we wanted to to remove 'w' bit for user '`jacobs`', or make sure that it was never there.

```
{ acl_alias1

method:append
fstype:solaris
user:jacobs:-w
}
```

Note that the method used here is append. That means that, whatever other access permissions we might have granted on this file, the user '`jacobs`' (a known cracker) will have no write permissions on the file. Had we used the method `overwrite` above, we would have eliminated all other access permissions for every user and added the above. If we really wanted to burn '`jacobs`', we could remove all rights to the file like this

```
user:jacobs:noaccess
```

The keyword `noaccess` removes all bits. Note that this is not necessarily the same as doing a `-rwx`, since some filesystems, like DFS, have more bits than this. Then, if we want to forgive and forget, the ACLs may be removed for `jacobs` with the syntax

```
user:jacobs:default
```

In Solaris, files inherit default ACLs from the directory they lie in; these are modified by the umask setting to generate their own default mask.

DFS ACLs look a little different. They are examined with the '`acl_edit`' command or with

```
dcecp -c acl show <filename>
```

In order to effect changes to the DFS, you have to perform a DCE login to obtain authentication cookies. The user '`cell_admin`' is a special user account for administrating a local DFS cell. Suppose we have a file with the following DCE ACL:

```
mask_obj:r-x---
user_obj:rwxcid
user:cell_admin:r--c-- #effective:r-----
group_obj:r-x--d       #effective:r-x---
other_obj:r-x---
```

Now we want to add '`wx`' permissions for user '`cell_admin`', and add new entries with '`rx`' permissons for group `acct-admin` and user '`root`'. This is done with the following ACL alias:

```
{ acl_alias2

method:append
fstype:dfs
user:/.../iu.hioslo.no/cell_admin:wx
group:/.../iu.hioslo.no/acct-admin:rx
user:/.../iu.hioslo.no/root:rx
user:*:-x
}
```

The local cell name '`/.../iu.hioslo.no`' is required here. Cfengine can not presently change ACLs in other cells remotely, but if your cfengine program covers all of the cell servers, then this is no limitation, since you can still centralize all your ACLs in one place. It is just that the execution and checking takes place at distributed locations. This is the beauty of cfengine. After running cfengine, with the above program snippet, the ACL then becomes:

```
mask_obj:r-x---
user_obj:rwcid
user:cell_admin:rwxc-- #effective:r-x---
user:root:r-x---       #effective:r-x---
group_obj:r-x--d       #effective:r-x---
group:acct-admin:r-x---
other_bj:r-x---
```

For the sake of simplicity we have only used standard Unix bits '`rwx`' here, but more complicated examples may be found in DFS. For example,

```
user:mark:+rwx,-cid
```

which sets the read, write, execute flags, but removes the control, insert and delete flags. In the DFS, files inherit the inital object ACL of their parent directory, while new directories inherit the initial container object.

The objects referred to in DFS as `user_obj`, `group_obj` and so forth refer to the owner of a file. i.e. they are equivalent to the same commands acting on the user who owns the file concerned. To make the cfengine user-interface less cryptic and more in tune with the POSIX form, we have dropped the '`_obj`' suffices. A user field of '`*`' is a simple abbreviation for the owner of the file.

A problem with any system of lists is that one can generate a sequence which does one thing, and then undoes it and redoes something else, all in the same contradictory list. To avoid this kind of accidental interaction, cfengine insists that each user has only one ACE (access control entry), i.e. that all the permissions for a given user be in one entry.

5 Using cfengine as a front end for `cron`

One of cfengine's strengths is its use of classes to identify systems from a single file or set of files. Many administrators think that it would be nice if the cron daemon also worked in this way. One possible way of setting up cron from a global configuration would be to use the cfengine `editfiles` facility to edit each cron file separately. A much better way is to use cfengine's time classes to work like a user interface for cron. This allows you to have a single, central cfengine file which contains all the cron jobs on your system without losing any of the fine control which cron affords you. All of the usual advantages apply:

- It is easier to keep track of what cron jobs are running on the system when you have everything in one place.
- You can use all of your carefully crafted groups and user-defined classes to identify which host should run which programs.

The central idea behind this scheme is to set up a regular cron job on every system which executes cfengine at frequent intervals. Each time cfengine is started, it evaluates time classes and executes the shell commands defined in its configuration file. In this way we use cfengine as a wrapper for the cron scripts, so that we can use cfengine's classes to control jobs for mulitple hosts. Cfengine's time classes are at least as powerful as `cron`'s time specification possibilities, so this does not restrict you in any way, See ⟨undefined⟩ [Building flexible time classes], page ⟨undefined⟩. The only price is the overhead of parsing the cfengine configuration file.

To be more concrete, imagine installing the following '`crontab`' file onto every host on your network:

```
#
# Global Cron file
#
0,15,30,45 * * * * /usr/local/cfengine/inputs/run-cfengine
```

This file contains just a single cron job, namely a script which calls cfengine. Here we are assuming that you will not want to execute any cron script more often than every fifteen minutes. If this is too restrictive, the above can be changed. We refer to the time interval between runs of the script '`run-cfengine`' as the 'scheduling interval' and discuss its implications in more detail below.

The script '`run-cfengine`' would replace any '`cfdaily`' or '`cfhourly`' scripts which you might have, and can as simple as this

```
#!/bin/sh
#
# Script run-cfengine

export CFINPUTS=/usr/local/cfengine/inputs

/usr/local/gnu/bin/cfengine

#
# Should we pipe mail to a special user?
#
```

or it could be more fancy. You could also use the '`cfwrap`' script, See ⟨undefined⟩ [cfwrap], page ⟨undefined⟩, if you have perl on all your systems, to pipe mail to the mail address described in the cfengine file, See ⟨undefined⟩ [sysadm], page ⟨undefined⟩.

```
#
# Global Cron file
#
0,15,30,45 * * * * path/cfwrap path/run-cfengine
```

You might not want to run your entire system configuration '`cfengine.conf`' every time cron fires up cfengine. An alternative would be to keep a separate fil for cron jobs called, say, '`cf.cron`'. You would then replace the '`run-cfengine`' file by

```
#!/bin/sh
#
# Script run-cfengine

export CFINPUTS=/usr/local/cfengine/inputs

/usr/local/gnu/bin/cfengine -f cf.cron

#
# Should we pipe mail to a special user?
#
```

There is no particular advantage to doing this unless you are running cfengine on some very slow hardware. A better way to approach the problem is to think of the '`cf.cron`' file as a module which can be imported into the main configuration file. This gives you the maximum amount of flexibilty, since it allows you to decide exactly what you want to happen any any given time from the central file.

5.1 Structuring '`cfengine.conf`'

The structure of '`cfengine.conf`' needs to reflect your policy for running jobs on the system. You need to switch on relevant tasks and switch off unwanted tasks depending on the time of day. This can be done in three ways:

- By placing individual actions under classes which restrict the times at which they are executed,

  action:

  ```
        Hr00.Min10_15||Hr12.Min45_55::

           Command
  ```

- By choosing a different `actionsequence` depending on the time of day.

  ```
  control:

        Hr00::   # Action-sequence for daily run at midnight

           actionsequence = ( sequence )

       !Hr00::   # Action-sequence otherwise

            actionsequence = ( sequence )
  ```

- By importing modules based on time classes.

  ```
  import:

        Hr00:: cf.dailyjobs

        any::  cf.hourlyjobs
  ```

The last of these is the most efficient of the three, since cfengine does not even have to spend time parsing the files for actions which you know you will not want.

5.2 Splaying host times

The trouble with starting every cfengine at the same time using a global cron file is that it might lead to contention or inefficiency. For instance, if a hundred cfengines all suddenly wanted to copy a file from a master source simultaneously this would lead to a big load on the server. We can prevent this from happening by introducing a time delay which is unique for each host and not longer than some given interval. Cfengine uses a hashing algorithm to generate a number between zero and a maximum value in minutes which you define, like this:

```
control:

   SplayTime = ( 60 ) # minutes
```

If this number is non-zero, cfengine goes to sleep after parsing its configuration file and reading the clock. Every machine will go to sleep for a different length of time, which is no longer than the time you specify in minutes. A hashing algorithm, based on the fully qualified name of the host, is used to compute a unique time for hosts. The shorter the interval, the more clustered the hosts will be. The longer the interval, the lighter the load on your servers. This 'splaying' of the run times will lighten the load on servers, even if they come from domains not under your control but have a similar cron policy.

Splaying can be switched off temporarily with the '`-q`' or '`--no-splay`' options.

5.3 Building flexible time classes

Each time cfengine is run, it reads the system clock and defines the following classes based on the time and date:

`Yr`*xx*`::` The current year, e.g. '`Yr1997`', '`Yr2001`'. This class is probably not useful very often, but it might help you to turn on the new-year lights, or shine up your systems for the new millenium!

Month`::` The current month can be used for defining very long term variations in the system configuration, e.g. '`January`', '`February`'. These classes could be used to determine when students have their summer vacation, for instance, in order to perform extra tidying, or to specially maintain some administrative policy for the duration of a conference.

Day`::` The day of the week may be used as a class, e.g. '`Monday`', '`Sunday`'.

`Day`*xx*`::` A day in the month (date) may be used to single out by date, e.g. the first day of each month defines '`Day1`', the 21st '`Day21`' etc.

`Hr`*xx*`::` An hour of the day, in 24-hour clock notation: '`Hr00`'...'`Hr23`'.

`Min`*xx*`::` The precise minute a which cfengine was started: '`Min00`' ... '`Min59`'. This is probably not useful alone, but these values may be combined to define arbitrary intervals of time.

`Min`*xx*`_`*xx*`::`
The five-minute interval in the hour at which cfengine was executed, in the form '`Min00_05`', '`Min05_10`' .. '`Min55_00`'.

Time classes based on the precise minute at which cfengine started are unlikely to be useful, since it is improbable that you will want to ask cron to run cfengine every single minute of every day: there would be no time for anything to complete before it was started again. Moreover, many things could conspire to delay the precise time at which cfengine were started. The real purpose in being able to detect the precise start time is to define composite classes which refer to arbitrary intervals of time. To do this, we use the `group` or `classes` action to create an alias for a group of time values. Here are some creative examples:

```
classes:  # synonym groups:

  LunchAndTeaBreaks = ( Hr12 Hr10 Hr15 )

  NightShift        = ( Hr22 Hr23 Hr00 Hr01 Hr02 Hr03 Hr04 Hr05 Hr06 )

  ConferenceDays    = ( Day26 Day27 Day29 Day30 )
```

```
QuarterHours      = ( Min00 Min15 Min30 Min45 )

TimeSlices        = ( Min01 Min02 Min03 Min33 Min34 Min35)
```

In these examples, the left hand sides of the assignments are effectively the ORed result of the right hand side. This if any classes in the parentheses are defined, the left hand side class will become defined. This provides an excellent and readable way of pinpointing intervals of time within a program, without having to use '|' and '.' operators everywhere.

5.4 Choosing a scheduling interval

How often should you call your global cron script? There are several things to think about:

- How much fine control do you need? Running cron jobs once each hour is usually enough for most tasks, but you might need to exercise finer control for a few special tasks.
- Are you going to run the entire cfengine configuration file or a special light-weight file?
- System latency. How long will it take to load, parse and run the cfengine script?

Cfengine has an intelligent locking and timeout policy which should be sufficient to handle hanging shell commands from previous crons so that no overlap can take place, See ⟨undefined⟩ [Spamming and security], page ⟨undefined⟩.

6 Cfengine and network services

This chapter describes how you can set up a cfengine network service to handle remote file distribution and remote execution of cfengine without having to open your hosts to possible attack using the `rsh` protocols.

6.1 Cfengine network services

By starting the daemon called `cfd`, you can set up a line of communication between hosts, allowing them to exchange files across the network or execute cfengine remotely on another system. Cfengine network services are built around the following components:

`cfengine` The configuration engine, whose only contact with the netork is via remote copy requests. This component does the hard work of configuring the system based on rules specified in the file '`cfengine.conf`'. It does not and cannot grant any access to a system from the network.

`cfd` A daemon which acts as both a file server and a remote-cfengine executor. This daemon authenticates requests from the network and processes them according to rules specified in '`cfd.conf`'. It works as a file server and as a mechanism for starting cfengine on a local host and piping its output back to the network connection.

`cfrun` This is a simple initiation program which can be used to run cfengine on a number of remote hosts. It cannot be used to tell cfengine what to do, it can only ask cfengine on the remote host to run the configuration file it already has. Anyone could be allowed to run this program, it does not require any special user privileges. A locking mechanism in cfengine prevents its abuse by spamming.

`cfwatch` This program (which is not a part of the distribution: it is left for others to implement) should provide a graphical user interface for watching over the configuration of hosts running cfengine and logging their output.

With these components you can emulate programs like `rdist` whose job it is to check and maintain copies of files on client machines. You may also decide who has permission to run cfengine and how often it may be run, without giving away any special user privileges.

6.2 How it works

6.2.1 Remote file distribution

This section describes how you can set up `cfd` as a remote file server which can result in the distrubution of files to client hosts in a more democratic way than with programs like rdist.

An important difference between cfengine and other systems has to do with the way files are distributed. Cfengine uses a 'pull' rather than a 'push' model for distributing network files. The `rdist` command, for instance, works by forcing an image of the files on one server machine onto all clients. Files get changed when the server wishes it and the clients have no choice but to live with the consequences. Cfengine cannot force its will onto other hosts in this way, it can only signal them and ask them to collect files if they want to. In other words, cfengine simulates a 'push' model by polling each client and running the local cfengine configuration script giving the host the chance to 'pull' any updated files from the remote server, but leaving it up to the client machine to decide whether or not it wants to update.

Also, in contrast to programs like `rdist` which distribute files over many hosts, cfengine does not require any general `root` access to a system using the '`.rhosts`' file or the '`/etc/hosts.equiv`' file. It is sufficient to run the daemon as root. You can not run it by adding it to the '`/etc/inetd.conf`' file on your system however. The restricted functionality of the daemon protects your system from attempts to execute general commands as the root user using `rsh`.

To remotely access files on a server, you add the keywork `server=`*host* to a copy command. Consider the following example which illustrates how you might distribute a password file from a masterhost to some clients.

```
copy:

  PasswdClients::

    /etc/passwd  dest=/etc/passwd owner=root group=0 server=server-host
```

Given that the `cfd` daemon is running on *server-host*, cfengine will make contact with the daemon and attempt to obtain information about the file. During this process, cfengine verifies that the system clocks of the two hosts are reasonably synchronized. If they are not, it will not permit remote copying. If cfengine determines that a file needs to be updated from a remote server it begins copying the remote file to a new file on the same filesystem as the destination-file. This file has the suffix '`.cfnew`'. Only when the file has been successfully collected will cfengine make a copy of the old file, See ⟨undefined⟩ [repository], page ⟨undefined⟩ and rename the new file into place. This behaviour is designed to avoid race-conditions which can occur during network connections and indeed any operations which take some time. If files were simply copied directly to their new destinations it is conceivable that a network error could interrupt the transfer leaving a corrupted file in place.

Cfengine places a timeout of a few seconds on network connections to avoid hanging processes.

Normally the daemon sleeps, waiting for connections from the network. Such a connection may be initiated by a request for remote files from a running cfengine program on another host, or it might be initiated by the program `cfrun` which simply asks the host running the daemon to run the cfengine program locally.

Make sure that you are running cfengine from a shell which has sensible limits set. The error 'too many open files' can occur in long recursions if you only have a small number of valid descriptors per shell. It is probably a good idea to set the number of descriptors to 1024.

6.2.2 Remote execution of cfengine

It is a good idea to execute cfengine by getting `cron` to run it regularly. This ensures that cfengine will be run even if you are unable to log onto a host to run it yourself. Sometimes however you will want to run cfengine immediately in order to implement a change in configuration as quickly as possible. It would then be inconvenient to have to log onto every host in order to do this manually. A better way would be to issue a simple command which contacted a remote host and ran cfengine, printing the output on your own screen:

```
myhost% cfrun remote-host -v

  output....
```

A simple user interface is provided to accomplish this. `cfrun` makes a connection to a remote cfd-daemon and executes cfengine on that system with the privileges of the cfd-daemon (usually `root`). This has a two advantages:

- You avoid having to log in on a remote host in order to reconfigure it.
- Users other than root can run cfengine to fix any problems with the system.

A potential disadvantage with such a system is that malicious users might be able to run cfengine on remote hosts. The fact that non-root users can execute cfengine is not a problem in itself, after all the most malicious thing they would be able to do would be to check the system configuration and repair any problems. No one can tell cfengine what to do using the cfrun program, it is only possible to run an existing configuration. But a more serious concern is that malicious users might try to run cfengine repeatedly (so-called 'spamming') so that a system became burdened with running cfengine constantly, See ⟨undefined⟩ [Spamming and security], page ⟨undefined⟩.

6.2.3 `cfrun`

The syntax of the `cfrun` command is

```
cfrun -option --longoption class1 class2 ...
```

With the exception of the '`-d`' and '`-S`' options, all options are passed on to the remote hosts and are ignored locally. The '`-q`' option is always assumed when executing cfengine remotely, so that SplayTime is effectively zero when polling hosts serially. If an option includes a name such as '`-Dnewclass`', there should not be a space between the option letter and the name string. The remaining options are treated as classes to be sent to all the hosts on the network.

Each host evaluates the classes sent by `cfrun` and decides whether cfengine should be invoked. Only hosts which belong to the classes defined on the `cfrun` command line are executed. This allows you to single out groups of hosts which should execute cfengine, based on the very classes which you have defined for your configuration. If no classes are sent on the command line, then all hosts are run.

`cfrun` uses a configuration file which is located under the `CFINPUTS` directory in order to determine which hosts and in which order it should try to connect. Because cfengine always uses a reliable TCP protocol for connections, it verifies each connection rather than simply broadcasting openly. Using this file you can even simulate broadcasting to hosts outside your subnet.

This file should contain every host name you ever want to configure remotely, because you can still select subsets of the file by specifying classes which the remote host will understand. If the remote host is not in one of the classes you specify when you run `cfrun`, then it will simply ignore the request. Conversely, if you do not place a host in this file, it will never be contacted when you use the `cfrun` command. The format of the file is as follows

```
#
# Comment ..
#
domain=my.domain

hostname1  options
hostname2  options
...
```

It is important to add the domain-name to this file. The options you specifiy in this file, per host, are added to those you might specify on the command line when invoking cfengine remotely. For instance, you might know of a bug on one host and decide not to perform interface configuration on that one machine. You would write a line like this:

```
funny.domain -i  # problem host
```

You could use `cfrun` inside one of your cfengine configuration files in order to remotely execute cfengine on all of the other network machines, by setting up a host list. Be careful not to include the name of the master host in the list. The locks should prevent cfengine from being run on the masterhost, avoiding an infinite loop. This way you do not have to rely on cron running on every system. The disadvantage however is that cfengine has to poll the systems on the network, which means that cfengine cannot be working in parallel on all hosts. This could be inefficient in the long run.

6.2.4 Spamming and security

The term 'spamming' refers to the senseless repetition of something in a malicious way intended to drive someone crazy[1]. In the computer world some malicious users, a bit like 'flashers' in the park[2] like to run around the net a reveal themselves ad nauseum by sending multiple mail messages or making network connections repeatedly to try to overload systems and people[3].

Whenever we open a system to the network, this problem becomes a concern. Cfengine is a tool for making peace with networked systems, not a tool to be manipulated into acts of senseless aggression. The cfengine daemon does make it possible for anyone to connect and run a cfengine process however, so clearly some protection is required from such attacks.

Cfengine's solution to this problem is a locking mechanism. Rather than providing user-based control, cfengine uses a time based locking mechanism which prevents actions from being executed unless a certain minimum time has elapsed since the last time they were executed. By using a lock which is not based on user identity, we protect several interests in one go:

---

[1] Recall the 'spam' song from Monty Python's flying circus?

[2] Recall the 'spam' song from Monty Python's flying circus?

[3] Recall the 'spam' song ... get the idea?

- Restricting cfengine access to root would prevent regular users, in trouble, from being able to fix problems when the system administrator was unavailable. A time-based lock does not prevent this kind of freedom.
- Accidents with cron or shell scripts could start cfengine more often than desirable. We also need to protect against such accidents.
- We can prevent malicious attacks regardless of whom they may come from.

Cfengine is controlled by a series of locks which prevent it from being run too often, and which prevent it from spending too long trying to do its job. The locks work in such a way that you can start several cfengine processes simultaneously without them crashing into each other. Coexisting cfengine processes are also prevented from trying to do the same thing at the same time (we call this 'spamming'). You can control two things about each kind of action in the action sequence:

- The minimum time which should have passed since the last time that action was executed. It will not be executed again until this amount of time has elapsed. (Default time is 0 minutes so that users un-aware of this feature will not be confused by it.)
- The maximum amount of time cfengine should wait for an old instantiation of cfengine 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*.

```
actionsequence
  (
  action.IfElapsedtime-in-mins
  action.ExpireAftertime-in-mins
  )
```

or globally,

```
control:

   IfElapsed   = ( time-in-mins )

   ExpireAfter = ( time-in-mins )
```

For example:

```
control:

  actionsequence =
    (
    files.IfElapsed240.ExpireAfter180
    copy
    tidy
```

```
    )

    IfElapsed = ( 30 )
```

In this example, we treat the files action differently to the others. For all the other actions, cfengine will only execute the files part of the program if 30 minutes have elapsed since it was last run. Since no value is set, the expiry time for actions is 60 minutes, which means that any cfengine process which is still trying to finish up after 60 minutes will be killed automatically by the next cfengine which gets started.

As for the files action: this will only be run if 240 minutes (4 hours) have elapsed since the last run. Similarly, it will not be killed while processing 'files' until after 180 minutes (3 hours) have passed.

These locks do not prevent the whole of cfengine from running, only so-called 'atoms'. Several different atoms can be run concurrently by different cfengines. Assuming that the time conditions set above allow you to start cfengine, the locks ensure that atoms will never be started by two cfengines at the same time, causing contention and wasting CPU cycles. Atoms are defined to maximize the security of your system and to be efficient. If cfengine were to lock each file it looked at seperately, it would use a large amount of time processing the locks, so it doesn't do that. Instead, it groups things together like this:

`copy, editfiles, shellcommands`
: Each separate command has its own lock. This means that several such actions can be processed concurrently by several cfengine processes. Multiple or recursive copies and edits are treated as a single object.

`netconfig, resolve, umount, mailcheck, addmounts, disable, processes`
: All commands of this action-type are locked simultaneously, since they can lead to contention.

`mountall, mountinfo, required, checktimezone`
: These are not locked at all.

Cfengine creates a directory '`~/.cfengine`' for writing lock files for ordinary users.

The option '`-K`' or '`--no-lock`' can be used to switch off the locking checks, but note that when running cfengine remotely via `cfd`, this is not possible.

6.2.5 Some points on the cfd protocol

Cfd uses a form for host-based authorization. Each atomic operation , such as statting, getting files, reading directories etc, requires a new connection and each connection is verified by a double reverse lookup in the server's DNS records. Single stat structures are cached during the processing of a file.

MD5 checksums are transferred from client to server to avoid loading the server. Even if a user could corrupt the MD5 checksum, he or she would have to get past access control with TCP wrappers and the worst that could happen would be to get the right version of the file. Again this is in keeping with the idea that users can only harm themselves and not others with cfengine.

6.2.6 Deadlocks and runaway loops

Whenever we allow concurrent processes to share a resource, we open ourselves up the possibilty of deadlock. This is a situation where two or more processes are locked in a vicious stalemate from which none can escape. Another problem is that it might be possible to start an infinite loop: cfengine starts itself.

Cfengine protects you from such loops to a large degree. It should not be possible to make such a loop by accident. The reason for this is the locking mechanism which prevents tasks being repeated too often. If you start a cfengine process which contains a shell-command to start cfengine again, this shell command will be locked, so it will not be possible to run it a second time. So while you might be able to start a second cfengine process, further processes will not be started and you will simply have wasted a little CPU time. When the first cfengine returns, the tasks which the second cfengine completed will not be repeated unless you have set the `IfElapsed` time or the `ExpireAfter` time to zero. In general, if you wish to avoid problems like this, you should not disable the locking mechanism by setting these two times to zero.

The possibility of deadlock arises in network connection. Cfengine will not attempt to use the network to copy a file which can be copied internally from some machine to itself. It will always replace the `server=` directive in a copy with 'localhost' to avoid unnecessary network connections. The prevents one kind of deadlock which could occur: namely cfrun executes cfengine on host A (cfd on host A is then blocked until this completes), but the host A configuration file contains a remote copy from itself to itself. This remote copy would then have to wait for cfd to unblock, but this would be impossible since cfd cannot unblock until it has the file. By avoiding remote copies to localhost, this possibility is avoided.

6.3 Configuring `cfd`

6.3.1 Installation of `cfd`

To install the cfengine daemon component, you will need to register a port for cfengine by adding the following line to the system file '`/etc/services file`'

```
cfengine        5308/tcp
```

You could do this for all hosts by adding the following to your cfengine configuration

```
editfiles:

  { /etc/services

   AppendIfNoSuchLine "cfengine        5308/tcp"
  }
```

To start cfengine at boot time, you need to place a line of the following type in your system startup files:

```
# Start cfengine server
cfd
```

Note that `cfd` will reread its configuration file whenever it detects that it has been changed, so you should not have to restart the daemon, not send it the HUP signal as with other daemons.

6.3.2 Configuration file 'cfd.conf'

The server daemon is controlled by a file called '`cfd.conf`'. The syntax of this configuration file is deliberately modelled on cfengine's own configuration file, but despite the similarities, you cannot mix the contents of the two files.

Though they are not compatible, '`cfengine.conf`' and '`cfd.conf`' are similar in several ways:

- Both files use classes to label entries, so that you may use the same configuration file to control all hosts on your network. This is a convenience which saves you the trouble of maintaining many different files.
- Both files are searched for using the contents of the variable `CFINPUTS`.
- You can use `groups` and `import` in both files to break up files into convenient modules and to import common resources, such as lists of groups.

Note that the classes in the '`cfd.conf`' file do not tell you the classes of host which have access to files and directories, but rather which classes of host pay attention to the access and deny commands when the file is parsed.

Host name authentication is not by class or group but by hostname, like the '`/etc/exports`' file on most unix systems. The syntax for the file is as follows:

```
control:

  classes::

        domain = ( DNS-domain-name )

        cfrunCommand = ( "script/filename" )  # Quoted

        MaxConnections = ( maximum number of forked daemons )

        ChecksumDatabase = ( filename )

        IfElapsed = ( time-in-minutes )

groups:

  Group definitions

import:

  Files to import

admit: | grant:

  classes::

       /file-or-directory

         wildcards/hostnames

deny:

  classes::

       /file-or-directory

         wildcards/hostnames
```

The file consists of a control section and access information. You may use the control section to define any variables which you want to use in the remainder of your file. Two variables are special here, they are reserved.

`cfrunCommand`
: This string is the command which you would like to be executed remotely by the `cfrun` command.

`MaxConnections`
: This integer value sets a limit on the maximum number of child daemon processes which cfengine will fork in order to handle remote requests. The default value is ten daemons.

`IfElapsed`
: The `IfElapsed` anti-spamming filter is also built into `cfd` so that a remote user cannot even get as far as causing cfengine to parse its input files (which could be used for spamming in itself). The time is in minutes, the default is one hour.

`ChecksumDatabase`
: This is the path and filename to a database which will cache MD5 checksum values server-side. This optimization is only available if you have the Berkeley database library '`libdb`' on your system. If this variable is not defined, no database caching will be used and checksum values will be computed directly on request. The utility of this solution is a trade-off between the time it takes to compute the checksum versus the time for a disk-based lookup.

Following the control section comes a list of files or directories and hosts which may access these. If permissions are granted to a directory then all sub directories are automatically granted also. Note that symbolic links are not checked for, so you may need to specifically deny access to links if they are plain files, but cfd does not follow symbolic links and give access to files in other directories.

Fully qualified hostnames should be given in this file. Do not forget to define the domain name. Authentication calls the unix function `gethostbyname()` and so on to identify and verify connecting hosts, so the names in the file must reflect the type on names returned by this function. You may use wildcards in names to match, for instance, all hosts from a particular domain.

Here is an example file

```
#####################################################
#
# This is a cfd config file
#
#####################################################

groups:

  PasswdHost = ( nexus )

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

control:

  #
  # Assuming CFINPUTS is defined
  #

  cfrunCommand = ( "/usr/local/bin/cfengine" )

  variable = ( /usr/local/publicfiles )

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

admit:   # Can also call this grant:
```

```
        PasswdHost::

          /etc/passwd

             *.iu.hioslo.no

         FtpHost::

         # An alternative to ftp, grant anyone

            /local/ftp/pub

              *

         any::

            $CFINPUTS/cfrun.sh

              *.iu.hioslo.no

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

deny:

   /etc/services

       borg.iu.hioslo.no

  /local/ftp

       *.pain-in-the-ass.com
```

NOTE I: cfd is not `rpc.mountd`*, access control is by filename, not by device name. Do not assume that files lying in subdirectories are not open for access simply because they lie on a different device. You should give the real path name to file and avoid symbolic links.*

NOTE II: access control is per host, not per user. If you open a file for a host you open it for every user on that host.

If you still have problems with lack of access, it could be that you have forgotten to define the domain name for your network, or that you do not understand the TCP wrappers files '`/etc/hosts.access`' and '`/etc/hosts.deny`'.

6.3.3 TCP wrappers

Cfengine tries to incorporate the TCP wrappers package if you have it on your system. If you do, then the files '`/etc/hosts.allow`' and '`/etc/hosts.deny`' allow you to give the cfengine/cfd service an extra level of protection from 'clever' spoofing attempts.

7 Security and cfengine

7.1 Security hints

Cfengine is not specifically a tool for implementing high security solutions for system administration, but it has many features which can be used to monitor the state of your systems and warn about potential breaches in security. Here are some suggestions as to how you can be more security conscious with cfengine's help.

CERT advisories
: The CERT coordination centre (Computer Emergency Response Team) publishes warnings about known bugs and security risks in computer systems which can lead to compromised security. Their recommendations often involve disabling certain programs, changing permissions to remove setuid root flags and editing configuration files. These are things which you can deal with using cfengine.

disabling binaries
: When to elect to disable a file, cfengine renames it, moves it to a file repository (if you have defined one) and changes the mode of the file to read only for its owner. This is sufficient to disable binary programs and plain files.

The setuid log
: Cfengine is always on the lookout for files which are setuid or setgid root. It doesn't go actively looking for them, but whenever you get cfengine to check a file or directory with the `files` feature, it will make a note of setuid programs it finds there. These are recorded in the file '`cfengine.`*host*`.log`' which is stored under '`/etc/cfengine`' or '`/var/log/cfengine`'. When new setuid programs are discovered, a warning is printed, but only if you are *root*. If you ever want a complete list, delete the log file and cfengine will think that all of the setuid programs it finds are new. The log file is not readable by normal users.

Suspicious filenames
: Whenever cfengine opens a directory and scans through files (`files`, `tidy`, `copy`), it is on the lookout for for suspicious filenames, i.e. files like '`.. .`' containing only space and/or dots. Such files are never created by sensible people, but are often used by hackers to try to hide dangerous programs. Cfengine prints warnings about such files.

Spoofing
: Spoofing refers to attempts to masquerade as another host when sending network transmissions. The `cfd` program attempts to unmask such attempts by performing double reverse lookups in the name service. This verifies by a trusted server that the socket address and the host name are really who they claim to be. If you have the TCP wrappers package on your system (libwrap) then cfd will attempt to use it to detect other spoofs too, See ⟨undefined⟩ [TCP wrappers], page ⟨undefined⟩. If you don't have TCP wrappers, then the only line of defense is the double reverse lookup.

Race conditions in file copying
: When copying files from a source, it is possible that something might go wrong during the operation and leave a corrupt file in place. For example, the disk might become full while copying a file. This could lead to problems. Cfengine deals with this by always

copying to a new file on the destination filesystem (prefix '`.cfnew`') and then renaming it into place, only if the transfer was successful. This ensures that there is space on the filesystem and that nothing went wrong with the network connection or the disk during copying.

`size=` *in copy*
: As a further check on copying, cfengine allows you to define acceptable limits on the size of files. After all, sometimes errors might occur quite independently of anything you are doing with cfengine. Perhaps the master password file got emptied somehow, or got replaced by a binary, through some silly mistake. By checking making an estimate of the expected size of the file and adding it to the copy command, you can avoid installing a corrupt file and making a localized problem into a global one.

`useshell=` *in shellcommands*
: There are dangers in starting scripts from programs which run with root privileges. Normally, shell commands are started by executing them with the help of a '`/bin/sh -c`' command. The trouble with this is that it leaves one open to a variety of attacks. One example is fooling the shell into starting foreign programs by manipulating the `IFS` variable to treat '/' sa a separator. You can ask cfengine to start programs directly, without involving an intermediary shell, by setting the `useshell` variable to false. The disadvantage is that you will not be able to use shell directives such as '`|`' and `>` in your commands.

7.2 Who do you trust?

All the developments of the last few years point to the unpleasant fact that we need to be extra security conscious on the net. In order to have any meaningful discussion about security, you need to determine who you trust and who you don't trust. No one from outside your network can force cfengine to do anything you don't want it to do (unless root access to your system has been compromised by another route), but you might decide to collect a file from a remote server which could sabotage your system a treat. Cfengine does not implement more exacting security than normal host validation. If you are collecting files from remote servers, you should make sure that they come from a machine that you trust, particularly if they are files which could lead to privileged access to your system. Cfengine places the responsibility on you. You can make cfengine destroy your system, but no one else can, so make sure you think about what you are doing.

For example, it would be an extremely foolish idea to copy a binary program such as '`/bin/ps`' from a host you know nothing about. This program runs as root. If someone were to replace that version of '`ps`' with a trojan horse command, you would have effectively opened your system to attack.

Cfengine performs no cryptographic coding of messages at present, so if you are sending sensitive data via cfengine, it should be coded in advance.

7.3 Firewalls

Cfengine is a useful tool for implementing, monitoring and maintaining firewalls. You can control what programs are supposed to be on the firewall and what programs are not supposed to be there. You can control file permissions, processes and a dozen other things which make up the configuration of a bastion host. At some point in the future this space might expand into a discussion about how you set up a bastion host using cfengine.

PART II

Reference section

8 Command reference

In this section you will find each facet of a cfengine program listed together with an appropriate explanation. The commands are presented in alphabetical order for ease of lookup. Use this section in conjunction with the example program See ⟨undefined⟩ [Example configuration file], page ⟨undefined⟩.

8.1 acl

```
acl:

   class::

      { acl-alias

      action
      }
```

Cfengine's `ACL` feature is a common interface for managing filesystem access control lists (ACLs). An access control list is an extended file permission. It allows you to open or close a file to a named list of users (without having to create a group for those users); similarly, it allows you to open or close a file for a list of groups. Several operating systems have access control lists, but each typically has a different syntax and different user interface to this facility, making it very awkward to use. This part of a cfengine configuration simplifies the management of ACLs by providing a more convenient user interface for controlling them and—as far as possible—a common syntax.

An ACL may, by its very nature, contain a lot of information. Normally you would set ACLs in a `files` command, See ⟨undefined⟩ [files], page ⟨undefined⟩, or a `copy` command, See ⟨undefined⟩ [copy], page ⟨undefined⟩. It would be too cumbersome to repeat all of the information in every command in your configuration, so cfengine simplifies this by first associating an alias together with a complex list of ACL information. This alias is then used to represent the whole bundle of ACL entries in a `files` or `copy` command. The form of an ACL is similar to the form of an `editfiles` command. It is a bundle of information concerning a file's permissions.

```
{ acl-alias

 method:overwrite/append
 fstype:posix/solaris/dfs/afs/hpux/nt

 acl_type:user/group:permissions
 acl_type:user/group:permissions
 ...
}
```

The name *acl-alias* can be any identifier containing alphanumeric characters and underscores. This is what you will use to refer to the ACL entries in practice. The method entry tells cfengine how to interpret the entries: should a file's ACLs be overwritten or only adjusted? Since the filesystems from different developers all use different models for ACLs, you must also tell cfengine what kind of filesystem the file resides on. Currently only solaris and DCE/DFS ACLs are implemented.

8.1.1 Access control entries

An access control list is build of any number of individual access control entries (ACEs). The ACEs has the following general syntax:

acl_type:*user/group*:*permissions*

The user or group is sometimes referred to as a *key*.

For an explanation of ACL types and their use, refer to your local manual page. However, note that for each type of filesystem, there are certain entries which must exist in an ACL. If you are creating a new ACL from scratch, you must specify these. For example, in solaris ACLs you must have entries for `user`, `group` and `other`. Under DFS you need what DFS calls a `user_obj`, `group_obj` and an `other_obj`, and in some cases `mask_obj`. In cfengine syntax these are called `user:*:`, `other:*:` and `mask:*:`, as described below. If you are appending to an existing entry, you do not have to re-specify these unless you want to change them.

Cfengine can overwrite (replace) or append to one or more ACL entries.

`overwrite`
: `method:overwrite` is the default. This sets the ACL according to the specified entries which follow. The existing ACL will be overwritten completely.

`append`
: `method:append` adds or modifies one or more specified ACL entries. If an entry already exists for the specified type and user/group, the specified permission bits will be added to the old permissions. If there is no ACL entry for the given type and user/group, a new entry will be appended.

If the new ACL exactly matches the existing ACL, the ACL is not replaced.

The individual bits in an ACE may be either added subtracted or set equal to a specified mask. The '`+`' symbol means add, the '`-`' symbol subtract and '`=`' means set equal to. Here are some examples:

```
acltype:id/*:mask

user:mark:+rx,-w
user:ds:=r
user:jacobs:noaccess
user:forgiven:default

user:*:rw
group:*:r
other:*:r
```

The keyword `noaccess` means set all access bits to zero for that user, i.e. remove all permissions. The keyword `default` means remove the named user from the access crontrol list altogether, so that the default permissions apply. A star/asterisk in the centre field indicates that the user or group ID is implicitly specified as of the owner of the file, or that no ID is applicable at all (as is the case for 'other').

8.1.2 Solaris ACLs

Under Solaris, the ACL type can be one of the following:

```
user
group
mask
other
default_user
default_group
default_mask
default_other
```

A user or group can be specified to the user, group, default_user and default_group types. Solaris ACL permissions are the normal UNIX permissions bits '`rwx`', where:

r - *Grants read privileges.*
w - *Grants write privileges.*
x - *Grants execute privileges.*

8.1.3 DFS ACLs

In DCE, the ACL type can be one of the following:

```
other
mask
any
unauthenticated
user
group
foreign_other
foreign_user
foreign_group
```

The `user`, `group`, `foreign_user` and `foreign_group` types require that you specify a user or group. The DCE documentation refers to types `user_obj`, `group_obj` and so on. In the cfengine implementation, the ugly '`_obj`' suffix has been dropped to make these more in keeping with the POSIX names. `user_obj::`, is equivalent to `user:*:` is cfengine. The star/asterisk implies that the ACL applies to the owner of the file object.

DFS permissions are comprised of the bits '`crwxid`', where:

c - *Grants control privileges, to modify an acl.*
r - *Grants read privileges.*
w - *Grants write privileges.*
x - *Grants execute privileges.*
i - *Grants insert privileges.*

d - *Grants delete privileges.*

See the DCE/DFS documentation for more information about this.

It is not possible to set ACLs in foreign cells currently using cfengine, but you can still have all of your ACL definitions in the same file. You must however arrange for the file to be executed on the server for the cell concerned. Note also that you must perform a DCE login (normally as user '`cell_admin`') in order to set ACLs on files which are not owned by the owner of the cfengine-process. This is because you must have a valid security ticket.

8.1.4 ACL Example

Here is an example of a configuration file for one Solaris ACL and one DCE/DFS ACL:

```
control:
        actionsequence = ( files )
        domain = ( iu.hioslo.no )

files:
        $(HOME)/tt    acl=acl_alias1    action=fixall
        /:/bigfile    acl=acl_alias2    action=fixall

acl:
        { acl_alias1

        method:overwrite
        fstype:posix

        user:*:rwx
        user:mark:=rwx
        user:sowille:=rx
        user:toreo:=rx
        user:torej:default
        user:ds2:+rwx
        group:*:rx
        group:iu:r
        group:root:x
        mask:*:rx
        other:*:rx

        default_user:*:=rw
        default_user:mark:+rwx
        default_user:ds:=rwx
        default_group::=r
        default_group:iu:+r
        default_mask::w
        default_other::rwx
        }

        { acl_alias2
```

```
method:overwrite
fstype:dfs

user:*:rwxcid
group:*:rxd
other:*:wxir
mask:*:rxw
user:/.../iu.hioslo.no/cell_admin:rc
group:/.../iu.hioslo.no/acct-admin:rwxcid
user:/.../iu.hioslo.no/root:rx
}
```

8.2 binservers

The `binservers` declaration need only be used if you are using cfengine's model for mounting NFS filesystems. This declaration informs hosts of which other hosts on the network possess filesystems containing software (binary files) which client hosts should mount. This includes resources like programs in `/usr/local` and so on. A host may have several binary servers, since there may be several machines to which disks are physically attached. In most cases, on a well organized network, there will be only one *architecture server* per UNIX platform type, for instance a SunOS server, an ULTRIX server and so on.

Binary servers are defined as follows:

```
binservers:

   physics.sun4::   sunserver sunserver2
   physics.linux::  linuxserver
```

The meaning of this declaration is the following. All hosts of type `sun4` which are members of the group `physics` should mount any binaries declared in the `mountables` resource list which belong to hosts `sunserver` or `sunserver2`. Similarly all `linux` machines should mount binary filesystems in the mountables list from `linuxserver`.

Cfengine knows the difference between binaries and home directories in the `mountables` list, because home directories match the pattern given by `homepattern`. See ⟨undefined⟩ [homepattern], page ⟨undefined⟩. See ⟨undefined⟩ [homeservers], page ⟨undefined⟩.

Note that every host is a binary server for itself, so that the first binary server (and that with highest priority) is always the current host. This ensures that local filesystems are always used in preference to NFS mounted filesystems. This is only relevant in connection with the variable `$(binserver)`.

8.3 broadcast

This information is used to configure the network interface for each host.

Every local area network has a convention for determining which internet address is used for broadcast requests. Normally this is an address of the form `aaa.bbb.ccc.255` or `aaa.bbb.ccc.0`. The difference between these two forms is whether all of the bits in the last number are ones or zeroes respectively. You must find out which convention is used at your establishment and tell cfengine using a declaration of the form:

```
broadcast:

  any::

     ones     # or zeros, or zeroes
```

In most cases you can use the generic class `any`, since all of the hosts on the same subnet have to use the same convention. If your configuration file encompasses several different subnets with different conventions then you will need to use a more specific.

Cfengine computes the actual value of the broadcast address using the value specified above and the netmask See ⟨undefined⟩ [netmask], page ⟨undefined⟩.

8.4 control

The fundamental piece of any cfengine script or configuration file is the control section. If you omit this part of a cfengine script, it will not do anything! The control section is used to define certain variables, set default values and define the order in which the various actions you have defined will be carried out. Because cfengine is a declarative or descriptive language, the order in which actions appear in the file does not necessarily reflect the order in which they are executed. The syntax of declarations here is:

```
control:

   classes::

      variable = ( list or value )
```

The control section is a sequence of declarations which looks something like the following example:

```
control:

  site     = ( univ )
  domain   = ( univ.edu )
  sysadm   = ( admin@computing.univ.edu )
  netmask  = ( 255.255.252.0 )
  timezone = ( EDT )
  nfstype  = ( nfs )

  sensiblesize  = ( 1000 )
  sensiblecount = ( 2 )
  editfilesize  = ( 4000 )

  actionsequence =
     (
     links.some
     mountall
     links.others
     files
     )

  myvariable = ( something )
  mymacro    = ( somethingelse )
```

Parentheses are required when making a declaring information in cfengine.

The meaning of each of these lines is described below.

8.4.1 access

The `access` list is a list of users who are to be allowed to execute a cfengine program. If the list does not exist then all users are allowed to run a program.

```
access = ( user1 user2 ... )
```

The list may consist of either numerical user identifiers or valid usernames from the password database. For example:

```
access = ( mark aurora 22 456 )
```

would restrict a script to users mark, aurora and user id 22 and 456.

8.4.2 actionsequence

The action sequence determines the order in which collective actions are carried out. Here is an example containing the full list of possibilities:

```
actionsequence =
   (
   mountall              # mount filesystems in fstab
   mountinfo             # scan mounted filesystems
   checktimezone         # check timezone
   netconfig             # check net interface config
   resolve               # check resolver setup
   unmount               # unmount any filesystems
   shellcommands         # execute shell commands
   editfiles             # edit files
   addmounts             # add new filesystems to system
   directories           # make any directories
   links                 # check and maintain links (single and child)
   simplelinks           # check only single links (separate from childlinks)
   childlinks            # check only childlinks (separate from singlelinks)
   mailcheck             # check mailserver
   mountall              # (again)
   required              # check required filesystems
   tidy                  # tidy files
   disable               # disable files
   files                 # check file permissions
   copy                  # make a copy/image of a master file
   processes             # signal / check processes
   )
```

Here is a more complete description of the meaning of these keywords.

`addmounts`

causes cfengine to compute which NFS filesystems are missing from the current host and add them. This includes editing the filesystem table, creating the mount-directory, if required. This command relies on information provided by `mountinfo`, so it should normally only be called after `mountinfo`. If the filesystem already appears to be in the filesystem table, a warning is issued.

`checktimezone`
: runs a check on the timezone defined for the shell running cfengine.

`directories`
: executes all the commands defined under the `directories` section of the program. It builds new directories.

`disable`
: executes all the commands defined under the `disable` section of the program.

`editfiles`
: executes all the commands defined under the `editfiles` section of the program.

`files`
: executes all the commands defined under the `files` section of the program.

`links`
: executes all the commands defined under the `links` section of the program. Here one can also write `singlelinks` which checks only single (not multiply linked) objects, or `childlinks` which checks the remainder (multiply linked) objects. In this way one can separate these two actions if required, though normally this is not necessary.

`mailcheck`
: tests for the presence of the NFS-mounted mail spooling directory on the current host. The name of the mail spool directory is defined in the `mailserver` section of the cfengine program. If the current host is the same as the mailserver (the host which has the physical spool directory disk) nothing is done. Otherwise the filesystem table is edited so as to include the mail directory.

`mountall`
: mounts all filesystems defined in the hosts filesystem table. This causes new NFS filesystems added by `addmounts` and `mailcheck` to be actually mounted. This should probably be called both before `mountinfo` and after `addmounts` etc. A short timeout is placed on this operation to avoid hanging RPC connections when parsing NFS mounted file systems.

`mountinfo`
: builds internal information about which filesystems are presently mounted on the current host. Cfengine assumes that required-filesystems which are not found need to be mounted. A short timeout is placed on this operation to avoid hanging RPC connections when parsing NFS mounted file systems. If this times out, no further mount operations are considered reliable and are summarily cancelled.

`netconfig`
: checks the netmask, hostname, IP address and broadcast address for the current host. The correct values for the netmask and broadcast address are set if there is an error. The defaultroute is also added to the static routing table.

`required`
: executes all the commands defined under the `required` section of the program. It checks for the absence of important NFS resources.

`resolve`
: checks and corrects the DNS domain name and the order of nameservers in the file '`/etc/resolv.conf`'.

`shellcommands`
: executes all the commands defined under the `shellcommands` section of the program.

`tidy`
: executes all the commands defined under the `tidy` section of the program.

`unmount` executes all the commands defined under the `unmount` section of the program. The filesystem table is edited so as to remove the unwanted filesystems and the unmount operation is executed.

`processes`
executes commands defined under the `processes` section of the program.

Under normal circumstances this coarse ordering is enough to suit most purposes. In some cases you might want to, say, only perform half the link operations before mounting filesystems and then, say, perform the remainder. You can do this (and similar things) by using the idea of defining and undefining classes See ⟨undefined⟩ [Defining classes], page ⟨undefined⟩.

The syntax

```
actionsequence =
   (
   links.firstpass.include
   ...
   links.secondpass
   )
```

means that cfengine first executes `links` with the classes `firstpass` and `include` *defined*. Later it executes `links` with `secondpass` defined. You can use this method of adding classes to distinguish more finely the flow of control in programs.

A note about style: if you define and undefine lots of classes to do what you want to do, you might stop and ask yourself if your `groups` are defined as well as they should be. See ⟨undefined⟩ [groups], page ⟨undefined⟩. Programming in cfengine is about doing a lot for only a little writing. If you find yourself writing a lot, you are probably not going about things in the right way.

8.4.3 AddClasses

`AddClasses  = (` *list of identifiers* `)`

The `AddClasses` directive is used to define a list of class attributes for the current host. Normally only the hard classes defined by the system are 'true' for a given host. It is convenient though to be able to define classes of your own to label certain actions, mainly so that they can later be excluded so as to cut short or filter out certain actions. This can be done in two ways. See ⟨undefined⟩ [actionsequence], page ⟨undefined⟩.

To define a list of classes for the current session, you write:

```
AddClasses = ( exclude shortversion )
```

This is equivalent to (though more permanent than) defining classes on the command line with the `-D` option. You can now use these to qualify actions. For example

```
 any.exclude::
     ...
```

Under normal circumstances `exclude` is always true — because you have defined it to be so, but you can *undefine* it in two ways so as to prevent the action from being carried out. One way is to undefine a class on the command line when you invoke cfengine:

```
host#  cfengine -N exclude
```

or

```
host#  cfengine -N exclude.shortversion

host#  cfengine -N a.b.c.d
```

These commands run cfengine with the named classes *undefined*. That means that actions labelled with these classes are excluded during that run.

Another way to restrict classes is to add a list of classes to be undefined in the actionsequence. See next section.

8.4.4 CopyLinks

This list is used to define a global list of names or patterns which are to be copied rather than linked symbolically. For example

```
CopyLinks = ( *.config )
```

The same facility can be specified for each individual link operation using the `copy` option See ⟨undefined⟩ [links], page ⟨undefined⟩. Copying is performed using a file age comparison.

Note that all entries defined under a specified class are valid only as long as that class is defined. For instance

class::

`CopyLinks = (` *pattern* `)`

would define a pattern which was only valid when *class* is defined.

8.4.5 domain

`domain = (` *domain name* `)`

This variable defines the domainname for your site. You must define it here, because your system might not know its domainname when you run cfengine for the first time. The domainname can be used as a cfengine variable subsequently by referring to $(domain). The domainname variable is used by the action `resolve`.

8.4.6 DryRun

`DryRun = (` *on*/**off** `)`

This variable has the same effect as the command line options `--dry-run` or `-n`. It tells cfengine to only report what it should do without actually doing it.

classes::

```
DryRun = ( on )
```

8.4.7 editfilesize

```
EditfileSize  = ( size )
```

This variable is used by cfengine every time it becomes necessary to edit a file. Since file editing applies only to text files, the files are probably going to be relatively small in most cases. Asking to edit a very large (perhaps binary) file could therefore be the result of an error.

A check is therefore made as a security feature. Cfengine will refuse to edit a file which is larger than the value of `editfilesize` in bytes. This is to prevent possible accidents from occurring. The default value for this variable is 1000 bytes. If you don't like this feature, simply set the value to be a very large number or to zero. If the value is zero, cfengine will ignore it.

8.4.8 ExcludeCopy

This list is used to define a global list of names or patterns which are to be excluded from copy operations. For example

```
ExcludeCopy = ( *~ *% core )
```

The same facility can be specified for each individual link operation using the `exclude` option See ⟨undefined⟩ [copy], page ⟨undefined⟩.

Note that all entries defined under a specified class are valid only as long as that class is defined. For instance

```
class::

    ExcludeCopy = ( pattern )
```

would define a pattern which was only valid when *class* is defined.

8.4.9 ExcludeLinks

This list is used to define a global list of names or patterns which are to be excluded from linking operations. For example

```
ExcludeLinks = ( *~ *% core )
```

The same facility can be specified for each individual link operation using the `exclude` option See ⟨undefined⟩ [links], page ⟨undefined⟩.

Note that all entries defined under a specified class are valid only as long as that class is defined. For instance

```
class::

    ExcludeLinks = ( pattern )
```

would define a pattern which was only valid when *class* is defined.

8.4.10 ExpireAfter

This parameter controls the global value of the ExpireAfter parameter, See ⟨undefined⟩ [Spamming and security], page ⟨undefined⟩. This parameter controls the maximum time in minutes which a cfengine action is allowed to live. After this time cfengine will try to kill the cfengine which seems to have hung and attempt to restart the action.

```
ExpireAfter = ( time-in-minutes )
```

This parameter may also be set per action in the action sequence by appending a pseudo-class called `ExpireAfter`*time*. For instance,

```
actionsequence = ( copy.ExpireAfter15 )
```

sets the expiry time parameter to 15 minutes for this copy command.

8.4.11 HomePattern

```
HomePattern  = ( list of wildcards )
```

The `homepattern` variable is used by the cfengine model for mounting nfs filesystems. See ⟨undefined⟩ [NFS resources], page ⟨undefined⟩. It is also used in the evaluation of the pseudo variable `home`, See ⟨undefined⟩ [files], page ⟨undefined⟩, ⟨undefined⟩ [tidy], page ⟨undefined⟩.

`homepattern` is in fact a list and is used like a wildcard or *pattern* to determine which filesystems in the list of mountables are home directories. See ⟨undefined⟩ [mountables], page ⟨undefined⟩. This relies on your sticking to a rigid naming convention as described in the first reference above.

For example, you might wish to mount (or locate directly if you are not using a separate partition for home directories) your home directories under `mountpattern` in directories `u1`, `u2` and so on. In this case you would define `homepattern` to match these numbers:

```
homepattern = ( u? )
```

Cfengine now regards any directory matching `$(mountpattern)/u?` as being a user login directory.

Here is another example in which you split up a single partition into subdirectories. Suppose you want to create mount home directories under `$(mountpattern)/home` and make subdirectories for staff and students. Then you would write:

```
HomePattern = ( home/staff home/students )
```

Or you could combine the two:

```
HomePattern = ( u?/staff u?/students )
```

8.4.12 IfElapsed

This parameter controls the global value of the IfElapsed parameter, See ⟨undefined⟩ [Spamming and security], page ⟨undefined⟩. This parameter controls the minimum time which must have elapsed for an action in the action sequence before which it will be executed again.

```
IfElapsed = ( time-in-minutes )
```

This parameter may also be set per action in the action sequence by appending a pseudo-class called `IfElapsed`*time*. For instance,

```
ActionSequence = ( copy.IfElapsed15 )
```

sets the elapsed time parameter to 15 minutes for this copy command.

8.4.13 Inform

```
Inform = ( on/off )
```

This variable switches on the output level whereby cfengine reports changes it makes during a run. Normally only urgent messages or clear errors are printed. Setting `Inform` to `on` makes cfengine report on all actions not explicitly cancelled with a 'silent' option. To set this output level one writes:

```
classes::

 Inform = ( on )
```

8.4.14 InterfaceName

If you have an operating system which is installed on some non-standard hardware, you might have to specifically set the name of the network interface. For example:

```
control:

  nextstep.some::

     InterfaceName = ( en0 )

  nextstep.others::

     InterfaceName = ( ec0 )
```

It is only necessary to set the interface name in this fashion if you have an operating system which is running on special hardware. Most users will not need this. The choice set here overrides the system defaults and the choices made in the '`cfrc`' file, See ⟨undefined⟩ [cfrc resource file], page ⟨undefined⟩.

8.4.15 LinkCopies

This list is used to define a global list of names or patterns which are to be linked symbolically rather than copied. For example

```
excludelinks = ( *.gif *.jpg )
```

The same facility can be specified for each individual link operation using the `symlink` option See ⟨undefined⟩ [copy], page ⟨undefined⟩.

Note that all entries defined under a specified class are valid only as long as that class is defined. For instance

```
class::

    LinkCopies = ( pattern )
```

would define a pattern which was only valid when *class* is defined.

8.4.16 mountpattern

```
mountpattern  = ( mount-point )
```

The `mountpattern` list is used by the cfengine model for mounting nfs filesystems. See ⟨undefined⟩ [NFS resources], page ⟨undefined⟩. It is also used in the evaluation of the pseudo variable `home`, See ⟨undefined⟩ [files], page ⟨undefined⟩, ⟨undefined⟩ [tidy], page ⟨undefined⟩.

It is used together with the value of `homepattern` to locate and identify what filesystems are local to a given host and which are mounted over the network. For this list to make sense you need to stick to a rigid convention for mounting your filesystems under a single naming scheme as described in the section mentioned above. If you follow the recommended naming scheme then you will want to set the value of mountpattern to

```
mountpattern = ( /$(site)/$(host) )
```

which implies that cfengine will look for local disk partitions under a unique directory given by the name of the host and site. Any filesystems which are physically located on the current host lie in this directory. All mounted filesystems should lie elsewhere. If you insist on keeping mounted file systems in more than one location, you can make a list like this:

```
mountpattern = ( /$(site)/users /$(site)/projects )
```

8.4.17 netmask

```
netmask = ( aaa.bbb.ccc.ddd )
```

The netmask variable defines the partitioning of the subnet addresses on your network. Its value is defined by your network administrator. On most systems it is likely to be `255.255.255.0`. This is used to configure the network interface in `netconfig`. See ⟨undefined⟩ [actionsequence], page ⟨undefined⟩.

Every host on the internet has its own unique address. The addresses are assigned hierarchically. Each network gets a *domain name* and can attach something like 65,000 hosts to that network. Since this is usually too many to handle in one go, every such network may be divided up into subnets. The administrator of the network can decide how the division into subnets is made. The decision is a trade-off between having many subnets with few hosts, or many hosts on few subnets. This choice is made by setting the value of a variable called `netmask`. The netmask looks like an internet address. It takes the form:

```
aaa.bbb.ccc.mmm
```

The first two numbers 'aaa.bbb' are the address of the domain. The remainder 'ccc.mmm' specifies both the subnet and the hostname. The value of netmask tells all hosts on the network: how many of the bits in the second half label different subnets and how many label different hosts on each of the subnets?

The most common value for the netmask is '255.255.255.0'. It is most helpful to think of the netmask in terms of bits. Each base-10 number between 0-255 represents 8 bits which are either set or not set. Every bit which is set is a network address and every bit which is zero is part of a host address. The first two parts of the address '255.255' always takes these values. If the third number is '255', it means that the domain is divided up into 256 sub networks and then the remaining bits which are zero can be used to give 255 different host addresses on each of the subnets.

If the value had been '255.255.255.254', the network would be divided up into 2^15 subnets, since fifteen of the sixteen bits are one. The remaining bit leaves enough room for two addresses 0 and 1. One of those is reserved for *broadcasts* to all hosts, the other can be an actual host — there would only be room for one host per subnet. This is a stupid example of course, the main point with the subnet mask is that it can be used to trade subnets for hosts per subnet. A value of '255.255.254.0' would allow 128 different subnets with $2 * 256 - 1 = 511$ hosts on each.

We needn't be concerned with the details of the netmask here. Suffice it to say that its value is determined for your entire domain by the network administrator and each host has to be told what the value is.

Each host must also know what convention is used for the *broadcast address*. This is an address which hosts can send to if they wish to send a message to every other host on their subnet simultaneously. It is used a lot by services like NIS to ask if any hosts are willing to perform a particular service. There are two main conventions for the broadcast address: address zero (all host bits are zero) and the highest address on the subnet (all host bits are ones). The convention can be different on every subnet and it is decided by the network administrator. When you write a cfengine program you just specify the convention used on your subnet and cfengine works out the value of the broadcast address from the netmask and the host address See ⟨undefined⟩ [broadcast], page ⟨undefined⟩. Cfengine works out the value of the broadcast address using the value of the netmask.

8.4.18 nfstype

```
nfstype = ( nfs-type )
```

This variable is included only for future expansion. If you do not define this variable, its value defaults to "nfs".

At present cfengine operates only with NFS (the network file system). When cfengine looks for network file systems to mount, it adds lines in the filesystem table ('/etc/fstab','/etc/checklist' etc.) to try to mount filesystems of type "nfs". In principle you might want to use a completely different system for mounting filesystems over the network, in which case the 'mount type' would not be "nfs" but something else.

At the time of writing certain institutions are replacing NFS with AFS (the Andrew filesystem) and DFS (from the distributed computing environment). The use of these filesystems really excludes the need to use the mount protocol at all. In other words if you are using AFS or DFS, you don't need to use cfengine's mounting commands at all.

8.4.19 RepChar

```
RepChar  = ( character )
```

The value of this variable determines the characters which is used by cfengine in creating the unique filenames in the file repository. Normally, its value is set to '`_`' and each '`/`' in the path name of the file is changed to '`_`' and stored in the repository. If you prefer a different character, define it here. Note that the character can be quoted with either single or double quotes in order to encompass spaces etc.

8.4.20 Repository

```
Repository  = ( directory )
```

Defines a special directory where all backup and junk files are collected. Files are assigned a unique filename which identifies the path from which they originate. This affects files saved using `disable`, `copy`, `links` and `editfiles` See ⟨undefined⟩ [Disabling and the file repository], page ⟨undefined⟩.

8.4.21 SensibleCount

```
SensibleCount  = ( count )
```

This variable is used by the action `required`. It defines for cfengine what you consider to be the minimum number of files in a 'required' directory. If you declare a directory as being required, cfengine will check to see if it exists. Then, if the directory contains fewer than the value of `sensiblecount` files, a warning is issued. The default value for this variable is 2.

8.4.22 SensibleSize

```
SensibleSize  = ( size )
```

This variable is used by the action `required`. It defines for cfengine what you consider to be the minimum size for a 'required' file. If you declare a file as being required, cfengine will check to see if the file exists. Of course, the file may exist but be empty, so the size of the file is also checked against this constant. If the file is smaller than the value of `sensiblesize` a warning is issued. The default value for this variable is 1000 bytes.

8.4.23 SplayTime

```
SplayTime = ( time-in-minutes )
```

This variable is used to set the maximum time over which cfengine will share its load on a server, See ⟨undefined⟩ [Splaying host times], page ⟨undefined⟩.

8.4.24 site/faculty

```
site    = ( sitename )
faculty = ( facultyname )
```

This variable defines a convenient name for your site configuration. It is useful for making generic rules later on, because it means for instance that you can define the name of a directory to be

```
/$(site)/$(host)/local
```

without having to redefine the rule for a specific site. This is a handy trick for making generic rules in your files which can be imported into a configuration for any site.

`faculty` is a synonym for `site`. The two names may be used interchangeably.

8.4.25 Split

```
Split  = ( character )
```

The value of this variable is used to define the list separator in variables which are expected to be treated as lists. The default value of this variable is the colon '`:`'. Cfengine treats variables containing this character as lists to be broken up and iterated over in the following cases:

- in the 'to' field of a multiple link action,
- in the 'from' field of a copy action,
- in the directory field of a tidy action,
- in the directory field of the files action,
- in the ignore action.

This typically allows communication with PATH-like environment variables in the shell.

8.4.26 sysadm

```
sysadm = ( mail address )
```

The mail address of your system administrator should be placed here. This is used in two instances. If cfengine is invoked with the option `-a`, then it simply prints out this value. This is a handy feature for making scripts. See ⟨undefined⟩ [Using the help scripts], page ⟨undefined⟩.

The administrators mail address is also written into the personal log files which cfengine creates for each user after tidying files, so you should make this an address which users can mail if they have troubles.

8.4.27 Syslog

```
Syslog = ( on/off )
```

This variable activates syslog logging of cfengine output at the 'inform' level.

To set this output level one writes:

```
classes::

 Syslog = ( on )
```

8.4.28 timezone

```
timezone = ( 3-character timezone )
```

The timezone variable is a character string which defines your local timezone. Currently only the first three characters of this string are checked against the timezone which cfengine manages to glean from the system. If a mismatch is detected a warning message is printed. cfengine does not attempt to configure the timezone. This feature works only as a reminder, since the timezone should really be set once and for all at the time the system is installed. On some systems you can set the timezone by editing a file, a procedure which you can automate with cfengine See ⟨undefined⟩ [editfiles], page ⟨undefined⟩.

The value of the `timezone` can be accessed by variable substitution in the usual way:

```
shellcommands:

       "echo ${timezone} | mail ${sysadm}"
```

8.4.29 Verbose

```
Verbose = ( on/off )
```

This variable switches on the output level whereby cfengine reports everything it does during a run in great detail. Normally only urgent messages or clear errors are printed, See ⟨undefined⟩ [Inform], page ⟨undefined⟩. This option is almost equivalent to using the `--verbose` of `-v` command-line options. The only difference is that system environment reporting information, which is printed prior to parsing, is not shown. To set this output level on selected hosts one writes:

```
classes::

  Verbose = ( on )
```

For related more limited output, See ⟨undefined⟩ [Inform], page ⟨undefined⟩.

8.4.30 Warnings

```
Warnings = ( on/off )
```

This variable switches on the parser-output level whereby cfengine reports non-fatal warnings. This is equivalent to setting the command line switch `--no-warn`, or `-w`. To set this output level on selected hosts one writes:

```
classes::

  Warnings = ( on )
```

8.5 classes

The `classes` keyword is an alias for `groups` as of version 1.4.0 of cfengine.

8.6 copy

Cfengine copies files between locally mounted filesystems and via the network from registered servers. The copy algorithm avoids race-conditions which can occur due to network and system latencies by copying first to a file called '*file*.cfnew' on the local filesystem, and then renaming this quickly into place. The aim of this roundabout procedure is to avoid situations where the direct rewriting of a file is interrupted midway, leaving a partially written file to be read by other processes. Cfengine attempts to preserve hard links to non-directory file-objects, but see the caution below. The syntax summary is:

```
copy:

   class::

      master-file
                              dest=destination-file
                              mode=mode
                              owner=owner
                              group=group
                              action=silent/fix
                              backup=true/false
                              symlink=pattern
                              include=pattern
                              exclude=pattern
                              recurse=number/inf/0
                              type=ctime/checksum/sum/byte/binary
                              linktype=absolute/symbolic/relative/hard
                              define=class-list(,:.)
                              force=true/on/false/off
                              size=size limits
                              server=server-host
                              purge=true/false
```

`dest` The destination file is the only obligatory item. This must be the name of an object which matches the type of the master object i.e. if the master is a plain file, the destination must also be the explicit name of a plain file. An implicit 'copy file to directory' syntax is not allowed. Symbolic links are copied as symbolic links, plain files are copied as plain files and special files are copied as special files. If the master and image are directories then all of the child files *which are not directories* are copied from source to destination.

`mode, owner, group`
The file mode, owner and group of the images are specified as in the `files` function See ⟨undefined⟩ [files], page ⟨undefined⟩.

`action` The action may take the values `warn` or `silent`. The default action is `fix`, i.e. copy files. If `warn` is specified, only a warning is issued about files which require updating. If `silent` is given, then cfengine will copy the files but not report the fact.

`force` If set to 'true', this option causes cfengine to copy files regardless of whether it is up to date.

`backup` If the `backup` option is set to "false", cfengine will not make a backup copy of the file before copying.

Copy makes a literal image of the master file at the destination, checking whether the master is newer than the image. If the image needs updating it is copied. Existing files are saved by appending `.cfsaved` to the filename.

`recurse` Specifies the depth of recursion when copying whole file-trees recursively. The value may be a number or the keyword `inf`. Cfengine crosses device boundaries or mounted filesystems when descending recursively through file trees. To prevent this it is simplest to specify a maximum level of recursion.

`symlink` This option may be repeated a number of times to specify the names of files, or wildcards which match files which are to be symbolically linked instead of copied. A global list of patterns can also be defined in the control section of the program See ⟨undefined⟩ [linkcopies], page ⟨undefined⟩.

`include` This option may be repeated a number of times to specify the names of files, or wildcards which match files which are to be included in a copy operation. Specifying one of these automatically excludes everything else except further include patterns. A global list of patterns can also be defined in the control section of the program.

`exclude` This option may be repeated a number of times to specify the names of files, or wildcards which match files which are to be excluded from a copy operation. A global list of patterns can also be defined in the control section of the program 'excludes' override 'includes'. See ⟨undefined⟩ [excludelinks], page ⟨undefined⟩.

`type` Normally cfengine uses the ctime date-stamps on files to determine whether a file needs to be copied: a file is only copied if the master is newer than the copy or if the copy doesn't exist. If the type is set to '`checksum`' or '`sum`', then a secure MD5 checksum is used to determine whether the source and destination files are identical. If '`byte`' or '`binary`' is specified, a byte by byte comparison is initiated.

`server` If you want to copy a file remotely from a server, you specify the name of the server here. This must be the name of a host which is running the `cfd` daemon, and you must make sure that you have defined the variable `domain` in the control section of the '`cfengine.conf`' file. If you don't define a domain you will probably receive an error of the form 'cfengine: Hey! cannot stat file'.

`size` With this option you can specify that a file is only to be copied if the source file meets a size critereon. This could be used to avoid installing a corrupted file (the copying of an empty password file, for instance). Sizes are in bytes by default, but may also be quoted in kilobytes or megabytes using the notation:

number`bytes`
number`kbytes`
number`mbytes`

Only the first characters of these strings are significant, so they may be written however is convenient: e.g. `14kB`, `14k`, `14kilobytes` etc. Examples are:

```
size=<400  # copy if file size is < 400 bytes
size=400   # copy if file size is equal to 400 bytes
size=>400  # copy if file size > 400 bytes
```

`linktype` This option determines the type of link used to make links. This only applies if the file is linked rather than copied because it matches a pattern set by `symlink`. The default type is a direct symbolic link. The values '`relative`' or '`absolute`' may be used, but hard links may not be created in place of copied files, since hard links must normally reside on the same filesystem as their files, and it is assumed that most links will be between filesystems.

`define` This option is followed by a list of classes which are to be 'switched on' if and only if the named file was copied. In multiple (recursive) copy operations the classes become defined if any of the files in the file tree were copied. This feature is useful for switching on other actions which are to be performed after the installation of key files (e.g. package installation scripts etc).

`purge` If this option is set to true, cfengine will remove files in the destination directory which are not also in the source directory. This allows exact images of filesystems to be mantained.

Example:

```
copy:

      /local/etc/aliases dest=/etc/aliases m=644 o=root g=other
      /local/backup-etc  dest=/etc

   solaris::

      /local/etc/nsswitch.conf dest=/etc/nsswitch.conf
```

In the first example, a global aliases file is copied from the master site file '`/local/etc/aliases`' to '`/etc/aliases`', setting the owner and protection as specified. The file gets installed if '`/etc/aliases`' doesn't exist and updated if '`/local/etc/aliases`' is newer than '`/etc/aliases`'. In the second example, '`backup-etc`' is a directory containing master configuration files (for instance, '`services`', '`aliases`', '`passwd`'...). Each of the files in '`backup-etc`' is installed or updated under '`/etc`'. Finally, a global '`nsswitch.conf`' file is kept up to date for solaris systems.

The `home` directive can be used as a destination, in which case cfengine will copy files to every user on the system. This is handy for distributing setup files and keeping them updated:

```
copy:

   /local/masterfiles/.cshrc  dest=home/.cshrc mode=0600
```

You can force the copying of files, regardless of the date stamps by setting the option `force=true` or `force=on`. The default is `force=false` or `force=off`.

8.6.1 Hard links in copying

Hard links are not like symbolic links, they are not merely pointers to other files, but alternative names for the same file. The name of every file is a hard link, the first so to speak. You can add additional names which *really are* the file, they are not just pointers. For the technically minded, they are not separate inodes, they are additional directory references to the same inode. When you perform a copy operation on multiple files, cfengine attempts to preserve hard links but this is a difficult task.

Because a hard link just looks like an ordinary file (it cannot be distingiushed from the original, the way a symbolic link can) there is a danger that any copy operation will copy two hard links to the same file as two separate copies of the same file. The difference is that changes a hard-linked file propagate to the links, whereas two copies of a file are completely independent thereafter. In order to faithfully reproduce all hardlinks to all files, cfengine needs to examine every file on the same filesystem and check whether they have the same inode-number. This would be an enourmous overhead, so it is not done. Instead what happens is that cfengine keeps track of only the files which it is asked to examine, for each atomic copy-command, and makes a note of any repeated inodes within this restricted set. It does not try to go off, wandering around file systems looking to other files which might be hardlinks.

To summarize, cfengine preserves hardlinks during copying, only within the scope of the present search. No backups are made of hard links, only of the first link or name of the file is backed up. This is a necessary precaution to avoid dangling references in the inode table. As a general rule, hard links are to be avoided because they are difficult to keep track of.

8.6.2 Too many open files

In long recursive copies, where you descend into many levels of diretories, you can quickly run out of file descriptors. The number of file descriptors is a resource which you can often set in the shell. It is a good idea to set this limit to a large number on a host which will be copying a lot of files. For instance, in the C shell you would write,

```
limit descriptors 1024
```

Most systems should have adequate defaults for this parameter, but on some systems it appears to be set to a low value such as 64, which is not sufficient for large recursive tree searches.

8.7 defaultroute

Dynamical routing is not configurable in cfengine, but for machines with static routing tables it is useful to check that a default route is configured to point to the nearest gateway or router. The syntax for this statement is simply:

```
defaultroute:

   class::

      my_gateway
```

For example:

```
defaultroute:

  most::

     129.240.22.1


  rest::

     small_gw
```

Gateways and routers usually have internet address `aaa.bbb.ccc.1` — i.e. the first address on the subnet. You may use the numerical form or a hostname for the gateway.

8.8 disks

This is a synonyn for `required`, See ⟨undefined⟩ [required], page ⟨undefined⟩. This action tests for the existence of a file or filesystem. It should be called after all NFS filesystems have been mounted. You may use the special variable `$(binserver)` here.

```
disks:

  /filesystem freespace=size-limit define=class-list(,:.)
```

Files or filesystems which you consider to be essential to the operation of the system can be declared as 'required'. Cfengine will warn if such files are not found, or if they look funny.

Suppose you mount your filesystem `/usr/local` via NFS from some binary server. You might want to check that this filesystem is not empty! This might occur if the filesystem was actually *not* mounted as expected, but failed for some reason. It is therefore not enough to check whether the directory `/usr/local` exists, one must also check whether it contains anything sensible.

Cfengine uses two variables: `sensiblesize` and `sensiblecount` to figure out whether a file or filesystem is sensible or not. You can change the default values of these variables (which are 1000 and 2 respectively) in the `control` section. See ⟨undefined⟩ [control], page ⟨undefined⟩.

If a file is smaller than `sensiblesize` or does not exist, it fails the 'required' test. If a directory does not exist, or contains fewer than `sensiblecount` files, then it also fails the test and a warning is issued.

```
disks:

   any::

      /$(site)/$(binserver)/local
```

If you set the `freespace` variable to a value (the default units are kilobytes, but you may specify bytes or megabytes), e.g.

8.9 directories

Directories declarations consist of a number of directories to be created. Directories and files may also be checked and created using the `touch` option in the `files` actions. See ⟨undefined⟩ [files], page ⟨undefined⟩.

The form of a declaration is:

```
directories:

    classes::

        /directory
                         mode=mode
                         owner=uid
                         group=gid
                         define=classlist
```

For example

```
directories:

  class::

     /usr/local/bin  mode=755 owner=root group=wheel
```

The form of the command is similar to that of `files` but this command is only used to create new directories. Valid options are `mode`, `owner`, `group` and are described under `files` See ⟨undefined⟩ [files], page ⟨undefined⟩. This interface is only for convenience. It is strictly a part of the 'files' functionality and is performed together with other 'files' actions at run time.

The creation of a path will fail if one of the links in the path is a plain file or device node. A list of classes may optionally be defined here if a directory is created.

8.10 disable

Disabling a file means renaming it so that it becomes harmless. This feature is useful if you want to prevent certain dangerous files from being around, but you don't want to delete them— a deleted file cannot be examined later. The syntax is

```
disable:

  class::

     /filename
                    type=plain/file/link/links
                    rotate=empty/truncate/numerical-value
                    size=numerical-value
                    define=classlist
```

Cfengine renames a given file by appending the name of the file with the suffix '`.cfdisabled`'. A typical example of a file you would probably want to disable would be the `/etc/hosts.equiv` file which is often found with the '`+`' symbol written in it, opening the system concerned to the entire NIS universe without password protection! Here is an example:

```
disable:

      /etc/hosts.equiv
      /etc/nologin
      /usr/lib/sendmail.fc

   sun4::

      /var/spool/cron/at.allow
```

Hint: The last example disables a file which restricts access to the `at` utility. Such a command could be followed by a file action:

```
files:

   some::

      /var/spool/cron/at.allow =0644 N [root] [wheel] touch
```

See ⟨undefined⟩ [files], page ⟨undefined⟩ which would create an empty security file '`at.allow`'. See also your system manual pages for the `at` command if you don't understand why this could be useful.

Disabling a link deletes the link. If you wish you may use the optional syntax

```
disable:
```

```
/directory/name type=file
```

to specify that a file object should only be disabled if it is a plain file. The optional element `type=` can take the values `plain`, `file`, `link` or `links`. If one of these is specified, cfengine checks the type and only disables the object if there is a match. This allows you to disable a file and replace it by a link to another file for instance.

NOTE that if you regularly disable a file which then gets recreated by some process, the disabled file '*filename*`.cfdisabled`' will be overwritten each time cfengine disables the file and therefore the contents of the original are lost each time. The `rotate` facility was created for just this contingency.

The disable feature can be used to control the size of system log files, such as '`/var/adm/messages`' using a further option `rotate`. If the value rotate is set to 4, say,

```
disable:

   filename  rotate=4
```

then cfengine renames the file concerned by appending '.1' to it and a new, empty file is created in its place with the same owner and permissions. The next time disable is executed '.1' is renamed to '.2' and the file is renamed '.1' and a new empty file is created with the same permissions. Cfengine continues to rotate the files like this keeping a maximum of four files. This is similar to the behaviour of syslog.

If you simply want to empty the contents of a log file, without retaining a copy then you can use `rotate=empty` or `rotate=truncate`. For instance, to keep control of your World Wide Web server logs:

```
disable:

   Sunday|Wednesday::

       /usr/local/httpd/logs/access_log  rotate=empty
```

This keeps a running log which is emptied each Sunday and Wednesday.

The `size=` option in disable allows you to carry out a disable operation only if the size of the file is less than, equal to or greater than some specified size. Sizes are in bytes by default, but may also be quoted in kilobytes or megabytes using the notation:

```
numberbytes
numberkbytes
numbermbytes
```

Only the first characters of these strings are significant, so they may be written however is convenient: e.g. `14kB`, `14k`, `14kilobytes` etc. Examples are:

```
size=<400  # disable if file size is < 400 bytes
size=400   # disable if file size is equal to 400 bytes
```

```
size=>400  # disable if file size > 400 bytes
```

This options works with `rotate` or normal disabling; it is just an extra condition which must be satisfied.

If a disable command results in action being taken by cfengine, an optional list of classes becomes can be switched on with the aid of a statement `define=`*classlist* in order to trigger knock-on actions.

8.11 editfiles

You can perform simple control or editing on textfiles using a number of commands. These are fairly limited but are sufficient for many purposes and are definitely superior to using `awk` or `sed`[1]. The form of an editing command is

```
editfiles:

   class::

      { file-to-be-edited

      action "quoted-string..."
      }
```

Here are some examples:

```
editfiles:

   sun4::

      { /etc/netmasks

      DeleteLinesContaining "255.255.254.0"
      AppendIfNoSuchLine "128.39  255.255.255.0"
      }

   PrintServers::
      { /etc/hosts.lpd

      AppendIfNoSuchLine "tor"
      AppendIfNoSuchLine "odin"
      AppendIfNoSuchLine "borg"
      }
```

The first of these affects the file '`/etc/netmasks`' on all SunOS 4 systems, deleting any lines containing the string "255.255.254.0" and Appending a single line to the file containing "128.39 255.255.255.0" if none exists already. The second affects only hosts in the class 'PrintServers' and adds the names of three hosts: tor, odin and borg to the file '`/etc/hosts.lpd`' which specifies that they are allowed to connect to the printer services on any host in the class 'PrintServers'.

Note that single or double quotes may be used to enclose strings in cfengine. If you use single quotes, your strings may contain double quotes and vice-versa. Otherwise a double quoted string may not currently contain double quotes and likewise for single quoted strings.

---

[1] If you are editing a file which has hard links to it, be aware that editing the file will destroy the hard link references. This is also the case with shell commands. You should avoid hard links whenever possible.

As of version 1.3.0, you can use the '`home`' directive in edit filenames, enabling you to edit files for every user on the system, provided they exist. For example, to edit every user's login files, you would write

```
{ home/.cshrc

 AppendIfNoSuchLine "setenv PRINTER default-printer"
 AppendIfNoSuchLine "set path = ( $path /new/directory )"
}
```

If a user does not possess the named file, cfengine just skips that user. A new file is not created.

The meanings of the file-editing actions should be self-explanatory. Commands containing the word 'comment' are used to 'comment out' certain lines in a file rather than deleting them. `Hash` implies a shell comment of the type

```
# comment
```

`Slash` implies a comment of the C++ type:

```
// comment
```

`Percent` implies a comment of the type:

```
% comment
```

More general comment types may be defined using the `SetCommentStart`, `SetCommentEnd` and `CommentLinesMatching`, `CommentLinesStarting` functions.

A special group of editing commands is based on the GNU Regular Expression package. These use GNU regular expressions to search line by line through text and perform various editing functions. Some of these commands are based on the concept of a file pointer. The pointer starts at line one of the file and can be reset by 'locating' a certain line, or by using the reset-pointer commands. The current position of the pointer is used by commands such as `InsertLine` to allow a flexible way of editing the middle of files.

A simple decision mechanism is incorporated to allow certain editing actions to be excluded. For instance, to insert a number of lines in a file once only, you could write:

```
{ file

 LocateLineMatching "insert point..."
 IncrementPointer   "1"

 BeginGroupIfNoMatch "# cfengine - 2/Jan/95"

   InsertLine "# cfengine - 2/Jan/95"
   InsertLine "/local/bin/start-xdm"

 EndGroup
}
```

Since the first inserted line matches the predicate on subsequent calls, the grouped lines will only be carried out once.

The full list of editing actions is given below in alphabetical order. Note that some commands refer to regular expressions and some refer to 'literal strings' (i.e. any string which is not a regular expression). Variable substitution is performed on all strings. Be aware that symbols such as '`.`', '`*`' and so on are meta-characters in regular expressions and a backslash must be used to make them literal. The regular expression matching functions are the GNU regular expressions, as defined by the `regex-0.12` package See ⟨undefined⟩ [Regular expressions], page ⟨undefined⟩. Readers are referred to the manual for this package for details of the extended special features of GNU regular expressions. If you are not familiar with regular expressions, then be aware that you may always supply an exact string to be matched (this is the simplest regular expression), but be careful about backslashing meta-characters!

`AbortAtLineMatching` *quoted-regex*
: This command sets the value of a regular expression. In all editing operations (except `FixEndOfLine` and `GotoLastLine`) which involve multiple replacements and searches, this expression marks a boundary beyond which cfengine will cease to look any further. In other words, if cfengine encounters a line matching this regular expression, it aborts the current action. BE CAREFUL with this feature: once set, the string remains set for the remainder of the current file. It might therefore interact in unsuspected ways with other search parameters. Editing actions are always aborted as soon as the abort expression is matched. Use `UnsetAbort` to unset the feature.

`Append` *quoted-string*
: Add a line containing the quoted string to the end of the file. This should be used in conjunction with the decision structures `BeginGroupIfNoLineMatching` and `BreakIfLineMatches`.

`AppendIfNoSuchLine` *quoted-string*
: Add a line containing the quoted string to the end of the file if the file doesn't contain the exact line already.

`AppendIfNoLineMatching` *quoted-regex*
: A new version of the older `AppendIfNoSuchLine` which uses a regular expression instead of a literal string. The line which gets appended must be set previously using `SetLine`.

`AppendToLineIfNotContains` *quoted-string*
: This commands looks for an exact match of the quoted string in the current line. If the quoted string is not contained in the line, it is appended. This may be used for adding entries to a list, See ⟨undefined⟩ [FAQS and Tips], page ⟨undefined⟩.

`AutoCreate`
: If this command is listed anywhere in the file action list, cfengine will create the named file if it doesn't exist. Normally cfengine issues an error if the named file does not exist. New files are created with mode 644, read access for everyone and write access for the cfengine user (normally root). Note that if you set this, BeginGroupIfFileIsNewer will always be true.

`AutomountDirectResources` *quoted-string*
: This command is designed to assist with automounter configuration for users wishing to use the automounter for NFS filesystems, but still use the cfengine mount model.
: Applied to the current file, it is equivalent to saying: for each of the mountable resources in the list See ⟨undefined⟩ [mountables], page ⟨undefined⟩, append if not found a line

for a direct automount map command, to the current file. The string which follows can be used to specify any special mount options e.g. `"-nosuid"` for non setuid mounting (of all the mountables). Note that this is added to the current file and not to a file named '`/etc/auto_direct`'.

`BeginGroupIfFileExists` *quoted-string*
: The lines following, up to the first `EndGroup` are executed if the quoted filename exists (can be statted). Files which are not readable by the running process are for all intents and purposes non-existent.

`BeginGroupIfFileIsNewer` *quoted-string*
: The lines following, up to the first `EndGroup` are executed if the quoted filename is newer than the file being edited.

`BeginGroupIfNoLineContaining` *quoted-string*
: The lines following, up to the first `EndGroup` are executed if the quoted string does not appear in any line in the file.

`BeginGroupIfNoLineMatching` *quoted-regex*
: The lines following, up to the first `EndGroup` are executed if the quoted regular expression does not match any line in the file.

`BeginGroupIfNoMatch` *quoted-regex*
: The lines following, up to the first `EndGroup` are executed if the quoted regular expression does not match the current line.

`BeginGroupIfNoSuchLine` *quoted-string*
: The lines following, up to the first `EndGroup` are executed if the quoted literal string does not match any line in the file.

`BreakIfLineMatches` *quoted-regex*
: Terminates further editing of the current file if the current line matches the quoted regular expression.

`CommentLinesMatching` *quoted-regex*
: Use the current value of the comment delimiters set using `SetCommentStart` and `SetCommentEnd` to comment out lines matching the given regular expression in quotes.

`CommentLinesStarting` *quoted-string*
: Use the current value of the comment delimiters set using `SetCommentStart` and `SetCommentEnd` to comment out lines starting with the quoted literal string.

`CommentNLines` *quoted-string*
: Comments up to N lines from the current file, starting from the location of the current line pointer. If the end of the file is reached and less than N lines are deleted, a warning is issued, but editing continues. The current value of the comment delimiters is used to determine the method of commenting, (see `SetCommentStart`). After the operation the pointer points to the line after the commented lines.

`CommentToLineMatching` *quoted-regex*
: Use the current value of the comment delimiters set using `SetCommentStart` and `SetCommentEnd` to comment out lines from the current position in a file to a line matching the given regular expression in quotes.

`DefineClasses` "*class1*:*class2*:..."
: Activate the following colon, comma or dot-separated list of classes if and only if the file is edited.

`DeleteLinesAfterThisMatching` *quoted-regex*

`DeleteLinesContaining` *quoted-string*
: Delete all lines containing the exact string quoted.

`DeleteLinesMatching` *quoted-regex*
: Delete all lines matching the quoted regular expression.

`DeleteLinesStarting` *quoted-string*
: Delete all lines beginning with the exact string quoted.

`DeleteNLines` *quoted-string*
: Deletes up to N lines from the current file, starting from the location of the current line pointer. If the end of the file is reached and less than N lines are deleted, a warning is issued, but editing continues.

`DeleteToLineMatching` *quoted-regex*
: Delete lines from the current position, up to but not including a line matching the regular expression in the quoted string. If no line matches the given expression, a warning is only printed in verbose mode, but all edits are immediately abandoned.

`EmptyEntireFilePlease`
: Deletes all lines from the current file.

`EndGroup` Terminates a begin-end conditional structure.

`EndLoop` Terminates a loop. See `ForEachLineIn`

`FixEndOfLine`
: The quoted string which follows may be either '`dos`' or '`unix`' to fix the end of line character conventions to match these systems. This command should be executed last of all, since cfengine appends new lines with the conventions of the system on which is was complied during edit operations.

`ForEachLineIn` *quoted-filename*
: This marks the beginning of a for-loop which reads successive lines from a named file. The result is like using `SetLine` for each line in the file. Nested loops are not permitted.

`GotoLastLine`
: Moves the file pointer to the last line in the current file.

`HashCommentLinesContaining` *quoted-string*
: Add a '`#`' to the start of any line containing the quoted string.

`HashCommentLinesMatching` *quoted-regex*
: Add a '`#`' to the start of any line exactly matching the quoted regular expression.

`HashCommentLinesStarting` *quoted-string*
: Add a '`#`' to the start of any line starting with the quoted string.

`IncrementPointer` *quoted-number*
: Increments the value (in lines) of the file pointer by the number of lines specified in the quoted string (as a denary number). e.g. '`"4"`'. Negative values are equivalent to

decrementing the pointer. If a request is made to increment/decrement outside of the file boundaries the pointer 'bumps' into the boundary and remains there, i.e. either at start of file or end of file.

`InsertFile` *quoted-string*
: Inserts the named file after the current line position in the file. This should be used in conjunction with a begin-end construction in order to avoid including the file every time cfengine is run. If the file does not exist, or cannot be opened, there is only a warning issued in verbose mode. Note if the file is empty, or if the current line pointer is not set, the file is inserted at the start of the file.

`InsertLine` *quoted-string*
: Inserts the quoted string as a line at the current position in the file. After the insert, the file pointer is incremented by one so that subsequent inserted lines are placed after the first. This should probably be used in conjunction with the conditional begin-end tests to avoid lines being inserted on every run.

`LocateLineMatching` *quoted-string*
: Moves the current-position pointer to the start of the line matching the quoted regular expression. If there is no match, a warning is only issued in verbose mode, but all editing is immediately aborted. See also `WarnIfNoLineMatching` so that you can get an explicit warning, even out of verbose mode.

`PercentCommentLinesContaining` *quoted-string*
: Add a '`%`' to the start of any line containing the quoted string.

`PercentCommentLinesMatching` *quoted-regex*
: Add a '`%`' to the start of any line exactly matching the quoted regular.

`PercentCommentLinesStarting` *quoted-string*
: Add a '`%`' to the start of any line starting with the quoted string.

`Prepend` *quoted-string*
: Add a line containing the quoted string to the start of the file. This should be used in conjunction with the decision structures `BeginGroupIfNoLineMatching` and `BreakIfLineMatches`.

`PrependIfNoLineMatching` *quoted-regex*
: A new version of the older `PrependIfNoSuchLine` with uses a regular expression instead of a literal string. The string prepended is the one set using `SetLine`.

`PrependIfNoSuchLine` *quoted-string*
: Add a line containing the quoted string to the start of the file if the file doesn't contain the exact line already.

`ReplaceLineWith` *quoted-string*
: Replace the line at the current position with the text in the quoted string. The file pointer remains pointing to this line after the change.

`ReplaceAll` *quoted-regex* `With` *quoted-string*
: Replace all instances of strings matching the regular expression in the first quotes with the exact string in the second set of quotes, throughout the current file. Note that cfengine matches on a left to right basis, with the first match taking precedence, so

if your regular expression matches text ambiguously it is the first occurrence which is replaced. For example, if you replace '`cf.*`' with '`CFENGINE`' and cfengine encounters a line '`hello cfengine cfengine`', then this will be replaced with '`hello CFENGINE`' even though two possible strings match the regular expression. On the other hand if the expression is not ambiguous, say replacing '`cfengine`' with '`CFENGINE`', then the result would be '`hello CFENGINE CFENGINE`'.

`ReplaceLinesMatchingField` *quoted-number*

This command replaces any lines in the current file with the current line set by `SetLine` or `ForEachLineIn`, if the lines are split into fields (e.g. the password file) separated by the `SplitOn` character (':' by default), and the corresponding fields match.

The idea behind this command was to be able to override global passwords (from a file which gets distributed) by new passwords in a local file. Rather than maintaining the files separately, this simply overrides the entries with the new ones See ⟨undefined⟩ [FAQS and Tips], page ⟨undefined⟩.

`ResetSearch` *quoted-string*

Sets the current-position pointer to the line number in the quoted string. '`EOF`' indicates the end of the file.

`RunScript` *quoted-string*

Executes the named script command. Before executing the script any edits are saved to disk. After the script has executed, cfengine reloads the file for any further editing operations. The script (which may be any executable program) is appended with two arguments: the name of the file which is being edited and the system hard class (e.g. sun4, ultrix etc.) of the system executing the script.

CAUTION: cfengine knows nothing about the success or failure of anything that is done during the execution of user scripts. This feature is to be used at the users own peril!

`RunScriptIfLineMatching` *quoted-string*

Executes the script named with the `SetScript` command only if the current file contains a line matching the quoted regular expression.

CAUTION: cfengine knows nothing about the success or failure of anything that is done during the execution of user scripts. This feature is to be used at the users own peril!

`RunScriptIfNoLineMatching` *quoted-regex*

Executes the script named with the `SetScript` command if the current file contains no line matching the quoted regular expression.

CAUTION: cfengine knows nothing about the success or failure of anything that is done during the execution of user scripts. This feature is to be used at the users own peril!

`SetCommentStart` *quoted-string*

Specify which string should be used for starting a comment using the commands `CommentLineMatching` and `CommentLineStarting`. The default is the hash symbol '`#`' followed by a single space.

`SetCommentEnd` *quoted-string*
: Specify which string should be used for ending a comment using the commands `CommentLineMatching` and `CommentLineStarting`. The default is the empty string. For example, you could make C style comments by setting CommentStart to '`/*`' and comment end to '`*/`'.

`SetLine` *quoted-string*
: Sets a current line value which can be appended using `AppendIfNoLineMatching` using a regular expression.

`SetScript` *quoted-string*
: Sets the name of a user-supplied script for editing the current file.

`SlashCommentLinesContaining` *quoted-string*
: Add a '`//`' to the start of any line containing the quoted string.

`SlashCommentLinesMatching` *quoted-regex*
: Add a '`//`' to the start of any line exactly matching the quoted regular expression.

`SlashCommentLinesStarting` *quoted-string*
: Add a '`//`' to the start of any line starting with the quoted string.

`SplitOn` *quoted-string*
: This defines a single character which is to be interpreted as a field separator for editing files with columns. The default value for this is '`:`', as is used in the password and group files. It is used in conjunction with `ReplaceLinesMatchingField`.

`UnCommentLinesContaining` *quoted-string*
: Uncomment all lines in file containing the quoted string as a substring. The comment delimiters are assumed to be those set using SetCommentStart and SetCommentEnd.

`UnCommentLinesMatching` *quoted-regex*
: Uncomment all lines in file matching the quoted regular expression. The comment delimiters are assumed to be those set using SetCommentStart and SetCommentEnd.

`UnCommentNLines` *quoted-string*
: Uncomments N lines starting from the current position, using the currently defined method for commenting. Note that the comment start and end symbols are removed independently, i.e. they are not matched, so that a comment may be spread over several lines. e.g. If using C style '`/*`' and '`*/`' comments, the command `UnCommentNLines "3"` would uncomment

```
/* 1 */
/* 2 */
/* 3 */
```

and also

```
/* 1
   2
   3 */
```

`UnsetAbort` *quoted-string*
: Switches off the feature `AbortAtLineMatching`.

`WarnIfLineContaining` *quoted-string*
: Issue a warning if the quoted string is found as a substring of one or more lines in the file.

`WarnIfLineMatching` *quoted-regex*
: Issue a warning if the quoted regular expression matches one or more lines in the file.

`WarnIfLineStarting` *quoted-string*
: Issue a warning if the quoted string matches the start of one or more lines in the file.

`WarnIfNoLineContaining` *quoted-string*
: Issue a warning if the quoted string is not contained in one or more lines in the file.

`WarnIfNoLineStarting` *quoted-string*
: Issue a warning if the quoted string is not found at the start of one or more lines in the file.

`WarnIfNoSuchLine` *quoted-string*
: Issue a warning if the quoted regular expression does not match one or more lines in the file.

It is suggested that you use these editing functions with caution. Although all possible safeguards have been incorporated into them, it is still possible through carelessness to do damage to important files on your system. Always test editing programs carefully before committing them to your global site configuration.

8.12 files

The `files` facility allows you to touch (create), check for the existence, owner and permissions of files, change the permissions and test for setuid root programs.

8.12.1 Syntax

A files-statement can have several options. We can begin by examining the form of the statement in pseudo-code:

```
files:

   classes::

      /file-object
                          mode=mode
                          owner=uid-list
                          group=gid-list
                          action=fixall/other-options/warnall
                          links=false/stop/traverse/follow/tidy
                          include=pattern
                          exclude=pattern
                          define=classlist
```

An example would be the following:

```
any::

   /var/spool/printQ  mode=0775  r=0 o=daemon g=daemon  act=fixdirs
```

The meaning of these item is sketched out below and becomes clearer on looking at a number of examples. Note that, each of the options below can be written in either upper or lower case and abbreviated by any unique abbreviation.

/directory This is the only obligatory part of a file action. This is a directory at which a file search should begin. This may be a single file or a directory. The recursion specifier may be used to force cfengine to descend into subdirectories in a controlled fashion, starting from this point, checking files there also. The wildcard `home` may also be used. See ⟨undefined⟩ [home directive], page ⟨undefined⟩.

`mode=`*modestring*
: Specifies what the allowed permissions for files are. If cfengine finds that a file's mode is incorrect, the value of the `action` option determines what will be done about it. The modestring should consist of either a three digit octal numbers with '`+`', '`-`' or '`=`' symbols, or a text string like that used by the command `chmod`. For instance: `mode=u=rwx,og+rx` would mean set the read/write and execute flags for the user (file owner) and add the read/execute flags for others and group bits. An example of the numerical form might be `-002` which would mean that the read-for-others flag should

either not be set or should be unset, depending on the action you choose. `+2000` would mean that the setuid flag should be present or set, depending on the action. `+2000,-002` would be a combination of these. The '`=`' sign sets to an absolute value, so `=755` would set the file mode to mode 755.

`recurse=`*number/inf*
: This specifier tells cfengine whether or not to recurse into subdirectories. If the value is zero, only the named file or directory is affected. If the value is 1, it will open at most one level of subdirectory and affect the files within this scope. If the value is `inf` then cfengine opens all subdirectories and files beginning from the specified filename.See ⟨undefined⟩ [Recursion], page ⟨undefined⟩.

`owner=`*owner list*
: This is a list of allowed owners, or uids by number, separated by commas. For example `root,2,3,sysadm`. In cases where you ask cfengine to fix the ownership automatically, the owner will be set to the first owner in the list if and only if it is not one of the named uids in the list.

`group=`*group list*
: This is a list of allowed groups, or gids by number, separated by commas. For example `wheel,2,3,sysadm`. In cases where you ask cfengine to fix the ownership automatically, the group will be set to the first group in the list if and only if it is not one of the named gids in the list.

`action=`*action*
: The action is one of the following keywords.

```
warnall warndirs warnplain
 fixall fixdirs fixplain
touch linkchildren create
```

The upper line results only in warnings being issued. The actions beginning 'fix' prompt cfengine to fix encountered problems without bothering the user. No message is issued unless in verbose mode. The special features on the third line will be explained separately.

`include=`*wildcard/pattern*
: You can include this option several times to specify specific patterns which are to be included in the search. Once you specify one pattern you exclude all files not matching at least one of the patterns. The case be useful for restricting a search, or for modifying the permissions of only certain files.

`exclude=`*wildcard/pattern*
: You can include this option several times to specify specific patterns which are to be excluded from the search. This overrides any patterns given in the `include=` list.

`links=`*stop/traverse/tidy*
: Normally cfengine does not descend into subdirectories which are pointed to by symbolic links. If you wish to force it to do so (without using the `-l` command line option) you may give this option the value `true`, or `traverse`, or `follow`. To specify no recursion you set the value `false` or `stop`. Note that the value set here in the cfengine program *always overrides* the value set by the `-l` command line option, so you can protect certain actions from this command line option by specifying a negative value

here. If you specify no value here, the behaviour is determined by what you specify on the command line.

The value `links=tidy` has the same effect as the '`-L`' command line option except that here it may be specified per item rather than globally. Setting this value causes links which point to non-existent files to be deleted.

If the warn directive is used (for directories, plain files or both) then only a warning message is issued if the file being tested does not match the specification given. If the fix directives are used then cfengine does not issue a warning, it simply fixes the value silently. Non-existent files are created by the `touch` command. A directory may be touched (created) by writing the filename `/a/b/c/.` with a dot as the last character. (This may also be achieved with the `directories` directive. See ⟨undefined⟩ [directories], page ⟨undefined⟩).

`define=`*classlist*
: If a file operation results in action being taken to fix a file, the colon, comma or dot separated list of classes becomes defined. Warnings do not activate the classes.

The default values are `mode=+000`, `recurse=0`, `action=warnall` and any owner or group is acceptable. The default for `links` is to not traverse links unless the `-l` option is set on the command line.

8.12.2 Recursion

The recursion specifier tells cfengine what to do, starting from `/directory name`. A value of `r=0` means 'no recursion' and any checking is limited only to the named file or directory. A value of `r=inf` implies unlimited recursion. Cfengine then descends into all subdirectories checking or setting the permissions of files until it 'bottoms out' at a plain file. A value such as `R=4` means descend recursively into subdirectories, but no more than four levels. This is a useful safety net in preventing unforeseen accidents. A recursive search also bottoms out on device boundaries and symbolic links (provided the `-l` option is not used).

8.12.3 Directory permissions

When you specify the permissions for a whole file tree, using the recursion specifier it is awkward to have to remember that directories must be executable. cfengine will do this for you automatically. If you specify that a file tree is to have a read flag set, cfengine will ensure that the corresponding execute flag is also set for directories which live in the tree. So the command

```
files:

  myclass::

      /dir  mode=a+rw r=inf fixall
```

would set all plain files to mode 644 and all directories to 755, that is read/write for everyone on plain files and read/write/execute for everyone on directories.

8.12.4 `home` directive

If you want to check the files of all the users who have their login areas on the current host, you can use a wildcard directive `home` instead of a directory name. In this case the file action iterates over all home directories physically on the current host. The home directories are, of course, located by searching for files which match

```
$(mountpattern)/$(homepattern)
```

i.e. the values which are specified in the `control` part of the program. For example the following line is a very useful service to ignorant users.

```
files:

  any::

    home mode=o-w r=inf act=fixall
```

It ensures automatically that no user has files which can be written to by other arbitrary users.

As a corollary to this, you may write something like

```
  any::

     home/www mode=a+r fixall
```

to specify a special subdirectory of every users' home directory. This statement would check that all of the files in users' world wide web directories were readable for everyone.

8.12.5 Owner and group wildcards

If you do not want to explicitly state the owner or group of a file you may simply omit the group or owner options.

/*file-object* `m=0664 r=inf`

This example generate a warning if any files under the named directory do not have permission read/write for all users.

8.12.6 Files linkchildren

The `linkchildren` facility is almost identical to that already described under `links`. See ⟨undefined⟩ [Link Children], page ⟨undefined⟩. The only difference here is that the ownership and permissions on the links are set all in one operation. For example:

myclass::

```
    /local/lib/emacs m=0770 o=me g=mygroup act=linkchildren
```

8.12.7 touch

The `touch` facility creates a new file with the specified permissions and ownership, or corrects the permissions and ownership of an existing file, in addition to updating the time stamps.

```
myclass::

   /newfile mode=0644 action=touch
```

8.12.8 create

This is like `touch` except that an existing file's time stamps, permissions and ownership will not be modified if the file already exists. If the file does not exist, the attributes are set to the values specified, or to the default values of `0644`.

8.13 groups

The `groups` action (equivalently referred to as `classes` as of version 1.4.0) is used to define classes which stand for groups of hosts. If you use the NIS (network information service) facility for defining *netgroups* then this idea will already be familiar to you and you can probably use your already-defined netgroups in cfengine.

To define a group, you simply make a list and assign it a name. Here is an example of the syntax:

```
groups:

   science = ( saga tor odin )

   packages = ( saga )

   AllHomeServers   = ( saga )
   AllBinaryServers = ( saga )

   OIH_servers = ( saga )
   OIH_clients = ( tor odin )
```

To include a list of hosts from a NIS netgroup, you use the '`+`' symbol, or the '`+@`' construction. For example:

```
groups:

   science = ( +science-allhosts )

   physics = ( +physics-allhosts )

   physics_theory = ( +@physics-theory-sun4 dirac feynman schwinger )
```

Using an enormous netgroup does not use up any space. A group declaration results in the storage of only the class name regardless of how many hosts are in the list. The rule is that the left hand side of the assignment becomes defined (true) if the list on the right hand side includes the host which is parsing the file — i.e. `$(host)`.

In some cases your netgroups will not correspond exactly to the list you want, but it might be more convenient to use a netgroup *except* for certain hosts. You can 'undefine' or remove hosts from the netgroup list by using the minus '-' symbol. For example:

```
group = ( +mynetgroup -specialhost -otherhost )
```

which means, of course, all hosts in netgroup `mynetgroup` except for `specialhost` and `otherhost`. Finally, you may also subtract two netgroups in the following manner.

```
group = ( +bignetgroup -smallnetgroup )
```

The ‘minus’ command effectively eliminates its members from `bignetgroup` if they exist within that group. If none of the hosts in `smallnetgroup` exist in `bignetgroup` then the command has no effect.

Groups may now contain previously defined cfengine groups too. This allows one class to inherit the attributes of another class, for instance:

```
AllSun4Hosts   = ( sonny sunny solar stella )
AllUltrixHosts = ( ully olly wally golly )

AllBSD = ( AllSun4Hosts AllUltrixHosts )
```

The classes on the right hand side are effectively ORed together into the left hand side. This enables complex classes to be constructed from several other basic classes, e.g.

```
SpecialTimes = ( Hr00 Monday Day1 )
```

which evaluates to true every day when it between 00:00 hours and 00:59, all day Monday and all day on the first day of every month.

Finally, you can define groups (strictly classes) by the result of a shell command. A shell command or program is deemed to be ‘true’ if it exits with a status of zero, i.e. it calls `exit(0)`. Any other value is taken to be false. You can include shell commands as the members of groups in order to define classes based on the outcomes of your own scripts by enclosing the script in single or double quotes:

```
 have_cc = ( '/bin/test -f /usr/ucb/cc' )
```

The class `have_cc` will then be defined if the shell command returns true. Of course, you can put any script or program in the single quotes as long as they adhere to the convention that zero exit status means true. If you have several members which are shell commands, then the effect is to make the class the logical OR of the scripts' results.

As of version 1.4.0, you may use the synonym `classes` for `groups`.

8.14 homeservers

The `homeservers` declaration need only be used if you are using cfengine's model for mounting NFS filesystems. This declaration informs hosts of which other hosts on the network possess filesystems containing home directories (login areas) which client hosts should mount.

A sample homeserver declaration looks like this:

```
homeservers:

   Physics::  einstein
   Math::     riemann euler
```

The meaning of this declaration is the following. Any host which finds itself to be a member of the classes on the left hand side of the assignment need to mount all home directory resources from the hosts on the right hand side of the assignment. The pattern variable `homepattern` is used to determine which resources are home directories in the list of `mountables`. See ⟨undefined⟩ [mountables], page ⟨undefined⟩.

Let us consider an example in which `homepattern` is set to the wildcard value '`home?`' and the mountables list is given by

```
mountables:

   einstein:/mysite/einstein/home1
   einstein:/mysite/einstein/home2
   riemann:/mysite/riemann/local
   euler:/mysite/euler/home1
```

Any host in the group `Physics` would now want to mount all home directories from the host `einstein`. There are two of these. Both the filesystems listed for `einstein` match the `homepattern` variable since they end in '`home?`'. cfengine would therefore take this to mean that all hosts in `Physics` should mount both of these filesystems.

Hosts in `Math`, on the other hand, should mount only homedirectories from the hosts `riemann` and `euler`. There is only a single filesystem on `riemann` and it does not match `homepattern`, so it is not mounted. On `euler` there is a match, so this filesystem will be added to the appropriate hosts.

Cfengine picks out home directory resources from the `mountables` *list by trying to match the* `homepattern` *variable, starting from the end of the directory name. You do not therefore have to use the designation* `/site/host/home?` *but this is a simple choice and is highly recommended.*

8.15 ignore

When you specify a recursive search as part of a `files`, `tidy` or `copy` action, you would sometimes like to exclude certain directories from the list of sub directories. In most cases you will want to do this on a per-command basis (see the pages for these actions separately), but you can also make a global ignore list. This can be accomplished by adding the directory to the ignore-list. The syntax is

```
ignore:

   wildcards/directories/filenames
```

For example:

```
ignore:

   any::

      #
      # Prevent tidying .X11 directories in /tmp where
      # window managers write semaphores
      #

      .X11

      #
      # Don't tidy emacs locks
      #

      !*
      /local/lib/gnu/emacs/lock/
      /local/tmp
      /local/bin/top
      /local/lib/tex/fonts
      /local/etc
      /local/www
      /local/mutils/etc/finger.log
```

None of the above directories will be checked or entered during recursive descents unless a specific command is initiated to search those directories with their names as the top of the search tree.

A handy tip if you are tidying '`/tmp`' recursively is to include the directory '`.X11`' here. This directory is used by the X-windows system and deleting it while a window manager has an open session can cause the user some trouble.

Ignore refers to all recursive searches in tidy, files, copy and links.

8.16 import

To break up a large configuration file into smaller files you can use the include directive. This conditionally reads in files if the class on the left hand side of the assignment matches the host parsing the file. This enables also a variety of cfengine configuration scripts to read in a standard set of default settings. The syntax of the statement is:

```
import:

   any::

      cf.global_classes


   linux::

      cf.linux_classes
```

Note that, if you define variables in an imported file they will not be defined for operations in their parent files. This because cfengine reads in all the import files after the main file has been parsed—not at the place where you call import in your script. This means that variables or macros defined in imported files are only defined after the main program. Variables from earlier files are inherited by later includes, but not *vice-versa*.

8.17 links

The symbolic links function is one of the greatest plusses in cfengine as a system administration tool. It allows you to do two things: check single links for correctness and consistency (or make them if they do not exist), and check or make links to every file in a designated directory. This latter feature is called multiple linking or linking children. The `linkchildren` feature is also available from the `files` action See ⟨undefined⟩ [files], page ⟨undefined⟩. The syntax of a link item is:

```
 from-link ->[!] to-object
or
 from-link +>[!] to-object

          type=symbolic/absolute/abs/hard/relative/rel
          copy=pattern
          recurse=number/inf/0
          copytype=checksum/ctime
          include=pattern
          exclude=pattern
          action=silent
          deadlinks=kill/force
          define=classlist
```

The special variable `$(binserver)` *can be used in* `links`.

8.17.1 Single links

To define a single link, you create an entry of the following form:

```
links:

   class::

      linkname -> object_to_link_to
      linkname -> ./relative_link
      linkname -> ../relative_link
```

If links exists and point to their intended destinations then no action is taken. If a link exists but points incorrectly then a warning is issued, unless the pling operator '`!`' is given, in which case the correct value is forced. If the link exists and points to a file which does not exist a warning is issued unless the command line option `-L` is used, in which case the link is deleted. See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩.

Here is an example of some valid link statements.

```
links:
```

```
Physics.sun4::

 /usr/local       -> /$(site)/$(host)/local
 /home            -> /$(site)/$(host)/u1
 /etc/sendmail.cf -> /usr/local/mail/etc/global-sendmail.cf

 /usr/lib/sendmail ->! /local/lib/sendmail
```

cfengine makes any directories which are required leading up to the link name on the left hand side of the arrow automatically. In the last example the 'pling' forces cfengine to make the link even if a file for link exists previously. Plain files are saved by appending '`.cfsaved`' to the filename, or by moving to a repository, whereas old links are removed. The same effect can be enforced globally using the `-E` option, but only if the program is run interactively. (In this case a prompt is issued to make sure that you wish to use such a big hammer on your system!)

The link operation accepts a number of parameters

`type=`*hard/relative/absolute*
: If the link type is hard, a hard link is created See ⟨undefined⟩ [Hard links], page ⟨undefined⟩. Symbolic links may specify two special types. If `relative` is selected, and the 'to' object is an absolute path name, the link name will be rewritten as a pathname relative to the source file, using '`.`' and '`..`' to move relative to the current directory. For instance, a link from '`/usr/local/file`' to '`/usr/file`' would be linked as '`./../file`'. If the 'to' object is already relative, this has no effect.

  If `absolute` is specified, cfengine will try to resolve the true path location of the 'to' object, expanding any symbolic links or dots in the path name, up to a maximum of four levels of symbolic links.

`copy=`*pattern*
: This option can be repeated any number of times to build up a list of filenames or wildcards which are to be copied rather than linked symbolically. The copy is made on an age-comparison basis. A global variable may also be set to invoke this feature See ⟨undefined⟩ [copylinks], page ⟨undefined⟩. Directories cannot be copied in this way.

`copytype=`*checksum/ctime*
: This specifies the basis for deciding whether to update a file which is to be copied instead of linked See ⟨undefined⟩ [copy], page ⟨undefined⟩.

`nofile=`**kill**/*force*
: This decides what happens to links which point to non-existent files. The default action is to remove such links, or refuse to create them. By setting the *force* option you can force cfengine to make symbolic links to files which do not exist. This is useful for setting up links to filesystems which are not permanently mounted.

`exclude=`*pattern*
: This option can be repeated any number of times to build up a list of filenames or wildcards which are to be excluded from the linking process. A global variable may also be set to invoke this feature See ⟨undefined⟩ [excludelinks], page ⟨undefined⟩.

`recurse=`*number*/`inf`
: This option can only be used with multiple link operations See ⟨undefined⟩ [Multiple Links], page ⟨undefined⟩. If this option is specified, cfengine links only non-directory

objects. Directories are instead created and links within those directories are also created. The value of this option specifies the maximum number of levels to which cfengine should recursively descend a link tree. `inf` means infinite recursion. Cfengine also ignores files and directories in the ignore list See ⟨undefined⟩ [ignore], page ⟨undefined⟩.

`define=`*classlist*

If a link is created or replaced, the colon, comma or dot separated list of classes becomes defined.

The final feature of the links facility is connected to the use of the cfengine model for mounting NFS filesystems. In particular it concerns the variable `$(binserver)`. The easiest way to understand this feature is to illustrate a couple of examples. Consider the following:

```
links:

   any::

      /local -> /${site}/${binserver}/local
```

The result of this command is quite different depending on which host is executing it. The variable `$(site)` clearly has a fixed value, but the variable `$(binserver)` might expand to any valid binary server for the host executing the program. See ⟨undefined⟩ [binservers], page ⟨undefined⟩. The procedure cfengine adopts is to go through its list of mountables, keeping only those mountable resources which belong to defined binary servers for the current host. It then attempts to match a filesystem by substituting `$(binserver)` with each of its valid binservers in turn and it matches the first one binary server which yields an existing file.

Note that every host is a binary server for itself, so that the value of `$(binserver)` which has absolute priority is alway the same as the value of `$(host)`. This ensures that the link will always be made to a local filesystem if the rules of the model are upheld.

8.17.2 Multiple Links

With the link symbol `+>`, you opt to link all of the files in a directory to corresponding files in another directory. This procedure is sometimes useful for installing software. In the example

```
links:

  myclass::

     /usr/local/bin +>  /usr/local/lib/perl/bin
     /opt           +>! /local
```

every file in the directory `/usr/local/lib/perl/bin` is linked symbolically to a corresponding file in `/usr/local/bin`. The 'pling' character forces cfengine to replace old links or plain files already existing. Old links are removed, whereas old files are saved by appending '`.cfsaved`' to the filename See ⟨undefined⟩ [repository], page ⟨undefined⟩.

Each time cfengine runs it goes through all of the files in the directory concerned and checks the appropriate link accordingly. If new files appear, new links will be added. If a file disappears but the link to it remains, a warning will be issued, unless the `-L` command line option is used, in which case the link is deleted. See ⟨undefined⟩ [Runtime Options], page ⟨undefined⟩.

8.17.3 Link Children

The linkchildren directive is a closely related to the cfengine model for NFS filesystems. It is a way of making links which embodies a rudimentary kind of 'intelligence'.

Consider the following:

```
links:

   any::

      /usr/local/lib/emacs +> linkchildren
```

The word `linkchildren` automatically tells cfengine that it should look for an appropriate file to link to on a binary server for the current host. The exact meaning of the above statement is as follows. cfengine begins searching though the list of mountable resources, discarding any filesystems which do not belong to valid binary servers. It looks for a filesystem ending in 'emacs' (the last link of the left hand side). If all is well, these file systems are already mounted and they can be searched. If no resource is found ending in 'emacs', we go to the next link `lib` and look for a filesystem ending in 'lib'. If this is not found we go to `local` and so on. When a match is made, cfengine then tries to locate the file by checking whether it exists relative to the matched filesystem. For example, suppose 'local' matched with `host:/site/host/local`. It would then try to locate `host:/site/host/local/lib/emacs` and link all of the children therein to the local file directory `/usr/local/lib/emacs`.

Here is another example which makes reference to the cfengine model for mounting NFS filesystems. Suppose you have a host with some spare disk space. You want to mount `/usr/local` from the binary architecture server, but you also want to use the disk you have locally. The following lines

```
links:

   electron::

      /$(site)/electron/local +> linkchildren

   any::

      /usr/local              -> /$(site)/$(binserver)/local
```

have the effect of creating a directory `/$(site)/electron/local` and filling it with links to all of the files and directories on the binary server's mounted filesystem. It results in an exact copy (by linkage) on the local disk, but does not use up your local disk space. The space you have remaining could, for example, be used for software with a special license for that host. The second link links `/usr/local` to the 'nearest' binary server. But the nearest binary server is always `$(host)` which means this evaluates to a file which now exists because of the first command, so on the host 'electron' the directory `/usr/local` ends up being a link to `/$(site)/electron/local` which is full of links to the binary server.

If you've caught your breath after that mouthful you probably have mixed feelings about creating a bunch of links in this way. What happens if the files they point to are removed? Then you are left with a lot of useless links. Actually this is no problem for cfengine, since you can ask cfengine to simply remove links which point to non-existent files See ⟨undefined⟩ [files], page ⟨undefined⟩. Nevertheless, this feature clearly requires some caution and is mainly a spice for advanced users of the cfengine model.

8.17.4 Relative and absolute links

When specifying symbolic linking, you can ask cfengine to change the link type to be either relative to the source or to be an absolute path. What this means is the following. Consider the following link:

```
/var/tmp/cfengine -> /local/cfengine
```

If we add the option `type=relative`, then instead of creating a link which points to '`/local/cfengine`', the link is created pointing to the location

```
./../../local/cfengine
```

In other words, the link is relative to the calling directory '`/var/tmp`'.

If a link is specified as being absolute with the option `type=absolute`, then cfengine attempts to resolve to value of the link so as to be the true path of the target. If the target name contains a symbolic link, then this is expanded as far as possible to give the true path to the file. For example, if '`/local`' is really a link to '`/site/myhost/local`' then the link would point to '`/site/myhost/local/cfengine`'.

8.17.5 Hard Links

Cfengine will also allow you to create hard links to regular files. A hard link is in every way identical to the original file, it merely has a different name (technically, it is a duplicate inode). To create a hard link you use the link-option `type=hard`. For example:

```
links:

   /directory/newname -> /directory/othername type=hard
```

Cfengine will not create hard links to directories or other special files. This is always a slightly dubious practice and is best avoided anyway. POSIX says that the hard link can be on a different device to the file it points to, but both BSD and System 5 restrict hard links to be on the same device as their predecessors. Cfengine has no policy on this, but—in the theoretical case in which the hard link and the predecessor were on different file systems—it becomes near impossible to determine with certainly between a hard link and a very similar regular file, and thus cfengine issues a warning in verbose mode about this eventuality. Provided both link and predecessor are on the same filesystem cfengine determines the status of hard links by comparing the device and inode numbers of the file pointed to.

8.18 mailserver

The `mailserver` declaration need only be used if you are using cfengine's model for mounting NFS filesystems. This declaration informs hosts of which NFS filesystem contains mail for its users. All hosts apart from the mail-host itself must then mount the mail spool directory across the network. The declaration looks like this:

```
mailserver:

    class::      mailhost:/var/spool/mail
```

The result of the `checkmail` command in the action-sequence is now to mount the filesystem `/var/spool/mail` on the host `mailhost`. This action is carried out on any machine which does not already have that filesystem mounted.

The mail spool directory is mounted, by default, onto the official mail spool directory for the system which is parsing the program. In other words, on an HPUX system, the spool directory is mounted on `/usr/mail` by default, whereas on a Sun system it would be mounted on `/var/spool/mail`. The default location can be changed by using the resource file. See ⟨undefined⟩ [cfrc resource file], page ⟨undefined⟩.

8.19 miscmounts

If you do not use the cfengine model for statically mounting NFS filesystems (or if there are filesystems which do not naturally fall into the bounds of that model) then you can still statically mount miscellaneous filesystems using a statement of the form:

```
miscmounts:

   class::

      infohost:source-directory  destination  mode
```

For example

```
physics::

   libraryserver:/$(site)/libraryserver/data
                        /$(site)/libraryserver/data ro
```

This statement would mount the directory '`/$(site)/libraryserver/data`' physically attached to host libraryserver onto a directory of the same name on all hosts in the group `physics`. The modes `ro` and `rw` signify read-only and read-write respectively.

8.20 mountables

The `mountables` declaration need only be used if you are using cfengine's model for mounting NFS filesystems. This declaration informs hosts of what filesystem resources are available for mounting. This list is used in conjunction with `binservers` and `homeservers` to determine which filesystems a given host should mount, according to the cfengine model.

The syntax of the list is:

```
mountables:

   class::

   server:/site/server/u1
   server:/site/server/local
   linuxhost:/site/linuxhost/local
   linuxhost:/site/linuxhost/u1
```

Notice that binary and home-directory filesystems are mixed freely here. Cfengine determines which of the entries are homedirectories using the `homepattern` variable.

Every time you add a disk or a mountable partition to your network, you should add the partition to the list of mountables.

NOTE: This list is read in order, top down. Cfengine looks for the first filesystem matching a given binary server when expanding the variable `$(binserver)`*, so sometimes the ordering of filesystems matters.*

This list can be accessed in editfiles, to allow straightforward configuration of the automounter, using the command `AutomountDirectResources`.

8.21 processes

Using the processes facility, you can test for the existence of processes, signal (kill) processes and optionally restart them again. Cfengine opens a pipe from the system ps command and searches through the output from this command using regular expressions to match the lines of output from '`ps`'. The regular expression does not have to be an exact match, only a substring of the process line. The form of a process command is

```
processes:

    "quoted regular expression"
                            restart "shell command"
                            useshell=true/false
                            signal=signal name
                            matches=number
                            define=classlist
                            action=signal/do/warn

    SetOptionString "quoted option string"
```

By default, the options sent to ps are "-aux" for BSD systems and "-ef" for system 5. You can use the `SetOptionString` command to redefine the option string. Cfengine assumes only that the first identifiable number on each line is the process identifier for the processes, so you must not choose options for ps which change this basic requirement (this is not a problem in practice). Cfengine reads the output of the ps-command normally only once, and searches through it in memory. The process table is only re-consulted if `SetOptionString` is called. The options have the following meanings:

`signal=`*signal name*

This option defines the name of a signal which is to be sent to all processes matching the quoted regular expression. If this option is omitted, no signal is sent. The signal names have the usual meanings. The full list, with largely standardized meanings, is

```
hup      1   hang-up
int      2   interrupt
quit     3   quit
ill      4   illegal instruction
trap     5   trace trap
iot      6   iot instruction
emt      7   emt instruction
fpe      8   floating point exception
kill     9   kill signal
bus     10   bus error
segv    11   segmentation fault
sys     12   bad argument to system call
pipe    13   write to non existent pipe
alrm    14   alarm clock
term    15   software termination signal
urg     16   urgent condition on I/O channel
```

```
stop     17   stop signal (not from tty)
tstp     18   stop from tty
cont     19   continue
chld     20   to parent on child exit/stop
gttin    21   to readers pgrp upon background tty read
gttou    22   like TTIN for output if (tp->t_local&LTOSTOP)
io       23   input/output possible signal
xcpu     24   exceeded CPU time limit
xfsz     25   exceeded file size limit
vtalrm   26   virtual time alarm
prof     27   profiling time alarm
winch    28   window changed
lost     29   resource lost (eg, record-lock lost)
usr1     30   user defined signal 1
usr2     31   user defined signal 2
```

Note that cfengine will not attempt to signal or restart processes 0 to 3 on any system since such an attempt could bring down the system. The only exception is that the hangup (hup) signal may be sent to process 1 (init) which normally forces init to reread its terminal configuration files.

`restart` "*shell command*"

Note the syntax: there is no equals sign here. If the keyword 'restart' appears, then the next quoted string is interpreted as a shell command which is to be executed after any signals have been sent. This command is only issued if the number of processes matching the specified regular expression is zero. This could be used to restart a daemon for instance. Cfengine executes this command and *waits* for its completion so you should normally only use this feature to execute non-blocking commands, such as daemons which dissociate themselves from the I/O stream and place themselves in the background. Some unices leave a hanging pipe on restart (they never manage to detect the end of file condition). This occurs on POSIX.1 and SVR4 popen calls which use wait4. For some reason they fail to find and end-of-file for an exiting child process and go into a deadlock trying to read from an already dead process. This leaves a zombie behind (the parent daemon process which forked and was supposed to exit) though the child continues. A way around this is to use a wrapper script which prints the line "cfengine-die" to STDOUT after restarting the process. This causes cfengine to close the pipe forcibly and continue. Cfengine places a timeout on the restart process and attempts to clean up zombies, but you should be aware of this possibility.

`useshell=`*true/false*

When restarting processes, cfengine normally uses a shell to interpret and execute the restart command. This has inherent security problems associated with it. If you set this option to true, cfengine executes restart commands without using a shell. This is recommended, but it does mean that you cannot use any shell operators or features in the restart command-line.

`matches=`*number*

This option may be used to set a maximum, minimum or exact number of matches. If cfengine doesn't find a number of matches to the regular expression which is in accordance with this value it signals a warning. The '`<`', '`>`' symbols are used to specify upper and lower limits. For example,

```
matches=<6  # warn number of matches not less than 6
matches=1   # warn if not exactly 1 matching process
matches=>2  # warn if there are fewer than 2 matching processes
```

`define=`*classlist*
: The colon, command or dot separated list of classes becomes activated if the number of regular expression matches is non zero.

`action=`*signal/do/warn*
: The default value of this option is to silently send a signal (if one was defined using the `signal` option) to matching processes. This is equivalent to setting the value of this parameter to '`signal`' or '`do`'. If you set this option to '`warn`', cfengine sends no signal, but prints a message detailing the processes which match the regular expression.

Here is an example script which sends the hang-up signal to cron, forcing it to reread its crontab files:

```
processes:

   "cron" signal=hup
```

Here is a second example which may be used to restart the nameservice on a solaris system:

```
processes:

   solaris::

       "named" signal=kill restart "/usr/sbin/in.named"
```

A more complex match could be used to look for processes belonging to a particular user. Here is a script which kills ftp related processes belonging to a particular user who is known to spend the whole day FTP-ing files:

```
control:

    actionsequence = ( processes )

  #
  # Set a kill signal here for convenience
  #

    sig = ( kill )

  #
  # Better not find that dumpster here!
  #
```

```
     matches = ( 1 )

 processes:

    #
    #  Look for Johnny Mnemonic trying to dump his head, user = jmnemon
    #

    ".*jmnemon.*ftp.*" signal=$(sig) matches=<$(matches) action=$(do)

    # No mercy!
```

The regular expression '.*' matches any number of characters, so this command searches for a line containing both the username and something to do with ftp and sends these processes the kill signal. Further examples may be found in the FAQ section See ⟨undefined⟩ [FAQS and Tips], page ⟨undefined⟩.

8.22 required

This is a synonym for disks, See ⟨undefined⟩ [disks], page ⟨undefined⟩. This action tests for the existence of a file or filesystem. It should be called after all NFS filesystems have been mounted. You may use the special variable `$(binserver)` here.

```
required:

  /filesystem freespace=size-limit define=class-list(,:.)
```

Files or filesystems which you consider to be essential to the operation of the system can be declared as 'required'. Cfengine will warn if such files are not found, or if they look funny.

Suppose you mount your filesystem `/usr/local` via NFS from some binary server. You might want to check that this filesystem is not empty! This might occur if the filesystem was actually *not* mounted as expected, but failed for some reason. It is therefore not enough to check whether the directory `/usr/local` exists, one must also check whether it contains anything sensible.

Cfengine uses two variables: `sensiblesize` and `sensiblecount` to figure out whether a file or filesystem is sensible or not. You can change the default values of these variables (which are 1000 and 2 respectively) in the `control` section. See ⟨undefined⟩ [control], page ⟨undefined⟩.

If a file is smaller than `sensiblesize` or does not exist, it fails the 'required' test. If a directory does not exist, or contains fewer than `sensiblecount` files, then it also fails the test and a warning is issued.

```
required:

   any::

      /$(site)/$(binserver)/local
```

If you set the `freespace` variable to a value (the default units are kilobytes, but you may specify bytes or megabytes), e.g.

```
required:

  /site/host/home1 freespace=50mb define=dotidy
```

then cfengine will warn when the filesystem concerned has less than this amount of free space. By adding a `define` tag, you can switch on any number of classes if this happens. This allows you to activate special measures for dealing with a filesystem which is in danger of becoming full.

8.23 resolve

The file `/etc/resolv.conf` specifies the default nameserver for each host, as well as the local domain name. This file can also contain other information, but these are the only two things cfengine currently cares about. In specifying nameservers you should use the dotted numerical form of the IP addresses since your system may not understand the text form if it is not correctly configured. You may list as many nameservers as you wish, with the default server at the top of the list. The resolver normally ignores entries if you add more than three. The statement:

```
resolve:

  mygroup::

     129.240.22.35
     129.240.22.222
     129.240.2.3
```

declares a list of nameservers for hosts in the group or class `mygroup`. When you add the `resolve` command to the `actionsequence`, this declaration together with the `domain` variable (set here to `uio.no`) results in a `/etc/resolv.conf` file of the form:

```
domain uio.no
nameserver 129.240.22.35
nameserver 129.240.22.222
nameserver 129.240.2.3
```

Note that the `resolve` action does not delete anything from the file `/etc/resolv.conf`. It adds nameservers which do not previously exist and reorders the lines of servers which do exist.

As of version 1.3.11, you may use a quoted string to add non-nameserver lines to this file. For example:

```
resolve:

  mygroup::

     129.240.22.35
     129.240.22.222
     "# Comment line"
     "order bind, files"
```

If the line begins with a non-numeric character, the word 'nameserver' is not added to the line.

8.24 shellcommands

Cfengine focuses on fairly simple minded tasks in order to be as general as possible. In many cases you will therefore want to write a script to do something special on your system. You can still take advantage of the classes you have defined by executing these scripts or shell commands from this section.

The syntax is simply to quote the command you wish to be executed.

```
shellcommands:

  "command-string"  timeout=seconds useshell=true/false
```

If you set the optional timeout string, then cfengine will abort the shellcommand if it exceeds the specified time-limit in seconds. This can be useful for avoiding situations caused by hung network connections etc.

Variable substitution works within the strings. Here are some examples.

```
shellcommands:

   sun4::

       "/usr/lib/find/updatedb"

   AllHomeServers.Sunday::

       "/dir/noseyparker /$(site)/$(host)/u1 $(sysadm) nomail"

   AllBinaryServers.sun4.Saturday::

      "/usr/etc/catman -w -M /usr/local/man"
      "/usr/etc/catman -w -M /usr/local/X11R5/man"
      "/usr/etc/catman -w -M /usr/man"
      "/usr/etc/catman -w -M /usr/local/gnu/man"
```

Some scripts, such as `noseyparker` and a user-backup script, are included in the distribution to help you. See ⟨undefined⟩ [Using the help scripts], page ⟨undefined⟩.

If you need to write more advanced scripts, which make detailed use of the classes defined by cfengine, use the `$(allclasses)` variable to send a complete list of classes to your script in the format

```
CFALLCLASSES=class1:class2:class3...
```

This variable is kept up-to-date at any given time with only the classes which are defined. The command line option '`-u`' or '`--use-env`' can be used to define an environment variable which will be inherited by all scripts and contains the same information. This is not the standard approach, since some systems cannot cope with this rapid change of environment and generate a Bus Error.

Commands can be iterated over variable lists, provided there is at least one space between each variable. For example:

```
control:

      actionsequence =
         (
         shellcommands
         )

 var1 = ( a:b:c )
 var2 = ( x:y:z )

shellcommands:

   "/bin/echo $(var1) $(var2)"
```

This iterates over all values of the list variables See ⟨undefined⟩ [Iterating over lists], page ⟨undefined⟩. If you are iterating over a list, the timeout applies to each separate iteration, not to the sum of all the iterations.

8.25 tidy

The tidy function is used to delete (remove permanently) unwanted files from a system. It is useful for tidying up in `/tmp` or cleaning out `core` files from users' home directories. The form of an entry is:

```
tidy:

   class::

       /directory
                        pattern=wildcard
                        recurse=number/inf
                        age=days
                        size=number/empty
                        type=ctime/mtime/atime
                        dirlinks=keep/tidy/delete
                        rmdirs=true/false
                        links=stop/keep/traverse/tidy
                        define=classlist
```

Note that, each of the options below can be written in either upper or lower case and abbreviated by any unique abbreviation.

/directory This is the directory name to directories which mark the start of a search for files matching certain wildcards. The wildcard `home` may be used instead of an explicit directory, in which case cfengine iterates over all home directories. It is compulsory to specify a directory.

`pattern=`*wildcard*
A wildcard or filename to match the files you wish to be deleted. The pattern may contain the special symbols '`?`' which matches a single character and '`*`' which matches any number of characters as in the shell.

`recurse=`*number/inf*
This specifier tells cfengine whether or not to recurse into subdirectories. If the value is zero, only the named file or directory is affected. If the value is 1, it will open at most one level of subdirectory and affect the files within this scope. If the value is `inf` then cfengine opens all subdirectories and files beginning from the specified filename.See ⟨undefined⟩ [Recursion], page ⟨undefined⟩.

`age=`*days* The age of a file in days represents a minimum *access* time elapsed before the file will be deleted. In other word a file will be deleted if it has not been accessed for *days* days.

`links=`*stop/traverse/tidy*
Normally cfengine does not descend into subdirectories which are pointed to by symbolic links. If you wish to force it to do so (without using the `-l` command line option) you may give this option the value `true`, or `traverse`, or `follow`. To specify no recursion you set the value `false` or `stop`. Note that the value set here in the cfengine

program *always overrides* the value set by the `-l` command line option, so you can protect certain actions from this command line option by specifying a negative value here. If you specify no value here, the behaviour is determined by what you specify on the command line.

The value `links=tidy` has the same effect as the '`-L`' command line option except that here it may be specified per item rather than globally. Setting this value causes links which point to non-existent files to be deleted. This feature will not work on commands with the '`home`' wildcard feature. If you want to clean up old links you should either user a `files` command or the command line option which sets the tidy feature globally.

`size=`*number/empty*
: The value of this parameter decides the size of files to be deleted. Files larger than this value will be deleted if they also are older than the time specified in `age`. The default size is zero so that any file which gets matched by another critereon is deleted. However, if you want to single out only totally empty files, the `empty` may be used. With this option only empty files, nevery files with anything in them, will be deleted, if older than `age`. By default, the filesizes are in kilobytes, but kilobytes and megabytes may also be specified by appending b,k,m to the numbers. Only the first character after the number is significant so you may write the numbers however it might be convenient, e.g. `14k`, `14kB`, `14kilobytes`, the same as for `disable`.

`type=`*ctime/mtime/atime*
: This value is used to set the type of time comparison made using `age`. The default is to compare access times (atime) or the last time the file was read. A comparison by modification time (mtime) uses the last time the contents of the file was changed. The ctime parameter is the last time the contents, owner or permissions of the file were changed.

`dirlinks=`*keep/tidy/delete*
: This value is used to decide whether cfengine will delete links which point to directories. The default value is to keep the links. Note that, if the travlinks option is switched on, cfengine will not tidy or delete links which point to directories, instead it follows them into the subdirectory.

`rmdirs=`*true/false*
: Normally cfengine will not delete directories. If this option is set to 'true' then cfengine will delete any directories which are *empty*. Non-empty directories will not be touched and no message will be given unless in verbose mode. Note that this option overrides the above option `dirlinks`, so that even links which point to empty directories will be removed.

`define=`*classlist*
: The colon, comma or dot separated list of classes becomes defined if any file matching the specified pattern is deleted.

Take a look at the following example:

```
tidy:

   AllHomeServers::
```

```
        home     pattern=core   R=inf age=0
        home     pattern=*~     R=inf age=7
        home     pattern=#*     R=inf age=30


    any::

        /tmp/    pat=*            R=inf   age=1
        /        pat=core         R=2     age=0
        /etc     pat=hosts.equiv  r=0     age=0
```

In the first example, all hosts in the group `AllHomeServers` iterate a search over all user home directories looking for 'core' files (older than zero days) and `emacs` backup files '`*~`', '`#*`' older than seven days.

The default values for these options are the empty string for the wildcard pattern, zero for the recursion and a specification of the age is compulsory.

When cfengine tidies users' home directories, it keeps a log of all the files it deletes each time it is run. This means that, in case of accidents, the user can see that the file has been deleted and restore it from backup. The log file is called `.cfengine.rm` and it is placed in the home directory of each user. The file is owned by root, but is readable to the user concerned.

8.26 unmount

The unmount function unmounts non-required filesystems and removes the appropriate entry from the filesystem table (`/etc/fstab` or equivalent). The syntax is simply

```
unmount:

   class::

      mounthost:filesystem
```

For example:

```
unmount:

   physics::

      libraryserver:/$(site)/libraryserver/data
```

If the device is busy then the actual unmount will not take place until it becomes free, or the machine is rebooted. This feature should work on AIX systems, in spite of these machines inherent peculiarities in the form of the filesystem table.

9 Cfengine script gallery

Here is a gallery of simple-minded scripts to give you ideas for making your own. The absence of explicit testing in cfengine programs also makes these scripts transparent while offering a higher level of checking for no cost to the programmer. Similar shell scripts with this property would be complex indeed.

9.1 User scripts for tidying old files

Here is an example script for tidying old files in your own login area. If you want a long diagnostic, add the option `-v` to the first line of the script, before `-f`.

```
#!/usr/local/bin/cfengine -f
#
# Tidy
#

control:

   actionsequence =
      (
      tidy
      )


tidy:

      $(HOME)        pat=core   r=inf  age=0
      $(HOME)        pat=*~     r=inf  age=1
      $(HOME)        pat=#*     r=inf  age=7
      $(HOME)/code   pat=*.o    r=inf  age=7
      $(HOME)/tex    pat=*.dvi  r=inf  age=7
      $(HOME)/tex    pat=*.log  r=inf  age=7
      $(HOME)/tex    pat=*.aux  r=inf  age=7

      $(HOME)/ftp    pat=*.zip  r=inf  age=7
```

9.2 Controlled opening of files for friends and colleagues

```
#!/local/gnu/bin/cfengine -f
#
# Open my shared directory for others in my group
#
#


control:
```

```
  actionsequence =
     (
     files
     )

  gr = ( myshare )


files:

      $(HOME)       mode=0755 action=fixdirs r=0
      $(HOME)/share mode=0664 action=fixall  r=inf group=$(gr)
```

In this example, first your home directory is opened for the world, then all files in the subdirectory `share` and subdirectories are opened to the group `myshare`. This script could be made to run from a login/logout script of some kind (either `.login` or `.xsession`) so that any new files would automatically be controlled.

9.3 Root script for emergency disk clearing

A straightforward script could be used to clear space in cases where the disk hits the overflow level. This script tidies the whole system, not just the affected disk.

```
#!/local/gnu/bin/cfengine -f
#
# Emergency tidyup!
#
# (Users read their cfengine.rm files to see what got deleted!)
#

control:

site = ( mysite )

mountpattern = ( $(site)/$(host) )
homepattern   = ( u? )

actionsequence =
   (
   tidy
   )


tidy:

      home            pattern=core   R=inf   age=0
      home            pattern=*~     R=inf   age=0
```

```
      home            pattern=*.dvi  R=inf   age=1
      home            pattern=*.o    R=inf   age=0
      /tmp            pattern=*      R=inf   age=0  # could be risky
      /usr/tmp        pattern=*      R=inf   age=0  #       "

ignore:

     .X11
```

9.4 Script for making links

The following script could be used as part of a software installation procedure. Note that the link types can be made relative to the from-link by using `type=relative` See ⟨undefined⟩ [links], page ⟨undefined⟩.

```
#!/tmp/cfengine -v -f
#
# Simple example script to make links
#

control:

  actionsequence = ( links )

links:

 host::

   /usr/local/bin                  +> /usr/local/lib/soft/bin
   /usr/local/X11/lib/app-defaults +> /usr/local/lib/soft/app-defaults
```

It makes links from every binary file in the packages 'bin' directory to the more standard binary directory `/usr/local/bin`. This avoids having to place another search directory into the users' `path` variable. The second statement links the package's application defaults files (for the X-windows system) to a directory in the `XAPPLRESDIR` search path.

This script provides only one way of making the necessary files available to users. It is not the only solution to the problem.

9.5 Ftp server

This script carries out the necessary for setting up a safe anonymous ftp server on a sun workstation running SunOS4.1.

```
#!/local/gnu/bin/cfengine -f
##############################################################
#
# Cfengine script to set up an outgoing ftp server under
```

```
# SunOS 4.1.*.  Suitable for anonymous access.
#
################################################################

control:

 addclasses = ( local global )

 actionsequence =
    (
    editfiles.global
    directories
    shellcommands
    files
    editfiles.local
    )

 ftp_root = ( /oih/saga/local/ftp )   # macro for convenience
 ftp_id   = ( 99 )                    # uid/gid for ftp

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

editfiles:

 # Note the file /etc/ftpusers can contain a list of users
 # who can NOT use ftp to access files.

 global::

 { /etc/passwd

 AppendIfNoSuchLine "ftp:*:$(ftp_id):$(ftp_id): (line continues)
Anonymous ftp:$(ftp_root):/usr/ucb/ftp"
 }

 { /etc/group

 AppendIfNoSuchLine "ftp:*:$(ftp_id):"
 }

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

directories:

  $(ftp_root)           mode=0555 owner=ftp
  $(ftp_root)/pub       mode=0555 owner=ftp
  $(ftp_root)/bin       mode=0555 owner=root
  $(ftp_root)/usr       mode=0555 owner=root
```

```
  $(ftp_root)/dev       mode=0555 owner=root
  $(ftp_root)/etc       mode=0555 owner=root
  $(ftp_root)/dev       mode=0555 owner=root
  $(ftp_root)/usr/lib   mode=0555 owner=root

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

shellcommands:

  "/bin/cp /bin/ls $(ftp_root)/bin/ls"
  "/bin/cp /lib/libc.so.1.8* $(ftp_root)/usr/lib"
  "/bin/cp /usr/lib/ld.so  $(ftp_root)/usr/lib"
  "/bin/cp /usr/lib/libdl.so.1.0 $(ftp_root)/usr/lib/libdl.so.1.0"
  "/usr/etc/mknod $(ftp_root)/dev/zero c 3 12 > /dev/null 2>&1"

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

files:

 $(ftp_root)/bin/ls     mode=111 owner=root action=fixall
 $(ftp_root)/usr/lib    mode=555 owner=root action=fixall r=1
 $(ftp_root)/etc/passwd mode=444 owner=root action=touch
 $(ftp_root)/etc/group  mode=444 owner=root action=touch
 $(ftp_root)/pub        mode=644 owner=root action=fixall

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

editfiles:

 local::

 { $(ftp_root)/etc/passwd

 AppendIfNoSuchLine "ftp:*:$(ftp_id):$(ftp_id): (line continues)
Anonymous ftp:$(ftp_root):/usr/ucb/ftp"
 }

 { $(ftp_root)/etc/group

 AppendIfNoSuchLine "ftp:*:$(ftp_id):"
 }
```

10 Problem solving, bugs, FAQs and tips

10.1 'cf.preconf' bootstrap file

In some cases you will want to run cfengine on a system to configure it from scratch. If the system is in a very bad way, it might not even be able to parse the cfengine configuration file, perhaps because the network was not properly configured or the DNS (Domain Name Service) was out of action. To help prevent this situation, cfengine looks for a script called `cf.preconf` which gets executed prior to parsing and can be used to perform any emergency tests. This file needs only contain enough to get the system to parse the configuration files.

`cf.preconf` may be any script in any language. It need not exist at all! It is fed one argument by cfengine, namely the system hard-class for the current system (e.g. `ultrix`). Here is an example:

```
#!/bin/sh
#
# cf.preconf is an emergency/bootstrap file to get things going
# in case cfengine is unable to parse its config file
#

backupdir=/iu/nexus/local/iu/etc

 #
 # If these files don't exist, you might not be able to parse cfengine.conf
 #

if [ ! -s /etc/resolv.conf ]; then

 echo Patching basics resolv.conf file
 cat > /etc/resolv.conf << XX
domain iu.hioslo.no
nameserver 128.39.89.10
XX

fi

#
# SVR4
#

if [ "$1" = "solaris" ]; then

  if [ ! -s "/etc/passwd" ]; then

  echo Patching missing passwd file
    /bin/cp $backupdir/passwd /etc/passwd
  fi
```

```
  if [ ! -s "/etc/shadow" ]; then

   echo Patching missing passwd file
   /bin/cp $backupdir/shadow /etc/shadow
  fi
fi

#
# BSD 4.3
#

if [ "$1" = "linux" ]; then

   if [ ! -s "/etc/passwd"  ]
   then

    echo Patching missing passwd file
    /bin/cp $backupdir/passwd.linux /etc/passwd
   fi
fi
```

10.2 'cfrc' resource file

If, for some reason you are not satisfied with the defaults which cfengine uses, then you can change them by making an entry in the resource file. The default values are defined in the source code file `classes.c` in the distribution. The format of the resource file is:

```
hardclass.variable: value
```

For example, you might want to forget about where your HPUX system mounts its mail directory and mount it under `/usr/spool/mail`. In this case you would add the line:

```
hpux.maildir: /usr/spool/mail
```

To redefine the filesystem table for GNU/linux, you would write:

```
linux.fstab: /etc/linuxfstab
```

The full list of re-definable resources is:

```
   mountcomm        # command used to mount filesystems
   unmountcomm      # command used to unmount filesystems
   ethernet         # name of the ethernet device
   mountopts        # options to above mount command
   fstab            # the name of the filesystemtable
   maildir          # the location of the mail directory
   netstat          # the full path to netstat and options
   pscomm           # the path to the system's ps command
   psopts           # the options used by ps (default aux/ef)
```

You should never need to redefine resources unless you decide to do something non-standard. Interested readers are referred to the values in `classes.c`.

Cfengine is easily extensible so as to support a variety of architectures. You can even add your own. To do so you need, first of all, to define a new class for the operating system concerned. The file *classes.c* has been separated off from the remainder of the source code so that you can easily see which data structures need to be extended.

To make life as straightforward as possible, three unused classes have been defined. They are called (unremarkably) *unused1*, *unused2* and *unused3*. If you add any further classes, it will be necessary to increase the constant *clssattr* defined in *cf.defs.h* by one for every new addition. You do not need to change *clssattr* if you simple replace one of the unused classes by a real class.

To see fully the impact of what you need to do, you should make a search for the strings *unused?* in all of the source files. Certain special cases need to be handled for each operating system. For example, the form of the filesystem table is quite radically different on some systems such as AIX. One thing you must do is to fill in the default values for the new operating system in the file *classes.c*.

If you fill in the details for a new operating system before it finds its way into a new release, you might consider sending the details to the bug list in the next paragraph.

10.3 Problems with compilation and installation

Although every effort has been made to make the compilation of cfengine trouble free, you might still encounter some problems where non-standard features are concerned. The differences between systems is still a major headache.

Earlier versions of the GNU/Linux operating system do not have support for some of the facilities which cfengine uses. In particular, the ability to use NIS netgroups is absent from earlier versions. During the installation procedure, the `configure` script tests for this possibility and advises you if the facility cannot be used. You can still use cfengine in this case but netgroups will not be expanded. Another problem with GNU/Linux concerns a special socket call to the TCP/IP network interface. This is a command which configures the static routing table and appears to be absent from all versions of Linux and newer IRIX versions. There are also problems with NetBSD. These features are undocumented and will be fixed as soon as they have been understood! If you are running in verbose mode a warning message is printed, otherwise cfengine will ignore attempts to set a default route on the system.

A number of users have experienced a problem using flex and bison in place of lex and yacc. There appears to be a bug in one of these programs which causes cfengine to compile correctly but misinterpret its configuration files, generating an error of the form

```
cfengine:10:action contains invalid statement
```

for every line! The cure is to collect the latest versions of flex and bison from your nearest GNU site.

On really old systems, the configure program is not able to guess what kind of system you are working on. This is true of SunOS versions 4.0.* and also of BSD 4.3 systems. In such cases, you might be able to compile cfengine by using the autoconf option 'host' to specify the host-type.

```
configure --host=sparc-sun-sunos4.0
```

Some other systems which will compile if forced are:

```
m68k-hp-bsd4.3
?-?-bsd4.3
romp-ibm-aos
?-?-aos
```

On some systems, problems arise when using flex. Flex might generate a lexer file lex.yy.c which defines malloc or some other function to be of a type which conflicts with the system definition. If you obtain such a culture crash, edit the lexer file manually and simply delete the offending definitions, then run make again.

As of version 1.4.0 cfengine tries to link in features based on the Berkeley database library '`libdb`' and the TCP wrappers library '`libwrap`'. If you want to use these facilities, you will have to collect them and install them before compiling cfengine. Some problems have been experienced with the linux version of TCP wrappers. If you experience compilation problems, the best thing to do is to edit '`src/conf.h`' after configuration and remove the line beginning '`#define HAVE_LIBWRAP`'.

Newer solaris systems have ACLs. The ACL features only matured in version 2.5 of solaris however, and there have been some problems with the partial implementation in 2.4. If you obtain error messages about unknown ACL functions, edit the '`config.cache`' file in the cfengine root directory and set the value:

```
ac_cv_header_sys_acl_h=${ac_cv_header_sys_acl_h='no'}
```

If you use the DCE (Distributed computing environment) cfengine will try to compile the ACL extension for DFS. This requires the DCE library to be present on the system on which you are compiling. On some systems it also requires thread libraries to be present. Unfortunately, the autoconf program which generates the Makefiles cannot detect shared libraries, only archive libraries. This means that you need to edit the '`config.cache`' file to compile in this support. Set the following values:

```
ac_cv_lib_dce_main=${ac_cv_lib_dce_main='yes'}
ac_cv_lib_dce_main=${ac_cv_lib_thread_main='yes'}
```

Finally, although the autoconfiguration program appends the same libraries to each executable, the following libraries are required only by the following programs.

```
cfengine   -ldce -lthread -lm

cfd        -ldb -lpthread
```

10.4 Bug reports and suggestions

If you experience a problem with cfengine, find a bug or have another suggestion which you wish to air, you can send your thoughts to the special mail address `bug-cfengine@prep.ai.mit.edu`.

Always think a bit before sending a message to the list. This helps to keep down the traffic improves the signal to noise ratio of your thoughts! Try to solve the problem yourself first and look particularly to see whether your system is clean or whether you have installed software or patches which might conflict with cfengine (I can't really imagine how this would happen—but it might).

Always be clear about what type of operating system you are running and whether or not it is a complete installation.

Some vendors have begun the practice of distributing systems without key programs like the C compiler, lex and yacc. If you have this problem, you can pick up GNU replacements gcc, flex and bison from any GNU site.

10.5 FAQs and Tips

Here is a problem solver: an encyclopaedia of suggestions and uses for cfengine as accumulated over the years. If you have a contribution to make, please send it to `cfengine@iu.hioslo.no`. Format your submission like this:

Q: *How do I do....*

A: Very well thank-you....

The table below is updated as the tips occur to me, or as others contribute their own. Please note that any focusing on particular operating systems is purely a matter of personal usage/experience and should not be interpreted as a reflection of how many 'bugs' these systems may or may not contain.

10.5.1 General

Q: How can I check to see what cfengine will do without going through the whole program using '`-n`'?

A: Run cfengine with options:

```
cfengine -p -d3
```

This just parses the file and dumps the contents of the parser to the output.

Q: *Why doesn't cfengine have classes for each hour, instead of just for days?*

A: It does from version 1.3.20 and upward. The hours are denoted in 24 hour clock notation by `Hr00`—`Hr23`. Other time classes are also possible See ⟨undefined⟩ [Using cfengine as a front-end for cron], page ⟨undefined⟩.

Q: *How can I replace the stupid version of sendmail my vendor ships with my OS with, say, Berkeley sendmail?*

A: First of all, compile your new sendmail in a filesystem which is held separate from the OS, for example '`/local/mail`'. You can keep all the files under this new file tree. Now you need to replace '`/usr/lib/sendmail`' with the new version and '`/etc/sendmail.cf`' or '`/etc/mail/sendmail`' with the new files, so that the system can find them.

```
links:
   /usr/lib/sendmail ->! /local/mail/bin/sendmail
   /etc/sendmail.cf  ->! /local/mail/etc/sendmail.cf
```

Q: *How can I prevent big log-files like '`/var/adm/wtmpx`' and '`httpd/access_log`' from filling up my partitions?*

A: Add a line to disable the files once a week. That way you still get a chance to look at them, but you keep the size down:

```
disable::

   Sunday::

      #
      # Do this to throw away old entries
      #

      /var/adm/wtmpx rotate=truncate

      #
      # Or this to keep the last lot
      #

      /var/adm/wtmpx rotate=1
```

An alternative to using `disable` would be to use `tidy`, but then you lose the file once and for all. Note though, that '`wtmpx`' gets updated all the time, so an age `age=0` is necessary to have any effect at all. Some daemons, like '`httpd`', lose their ability to write to a log file if you rename and create a new file. The `rotate` feature in cfengine preserves the open file handle, fixing this problem.

Q: *How can I fix exports in cfengine?*

A: This is a complicated matter. There are lots of ways to do it. The key is either to edit the file '`/etc/exports`' ('`/etc/dfs/dfstab`' in solaris), or to execute an export (share) command directly from `shellcommands`. Under Solaris 2 this is quite easy owing to the fact that the file '`dfstab`' is just a script itself, rather than a configuration file like the old '`/etc/exports`' file. Since editing is limited and you need to specify a list of hosts which might change in time, one of the following is probably the best bet:

```
shellcommands:

   solaris::

      "/usr/sbin/share -F nfs -o rw=netgroup /var/mail"
```

On non-solaris systems:

```
editfiles:

   { /etc/exports
```

```
    AppendIfNoSuchLine "/site/host/fs -access=netgroup"
    }
```

Q: *How can I distribute key setup files to users and keep them up to date?*

A: The copy facility will distribute to all users if you use the `home` directive. For instance, to copy a basic '`.cshrc`' file or '`.xsession`', you could write

```
copy:

   /local/masterfiles/.cshrc     dest=home/.cshrc
   /local/masterfiles/.xsession  dest=home/.xsession
```

Q: *Some users set up their own IRC listen services called* "eggdrop" *which fill up the disk with all kinds of garbage. How can I kill all these processes?*

A:

```
processes:

  #
  # Most users
  #

  "eggdrop"  signal=kill

  #
  # One wise-guy has renamed the daemon!
  #

  ".*wiseguy.*myegg.*"  signal=kill
```

Q: *My license server keeps crashing! How can I check that it's ok?*

A:

```
processes:

  #
  # BSD - often need long descriptive lines
  #       to find this daemon
  #

  SetOptionString "-ax"

  # Exactly one should be running

  "lmgrd" matches=1
```

Q: *I want to use cfengine to keep DNS tables up to date, using editfiles. How can I make cfengine automatically restart the name server after the edits?*

A: This can be done in two ways. Probably, you need to update a serial number as well as restarting the daemon. You might use a Makefile to simplify this.

```
control:

  actionsequence = ( editfiles control )

  solaris::
           named = ( /usr/sbin/in.named)
  linux:
  freebsd:
           named = ( /usr/sbin/named )
  sun4:
           named = ( /usr/etc/named )

editfiles:

 # edit files here

shellcommands:

   #
   # If you use make to sort out the details
   #

  "/local/gnu/bin/make -f /local/named/Makefile > /dev/null"
```

Or if you need to explicitly restart the name daemon, you could supplement the above with an explicit restart command (this means you lose the cache),

```
processes:

  "named" signal=kill restart "$(named)"
```

Q: *How can I edit all users' login files?*

A: You can use the 'home' pseudo-variable to iterate over all users' homedirectories:

```
editfiles:

    { home/.cshrc

    # Local fixes
```

```
AppendIfNoSuchLine "alias lp  special-print-command"

# Security

DeleteLinesMatching "xhost +"
}
```

Q: How can I kill all processes except for root processes?

A: The following regular expression matches lines which do not contain the string root:

```
processes:

 "\(root\)\{0\}"  signal=term # or kill
```

Q: How can I make cfengine distribute my '/etc/motd' file?

A: You will need a master file which contains the text you want to put on your servers. Let us define a variable 'masterfile' which contains this. This master file needs to be available on all hosts on a common NFS filesystem, for instance. (This will change when remote copying is implemented in cfengine.) Now you can do something like the following script. Note that we define a version number for motd which just prevents cfengine from editing the file every single time. You have to change this version number yourself in the config file to force an update. If you don't care about this, just leave out the Begin..End parentheses.

```
control:

masterfile = ( /usr/local/admin/motd-master )

editfiles:

  any::

     { /etc/motd

     BeginGroupIfFileIsNewer "$(masterfile)"
       EmptyEntireFilePlease
       InsertFile "$(masterfile)"
       PrependIfNoSuchLine "This system is running $(class):$(arch)"
       AppendIfNoSuchLine "$(motd_version)"
     EndGroup
     }
```

Note that, if you want special messages added just for, say, linux, then you can single out linux using a special class, or add a special edit after this one.

Note, if you want to keep the first kernel line in this file, you can change this to:

```
editfiles:

   any::

   { /etc/motd

   BeginGroupIfFileIsNewer "$(masterfile)"
     IncrementPointer "1"
     DeleteLinesAfterThisMatching ".*"
     InsertFile "$(masterfile)"
     AppendIfNoSuchLine "$(motd_version)"
   EndGroup
   }
```

bug-cfengine exchange: (Reply courtesy of David Masterson).

I like cfengine a lot and it helps me very much, but I am a little concerned about security. I'm using cfengine to keep some files like /etc/hosts /etc/printcap /etc/mount etc. up to date. So cfengine is started by root in a cron job and reads its cfengine.conf file and all the other information from a filesystem which is common to all the systems. If now somebody manage to alter the cfengine.conf file he can do everything he wants. Wouldn't it be a good idea to make the cfengine.conf file something like a pgp signed messages, so that cfengine can test if this file was created by the right person? Or are there other tips to make it more secure?

I'm not sure, but I think you're over-reacting or you need to be more specific about where you think the holes are in Cfengine's security. If you follow the tips of any standard systems administrator using cfengine or not, there should be few issues concerning security (ie. if security broke, there would be little chance that cfengine could do anything about it anyway).

Ask yourself some of the standard questions with respect to security on UNIX:

- Who owns your script files?
- Who can update those script files?
- If those script files are updated by the wrong person, has your security been purposely broken or does it just have a hole in it?

If you're still worried about the security of your script (be it a cfengine script or not), you could always adjust your cron script to "decrypt" the script file before executing it (see crypt(1)).

Personally, I think if you've set the permissions on your script files properly, then, if someone breaks into those scripts, they've already broken into your system to a point where they could do what they wanted anyway.

Q: How can I distribute password files in cfengine, but keep certain passwords different on some machines, like I can with NIS?

A: If you keep a file with special local passwords, you can override the password file using `editfiles`. First you use `copy` to get the distributed file, then you edit the file like this:

```
editfiles:

  { /etc/passwd

  SplitOn ":"


  ForEachLineIn "/usr/local/etc/passwd.local"

     ReplaceLinesMatchingField "1"

  EndLoop
  }
```

This means, if the first field of each line in the files matches in both files (and both files have the same column format) then replace the line in '/etc/passwd' with the line from '/usr/local/etc/passwd.local'.

Q: How can I add entries to a list, like in the fiel '/etc/group'?

A: Okay, suppose you wanted to make sure that a special user was in the group 'adm', you would use a construction like this:

```
control:

    person = ( new-user )


editfiles:

 { /etc/group

 BeginGroupIfNoLineMatching "adm.*$(person).*"
   LocateLineMatching "adm.*"
   AppendToLineIfNotContains ",$(person)"
 EndGroup
 }
```

Q: How can I take backups with cfengine?

A: If you have a spare disk partition, you could make a mirror of the most important files. You would use something like this:

```
control:

     excludecopy = ( *.mp3 *.o *.dvi *.ps *.zip *tar*
                     core a.out *.au *.wav .* *.exe *.tgz )

copy:
```

```
        BackupHost.Hr21::

          /site/host/home dest=/site/host/backup2/u1 r=inf size=<4mb backup=false action=silent
```

for each partition you want to back up.

10.5.2 AIX

Q: *Hints about AIX?*

A: Send then to `bug-cfengine@prep.ai.mit.edu`.

Q: One of our Sysadmins has noted a limitation with line length under AIX. I'm not sure how easy it is to fix but it might be worth noting it somewhere in the cfengine docs. It appears that on the AIX machines the maximum line length we can use for cfengine files is defined by the constant YYLMAX which is set to be 200. On the Suns this constant is set to be the same as BUFSIZ which is currently set to be 1024. This manifested itself by very unusual behavior as cfengine variables began to be overwritten when line lengths in the config file exceeded 200 bytes. Peter can attest to this. Be forewarned "keep line lengths in cfengine less than 200 if you want them to work on AIX machines" Moral of the story "AIX users beware" Do you think we could just recompile cfengine and use larger buffer sizes all over, I don't know if this constant is all that should be tweaked or if it is somehow tied into the lexx implementation also, since lexx is used to create the parser for the config files.

A: This is a problem with lex and yacc, not with cfengine. The variable BUFSIZ is a system quantity, not related to cfengine's internal variable bufsize. I would recommend getting bison and flex and doing away with the old lex and yacc from the system. Michael Lachowski reports that this is also a problem with HPUX 10's lex/yacc.

10.5.3 HPUX

Q: Problems with line length in lex/yacc.

A: See the FAQ for AIX.

Q: *What is the difference between the classes '`hpux`' and '`hpux10`'?*

A: In version 10 of HPUX, the file structure is reorganized to look more like SVR4. If you have an HPUX-10 system, the appropriate hardclass is `hpux10` rather than `hpux`.

Q: *I set up the new sendmail but the configuration file doesn't work.*

A: There could be a frozen configuration file around. Try:

```
disable:

   hpux::

      /usr/lib/sendmail.fc
```

Q: *Why don't groups work in HPUX?*

A: HPUX uses the file '/etc/logingroup' not '/etc/group'. Make a link if you need to:

```
links:

  hpux::

     /etc/logingroup -> /etc/group
```

To encourage some cross-fertilization, here's a sanitized sendmail configuration script that I created for HPUX and HPUX10. (From David Masterson, posted to gnu.cfengine.help). David's script is nice since sendmail was the inspiration for cfengine's name.

```
#!/usr/local/bin/cfengine -f
#####################################################
#
# File:  sendmail.conf
#
# Description:   CFEngine script to setup the sendmail.cf.
#
#####################################################

control:

access = ( root )

# Postmaster
sysadm = ( myPostmaster )

# NIS domain and group server
site = ( myserver )

# DNS domain
domain = ( myDNSdomain )

# our gateway host
gtwyhost = ( mygateway )

# sendmail.cf can be big
editfilesize = ( 1000000 )

actionsequence =
(
copy
files
editfiles
shellcommands
)
```

```
# disable unwanted classes with "--undefine" option
addclasses = ( maildom mailhst )

################
# bindir - location of sendmail
# libdir - location of current mail files
# cfgdir - location of initial mail files
# etcdir - location of hosts.smtp
# own    - who should own result files
# grp    - what group should result files be in
################
    hpux::
bindir = ( /usr/lib )
libdir = ( /usr/lib )
cfgdir = ( /etc/newconfig )
etcdir = ( /etc )
own = ( root )
grp = ( sys )

    hpux10::
bindir = ( /usr/sbin )
libdir = ( /etc/mail )
cfgdir = ( /usr/newconfig/etc/mail )
etcdir = ( /etc )
own = ( root )
grp = ( sys )

# disable with "--no-copy" option
copy:

$(cfgdir)/sendmail.cf dest=$(libdir)/sendmail.cf type=checksum
mode=0644 owner=$(own) group=$(grp) force=true

# checks for other important files
files:

$(libdir)/aliases mode=444 owner=$(own) group=$(grp) action=touch
$(libdir)/rev-aliases mode=444 owner=$(own) group=$(grp) action=touch
$(etcdir)/hosts.smtp mode=444 owner=$(own) group=$(grp) action=touch

# disable with "--no-edit" option
editfiles:

    any::

# setup general part of sendmail.cf
{ $(libdir)/sendmail.cf
```

```
SetCommentStart '#'
SetCommentEnd ''
ResetSearch "1"
UnCommentLinesMatching "#OP.*" # activate Postmaster
ResetSearch "1"
UnCommentLinesMatching "#DY.*"
ResetSearch "1"
LocateLineMatching "DY.*"
ReplaceLineWith "DY$(site).$(domain)" # set site hiding
ResetSearch "1"
UnCommentLinesMatching "#DS.*"
ResetSearch "1"
LocateLineMatching "DS.*"
ReplaceLineWith "DS$(gtwyhost)" # all-knowing SMTP host
# Ruleset 0 setups
ResetSearch "1"
UnCommentLinesMatching "#R.*user@domain to SMTP relay.*"
ResetSearch "1"
LocateLineMatching "# try to connect to any host for user@domain"
IncrementPointer "1"
CommentNLines "1"
}

# add Postmaster alias
{ $(libdir)/aliases

SetLine "Postmaster: $(sysadm)"
AppendIfNoLineMatching "Postmaster.*"
}

# setup processing of local domain hosts
    maildom::

{ $(libdir)/sendmail.cf

SetCommentStart '#'
SetCommentEnd ''
ResetSearch "1"
LocateLineMatching "DL.*"
ReplaceLineWith "DL$(domain)"
# Ruleset 0 setups
ResetSearch "1"
LocateLineMatching "# connect to hosts in local domain"
IncrementPointer "1"
UnCommentNLines "1"
}

# setup processing via class S
```

```
    mailhst::

{ $(libdir)/sendmail.cf

SetCommentStart '#'
SetCommentEnd ''
ResetSearch "1"
UnCommentLinesMatching "#FS.*"
# Ruleset 0 setups
ResetSearch "1"
LocateLineMatching "# connect to hosts in class S"
IncrementPointer "1"
UnCommentNLines "1"
}

# setup of list of hosts for class S
{ $(etcdir)/hosts.smtp

EmptyEntireFilePlease
Append "localhost1"
Append "localhost2"
}

# disable with "--no-commands" option
shellcommands:

"$(bindir)/sendmail -bk"
"$(bindir)/sendmail -bi"
"$(bindir)/sendmail -bz"
"$(bindir)/sendmail -bd"

################
# End of File
################
```

10.5.4 IRIX

Q: *Hints about IRIX?*

A: Send them to `bug-cfengine@prep.ai.mit.edu`.

10.5.5 LINUX

Q: When I try to compile `cfd` I get this error

```
/usr/lib/libwrap.a(options.o): In function `twist_option':
options.o(.text+0x5f7): undefined reference to `deny_severity'
/usr/lib/libwrap.a(options.o): In function `severity_option':
options.o(.text+0x808): undefined reference to `deny_severity'
```

```
options.o(.text+0x81c): undefined reference to ‘deny_severity’
options.o(.text+0x821): undefined reference to ‘deny_severity’
options.o(.text+0x826): undefined reference to ‘deny_severity’
options.o(.text+0x82b): undefined reference to ‘allow_severity’
make[1]: *** [cfd] Error 1
```

A: There seems to be a problem with the distributed version of the TCP wrappers library. Edit the ‘`src/conf.h`’ file and comment out the ‘`#define HAVE_LIBWRAP 1`’ line. This means that you will not be able to use TCP wrappers security however. You might prefer to collect and compile a new version of TCP wrappers.

Q: I keep getting segmentation fault from my cfengine scripts when I install a completely new machine.

A: There is a bug (apparently in Linux) which means that undefined groups cause a segmentation fault. If a group is not in ‘`/etc/group`’ or in the NIS database, you should only get an error message, but linux generates a core dump. I don’t know of any fix except to edit group file manually before running cfengine, or place a fix in editfiles and have this run before files, directories etc, which make reference to the group.

Q: *Linux insists on rebuilding the message of the day file each time it boots, but that means I keep losing the messages I leave there.*

A: Add the following to your configuration files to comment out the offending lines in the startup scripts:

```
editfiles:

   linux::

    { /etc/rc.d/rc.S

    HashCommentLinesContaining "motd"
    }
```

10.5.6 OSF

Q: Hints about OSF/1?

A: Send them to `bug-cfengine@prep.ai.mit.edu`.

10.5.7 SUNOS (4.1.*)

Q: *How can I delete the ‘`+`’ sign from the ‘`/etc/hosts.equiv`’ file to improve security?*

A: Use editfiles to delete it:

```
editfiles:

   sun4::

      { /etc/hosts.equiv

      DeleteLinesMatching "+"
      }
```

10.5.8 SOLARIS 2

Q: *I keep getting a 'bad address' error when cfengine tries to reset the netmask and broadcast address.*

A: This is a bug in the sockets library on solaris. It is supposed to be fixed in solaris 2.5.

Q: *How can I add my own file 'rc.local' to the startup bootfiles automatically?*

A: For example, create a file called '`/local/etc/rc.local`' which looks something like this:

```
#
# rc.local
#
PATH=/local/gnu/bin:/bin:/usr/bin:/usr/sbin; export PATH

#!/bin/sh

if [ "`hostname`" = "net-server" ]; then

   echo Starting WWW server
   /local/httpd_1.4/httpd -d /local/httpd_1.4

   echo Starting GNU finger server
   /local/etc/fingerd

fi

echo Starting ypbind
/usr/lib/netsvc/yp/ypbind

echo Adding a default route and flushing table

route -f add default my-gateway 1

echo Starting xdm

/local/bin/start-xdm
```

Now add an entry to your '`cfengine.conf`' file like this

```
solaris::

   { /etc/rc3.d/S99rc-local

   AutoCreate
   AppendIfNoSuchLine "exec sh /local/etc/rc.local"
   }
```

Q: *The solaris installation program creates '`/tmp`' without the sticky bit set, so that any user can delete any files in '`/tmp`'. It also means that a race condition can occur in the kernel which can give away root access to any user!*

A:

Add the following line to the configuration immediately!

```
files:

   /tmp mode=1777 action=fixdirs
```

Q: *The ftp program will not allow me to log in to my own account!*

A: The problem is that your shell is not in the system file '`/etc/shells`'. Add a line something like this:

```
editfiles:

   { /etc/shells

   AppendIfNoSuchLine "/local/bin/tcsh"
   }
```

Q: `tcsh` *prints an error message on startup and will not read my '*`.cshrc`*' file.*

A: The problem is the central login file distributed with solaris. '`tcsh`' can't understand it. Add a line

```
disable:

    /etc/.login type=file
```

You might want to replace this with a link to your own file.

Q: *Why does solaris fill up the routing table with hundreds of addresses under the loop-back interface? (see netstat -r)*

A: First of all, get the latest patches for solaris, there are bugs in the kernel of solaris 2.4 which makes this worse. Second, make sure you have a file '`/etc/defaultrouter`' with the IP address of your local gateway, if you don't intend to run your system as a router. For instance:

```
files:

  solaris::

     /etc/defaultrouter o=root g=other m=644 act=touch

editfiles:

   solaris::

      { /etc/defaultrouter

      AppendIfNoSuchLine "xxx.xxx.xxx.1"
      }
```

where `xxx.xxx.xxx.1` is the IP address of your gateway.

Q: When trying to boot the system, solaris fails with the error message: *fork: rescource temporarily unavailable/vfork failed*. The system then claims that there is something wrong with one of the file systems.

A: The file '`/etc/system`' has probably been corrupted. If this file does not exist, solaris establish the kernel properly and will not fork any processes. Things usually die early on in the boot process. This causes the side effect that the first fork the system needs to perform (to check the disk file systems) fails and misinterprets the reason for failure of the command. This makes it look as though something is wrong with the disks. Add a line:

```
files:

   /etc/system o=root g=root m=0644 action=touch
```

Q:

I am currently involved with setting up machines with jumpstart. Jumpstart as you may know allows handsfree installation of solaris. One of the things it allows you to do is specify a "finish" script. I am running cfengine from this script to do the bulk of the configuration. During jumpstart, the root of the machine you are installing is actually under "/a". This leads to problems with cfengine with LOCKFILEDIR and LOGFILEDIR at the very least. It would cause problems with all assumptions cfengine makes about system files too. What would be execeedingly nice would be a command line option to redefine where root is assumed to be. I realize this would be pretty hairy with respect to mounting through cfengine, but it would be very useful. For file editing and such a root prefix macro woudlprobably work ok. Let me know what you think.

A:

Define the filenames

```
$(root)/filename
```

and set `$(root)` to `""` or `"/a"` depending on context? That way you could the above without screwing up other things which might be needed. You can switch off the locks with `-K`. And you could override the '`vfstab`' location for solaris in the resource file.

10.5.9 FreeBSD

Q: *How can I stop my FreeBSD system from running the '`/etc/daily`' script which mails me every single day, week and month?*

A: Add an editfiles command

```
freebsd::

  { /etc/crontab

  HashCommentLinesContaining "daily"
  HashCommentLinesContaining "weekly"
  HashCommentLinesContaining "monthly"
  }
```

Q: Why don't filesystems get mounted in the freebsd version of cfengine?

A: Cfengine fixes the '`/etc/fstab`' file, but has to choose between one of two courses of action when mounting, owing to a bug in the mount command on FreeBSD machines. Cfengine mounts filesystems each time it runs. On all other supported systems this causes no problems,: once a filesystem is mounted it will not be mounted again. Under FreeBSD however, a filesystem gets mounted again each time mount is run, leading to multiple mount information in the mount table. This causes cfengine to warn the filesystem is mounted many times, and could eventually result in a problem for the FreeBSD machine. The policy is therefore to use mount options which do not cause this behaviour, but an unfortunate side-effect is that newly defined filesystems do not get mounted. You can override the mount options if you want to force multiple mounting.

11 Using the help scripts

The following Perl scripts are included as examples and helpful tools in your system administration package. If you do not have Perl, you should get it — it is a very useful language for system administration.

11.1 cfwrap

It is useful to run cfengine on a daily basis from a `cron` script. Use a line like the following one to start cfengine each night. (Note the curiosities of older BSD cronfiles).

```
0 0 * * * /usr/local/lib/cfengine-3.0/bin/cfwrap cfdaily
```

where *cfdaily* is a script which looks something like

```
#!/bin/sh

CFINPUTS=/usr/lib/cfengine/inputs

/usr/local/bin/cfengine
```

You will need to include full path names to the scripts in the cron file. The syntax for using `cfwrap` is as follows.

```
host% cfwrap mycommand

host% cfwrap cfengine

host% cfwrap script_which_sets_CFINPUTS_and_calls_cfengine
```

When you run cfengine it normally only generates output if something is wrong which needs your attention. If you are running cfengine as a `cron` job then the results of each job are normally mailed back to you — or to root. But this causes problems in a networked environment, since mail to root is usually redirected to some central place which local system administrators cannot access. Moreover, you have no way of knowing which host sent the information. The solution is to use a script as a wrapper. The script simply executes some command and collects the output from that command into a file which then gets mailed to some address.

The address to be mailed to is obtained directly from cfengine by calling it with the -a switch. The name of the host running cfengine is prepended to the file before it is sent making it easy to see where each message originated. This is also transferred to the subject header of the mail message. `cfwrap` calls `cfmail` in order to mail the result of the command back to the system administrator.

11.2 cfmail

Because there are no standard mail-agents except for *sendmail*, the wrapper script *cfwrap* calls its own simple mail agent *cfmail* to send the message. Note that the *flags* variable in the script *cfmail* arranges for the mail message to be sent with a return address other than "root". This means that if the recipient of the mail should decide to hit 'r' for 'reply' to reply to the message, you have a chance of getting to see the message before it vanishes along with the rest of the mail to *root* into the same black hole that swallows up all those credit cards, house-keys and odd socks that disappear on a daily basis.

You might have to tweak the scripts slightly to tailor them to your own needs. They are used as follows:

```
host% echo test ....  | cfmail -s "Test message" mark
```

11.3 noseyparker and editquotas: software quotas

Noseyparker is a script which must be run by *root*. It is used to give a software warning about users who are hogging your disk. It is run with the command:

`noseyparker` *homedir* `$(sysadm) [nomail]`

The directory *homedir* should be one of the directories in which user's home directories reside. This is searched for a list of usernames. `$(sysadm)` is the mail address of the system administrator (which can be obtained from the cfengine variable of the same name) and is used to send information about users who have exceeded certain quotas. The option *nomail* prevents *noseyparker* from sending mail automatically to the users concerned.

Although many administrators use the quota checking facilities available in UNIX, there are problems with these. Many users need to generate large temporary files (when generating postscript or TEX, or when performing numerical calculations). Using hard quotas prevents these users from using their accounts effectively. *Noseyparker* takes the view that most users will tidy up files if they receive a polite reminder — and the few who cannot be dealt with by other means.

Noseyparker should be run once a week on each home directory on a homeserver. When you run *noseyparker*, the program generates statistics which it keeps from week to week in order to find out how fast data are growing. If the number of kilobytes for a given user exceeds a limit, a warning is generated: either a *pre-warning*, a *soft warning* or a *hard warning*. If *nomail* is set, only the system administrator gets a list of these users. If it is not set, then each user gets an automatic message telling him or her to tidy up. The 'degree' of the message depends on the extent of the crime. Every user gets at least three 'soft warnings' before putting in the boot. The system administrator can see from the mail report which type of warning each user has received.

A default quota is defined in the variable `$softlimit`. A pre-margin determines how early a pre-warning will be sent. Special quotas for particular users are set using the *editquotas* script

`editquotas` *homedir*

For example, to change the quota of a user whose home directory is on '`/mysite/myhost/u2`', one writes:

```
editquotas /mysite/myhost/u2
```

The file is edited so that it takes the form

username quota
user2 quota2

You can modify *noseyparker* to suit your own setup if necessary.

Here is a typical command to start *noseyparker*:

```
shellcommands:

   homeservers.Sunday::

      "$(cfbin)/noseyparker /home/$(host) $(sysadm) nomail"

   homeservers.cfengine_model.Sunday::

      "$(cfbin)/noseyparker /$(site)/$(host)/u1 $(sysadm) nomail"
```

11.4 cfbackup and cfrestore scripts

Today, most people would agree that diskspace is cheap, whilst time spent taking backups is expensive. The backup script included in the '`bin`' directory solves this problem for us at Oslo College by making automatic backups of users' directories using GNU-tar to compressed tar files which are then placed in a spare partition of a special disk. The script is called up as follows:

```
shellcommands:

  BackupHost.Sunday|BackupHost.Wednesday::

      #
      # Make a system backup of /iu/nexus/u? with Audun's script
      #

      "$(cfbin)/cfbackup -p -f /iu/dax/backup1 -s /iu/nexus/u1"
      "$(cfbin)/cfbackup -p -f /iu/dax/backup1 -s /iu/nexus/u2"
      "$(cfbin)/cfbackup -p -f /iu/dax/backup2 -s /iu/nexus/u3"
      "$(cfbin)/cfbackup -p -f /iu/dax/backup2 -s /iu/nexus/u4"
```

The directories to be taken backup of are listed using the `-s` option. The directory which the files get saved to is coded in the variable `$destination` of the (Perl) backup script. `-f` tells the script where to leave the tar-ed backup file: in this case the user partitions are spread between two backup areas. The `-p` option makes sure that each user owns his/her own backup file. This means that each user can access their backup files themselves using the `cfrestore` script (see below).

As the backup files build up on the backup disk, you can delete older files using the tidy function:

```
tidy:

   BackupHost::

      # Here we tidy old backup tar files from the backup area

      /iu/dax/backup1      pat=*  age=15
      /iu/dax/backup2      pat=*  age=15
```

This example would tidy all the old backupfiles after three weeks.

Each user's directory (assumed to be a child of the directories named using `-s`) is saved in a separate tar-file, labelled with the date on which the backup was taken. They look like this:

```
backup.16.8.95.iu.nexus.u2.wiikl
backup.16.8.95.iu.nexus.u2.wullumt
backup.16.8.95.iu.nexus.u2.wux
backup.20.8.95.iu.nexus.u1.agneta
```

A file can then be recovered by typing a command like

```
cfrestore myfile
```

`cfrestore` will then search through the tar-file for files matching the file you request and place them in a special directory under your home directory, labelled by the name (and date) of the tar file from which the file was extracted. If you know which backup file (the date) from which you want to restore a file, you can use the `-f` *backupfile* option.

These scripts will at the very least require minor modifications for a your local site configuration, but they can be used as a simple and effective way of taking backups automatically.

NOTE: the scripts make convenient use of GNU tar and fileutils packages.

11.5 cfbg

Normally cfengine blocks in subprocesses. That means that when you execute a shell command, cfengine waits for each command to exit before continuing execution. This serializes the execution of shellcommands. On occasions, you might wish to force a process to dissociate from parent cfengine process and run in parallel. For example, you might want the backup scripts (which can take quite a long time) to not hold up cfengine. To do this, you can use a simple wrapper script like '`cfbg`'. This command simply takes the remainder of the command line (after cfbg) as a command to execute and forks a new process for this command, without waiting. Cfengine is then free to continue running. Here is an example

```
control:

  cfbin = ( /local/gnu/lib/cfengine/bin )

shellcommands:

  "$(cfbin)/cfbg $(cfbin)/cfbackup"
```

The default script writes the output of the standard out and standard error to a file in the '`/tmp`' directory. Alternatively you might wish to send the output directly to '`/dev/null`'.

12 Example configuration files

Here is a sample from a large configuration file, just to give you some ideas. The file is broken up into manageable pieces for convenience.

12.1 cfengine.conf

```
######################################################################
#
#  CFENGINE CONFIGURATION FOR site = iu.hioslo.no
#
#  This file is for root only.
#
#######################################################################

###
#
# BEGIN cfengine.conf
#
###

import:

   #
   # Split things up to keep things tidy
   #

   any::
                    cf.groups
                    cf.main
                    cf.site
                    cf.motd

   hpux::           cf.hpux
   linux::          cf.linux
   solaris::        cf.solaris
   sun4::           cf.sun4
   ultrix::         cf.ultrix
   freebsd::        cf.freebsd

   #
   # Do you want to do this ?
   #

   AllHomeServers:: cf.users


###
```

```
#
# END cfengine.conf
#
###
```

12.2 cf.groups

```
##############################################################
#
# cf.groups - for iu.hioslo.no
#
# This file contains  all group/class definitions
#
#################################################################

###
#
# BEGIN cf.groups
#
###

groups:

   #
   # Define some groups
   #

   iu = ( nexus ferengi regula borg dax lore axis worf daystrom voyager
          aud1 aud2 aud3 aud4 bajor ds9 takpah takpeh nostromo galron
          thistledown rama chaos pc-steinarj pc-hildeh way jart kosh )

   diskless   = ( regula ferengi lore )

   standalone = ( nexus axis dax borg worf daystrom voyager
                  aud1 aud2 aud3 aud4 bajor ds9 takpah takpeh
                  nostromo galron thistledown rama pc-torejo
                  pc-steinarj pc-hildeh )

   AllHomeServers   = ( nexus )
   AllBinaryServers = ( nexus borg )

   XBootServer  = ( nexus )
   WWWServers   = ( nexus )
   FTPserver    = ( nexus )
   NameServers  = ( nexus )
   PasswdServer = ( nexus )
   BackupHost   = ( nexus )
```

```
   MailHub      = ( nexus )
   MailClients  = ( iu -nexus )

###
#
# END cf.groups
#
###
```

12.3 cf.main

```
#############################################################
#
# cf.main - for iu.hioslo.no
#
# This file contains generic config stuff
#
################################################################

###
#
# BEGIN cf.main
#
###

control:

   access    = ( root )        # Only root should run this

   site      = ( iu )
   domain    = ( iu.hioslo.no )
   sysadm    = ( drift@iu.hioslo.no )

   repository = ( /var/spool/cfengine )

   netmask   = ( 255.255.255.0 )
   timezone  = ( MET )
   nfstype   = ( nfs )

   sensiblesize  = ( 1000 )
   sensiblecount = ( 2 )
   editfilesize  = ( 20000 )

   mountpattern = ( /$(site)/$(host) )
   homepattern  = ( u? )

   #
```

```
# If we undefine this with cfengine -N longjob
# then we switch off all jobs labelled with this class
#

addclasses = ( longjob )

#
# Macros & constants are inherited downwards in imports
# but are not passed up to parent files. Good idea to
# define them all here
#

masterfiles = ( /iu/nexus/local/iu )
main_server = ( nexus )
cfbin       = ( /iu/nexus/local/gnu/lib/cfengine/bin )
gnu         = ( /local/gnu )
ftp         = ( /local/iu/ftp )
nisslave    = ( dax )
nisfiles    = ( /iu/nexus/local/iu/etc )

#
# The action sequence for daily (full) runs and
# for hourly updates (called with -DHourly)
#

Hr00::

   actionsequence =
      (
      copy
      mountall
      mountinfo
      checktimezone
      netconfig
      resolve
      unmount
      shellcommands
      addmounts
      links.Prepare
      files.Prepare
      directories
      links.Rest
      mailcheck
      mountall
      required
      tidy
      disable
      editfiles
```

```
          files.Rest
          processes
          )

    !Hr00::

       actionsequence =
          (
          resolve
          shellcommands
          copy
          editfiles
          processes
 links
          )

    force::

       actionsequence =
          (
          files.Prepare.Rest
          tidy
          )

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

homeservers:

   iu:: nexus

binservers:

   iu.solaris::                 nexus
   iu.linux::                   borg

mailserver:

   any:: nexus:/var/mail

mountables:

   any::
         nexus:/iu/nexus/u1
         nexus:/iu/nexus/u2
         nexus:/iu/nexus/u3
         nexus:/iu/nexus/u4
         nexus:/iu/nexus/u5
         nexus:/iu/nexus/u6
```

```
         nexus:/iu/nexus/ua
         nexus:/iu/nexus/ud
         nexus:/iu/nexus/local
         nexus:/opt/NeWSprint
         nexus:/opt/AcroRead
         borg:/iu/borg/local
         dax:/iu/dax/local

miscmounts:

   linux||freebsd::   nexus:/iu/nexus/local /iu/nexus/local ro

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

broadcast:

  ones

defaultroute:

   cadeler30-gw

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

resolve:

      128.39.89.10  # nexus
      158.36.85.10  # samson.hioslo.no
      129.241.1.99

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

tidy:

   #
   # Some global tidy-ups
   #

      /tmp/                    pat=*             r=inf     A=1
      /var/tmp                 pat=*             r=inf     A=1
      /                        pat=core          r=1       A=0
      /etc                     pat=core          r=1       A=0

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

ignore:                       # Don't check or tidy these directories

      /local/lib/gnu/emacs/lock/
```

```
      /local/tmp
      ftp
      projects
      /local/bin/top
      /local/lib/tex/fonts
      /local/iu/etc
      /local/etc
      /local/iu/httpd/conf
      /usr/tmp/locktelelogic
      /usr/tmp/lockIDE
      RootMailLog

      #
      # Emacs lock files etc
      #

      !*
      /local/lib/xemacs

      #
      # X11 keeps X server data in /tmp/.X11
      # better not delete this!
      #

      .X11

      #
      # Some users like to give a file or two 777 protection here
      # so netsurfers can update a log or counter when running as
      # ‘nobody’
      #

      www

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

disable:

   /etc/hosts.equiv
   /etc/nologin
   /usr/lib/sendmail.fc


###
#
# END cf.main
#
###
```

12.4 cf.site

```
##############################################################
#
# cf.site - for iu.hioslo.no
#
# This file contains site specific data
#
#################################################################

###
#
# BEGIN cf.site
#
###

links:

   Prepare::

      /local     -> /$(site)/$(binserver)/local
      /usr/local -> /local

   dax::

      /iu/dax/local             +> /iu/nexus/local
      /projects                 -> /iu/dax/local/projects
      /iu/nexus/u1/sowille/data -> /iu/dax/scratch/data

   XBootServer::

      #
      # Set up a /local/tftpboot area where all X terminal
      # stuff will be kept.
      #

      /tftpboot                  -> /local/tftpboot
      /local/tftpboot/td/configs -> /local/tftpboot/td/examples/configs
      /etc/bootptab              -> /tftpboot/bootptab
      /tftpboot/usr/lib/X11/td   -> /tftpboot/td

   NameServers::

      /etc/named.boot -> /local/iu/named/named.boot

   MailHub::
```

```
      /etc/mail/sendmail.cf ->! /iu/nexus/local/mail/sendmail.cf

   MailClients.solaris::

      /etc/mail/sendmail.cf ->! /iu/nexus/local/mail/client.cf

   nexus::

/local/bin +> /local/latex/bin

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

disable:

  #
  # We run Berkeley sendmail and the config files are
  # all under /iu/nexus/local/lib/mail
  #

    /etc/aliases

 WWWServers.Sunday::

   #
   # Disabling these log files weekly prevents them from
   # growing so enormous that they fill the disk!
   #

   /local/iu/httpd/logs/access_log   rotate=empty
   /local/iu/httpd/logs/agent_log    rotate=empty
   /local/iu/httpd/logs/error_log    rotate=empty
   /local/iu/httpd/logs/referer_log  rotate=empty

   #
   # CERT warning, security fix
   #

  any::

    /usr/lib/expreserve

  FTPserver.Sunday.Hr00::

   /local/iu/xferlog rotate=3

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

```
files:

  Prepare::

      /etc/motd              m=0644 r=0 o=root act=touch
      /.cshrc                m=0644 r=0 o=root act=touch

   PasswdServer::

      /local/iu/etc/passwd m=0644 o=root g=other action=fixplain
      /local/iu/etc/shadow m=0644 o=root g=other action=fixplain

   WWWServers.Rest::

      /local/iu/www                         m=775        g=www act=fixall r=inf
      /local/iu/httpd/conf                  m=664 o=root g=www act=fixall r=inf
      /local/iu/www/cgi-bin-public/count_file m=777 o=root g=www act=fixplain

   FTPserver::

      #
      # Make sure anonymous ftp areas have the correct
      # protection, or logins won't be able to read
      # files - or perhaps a security risk. This is
      # solaris 2 specific...
      #

      $(ftp)/pub        mode=755 o=ftp  g=ftp  r=inf act=fixall
      $(ftp)/Obin       mode=111 o=root g=other      act=fixall
      $(ftp)/etc        mode=111 o=root g=other      act=fixdirs
      $(ftp)/usr/bin/ls mode=111 o=root g=other      act=fixall
      $(ftp)/dev        mode=555 o=root g=other      act=fixall
      $(ftp)/usr        mode=555 o=root g=other      act=fixdirs

   Prepare::

      /etc/shells mode=0644 action=touch

   AllBinaryServers.Rest.longjob::

     /local mode=-0002 r=inf owner=root,bin group=0,1,2,3,4,5,6,7,staff
            links=tidy action=fixall

     /local/iu/RootMailLog  m=0666 action=touch

   dax.Rest::

    /iu/dax/scratch        r=0 o=root mode=1777 action=fixall
```

```
    /iu/dax/local/projects r=0 o=root mode=755  action=fixdirs

   nexus::

    /local/mail/sendmail.cf o=root m=444 act=fixplain

    /iu/nexus/ua/robot/.rhosts o=robot m=600 act=touch

    /local/iu/named/pz         o=root  m=644 act=fixall r=1

    /local/latex/lib/tex/texmf/fonts  owner=root
                                      mode=1666
                                      recurse=inf
                                      action=fixall

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

tidy:

      #
      # Make sure the file repository doesn't fill up
      #

      /var/spool/cfengine pattern=*    age=3

      /var                pattern=core age=0  r=inf
      /var/spool/mqueue   pattern=*    age=14 type=mtime

   BackupHost::

      # Here we tidy old backup tar files from the backup area
      # A special tmp area gets cleared every 4 days. The files
      # are created by Audun's backup help script (see shellcommands)

      /iu/nexus/backup1      pat=*  age=7

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

shellcommands:

   PasswdServer::

      # Build and install the BSD compatible passwd file
      # from the master passwd/shadow file on solaris

      "/local/iu/bin/BuildPasswdFiles"
      "/local/iu/bin/BuildGroupFiles"
```

```
  BackupHost.Sunday.Hr00|BackupHost.Wednesday.Hr00::

      #
      # Make a system backup of /iu/nexus/u? with Audun's script
      #

      "$(cfbin)/cfbackup -p -f /iu/nexus/backup1 -s /iu/nexus/ud"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup1 -s /iu/nexus/ua"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup1 -s /iu/nexus/u1"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup1 -s /iu/nexus/u2"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup2 -s /iu/nexus/u3"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup2 -s /iu/nexus/u4"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup2 -s /iu/nexus/u5"
      "$(cfbin)/cfbackup -p -f /iu/nexus/backup2 -s /iu/nexus/u6"

  nexus.Sunday.longjob.Hr00::

      #
      # See how much rubbish users have accumulated each Sunday
      #

      "$(cfbin)/noseyparker /iu/nexus/u1 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/u2 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/u3 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/u4 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/u5 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/u6 $(sysadm) "
      "$(cfbin)/noseyparker /iu/nexus/ua $(sysadm) nomail"
      "$(cfbin)/noseyparker /iu/nexus/ud $(sysadm) nomail"

   nexus.longjob.Hr00::

      #
      # Update the GNU find/locate database each night
      #

      "$(gnu)/lib/locate/updatedb"
      "/local/iu/bin/newhomepage.sh"

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

editfiles:

    #
    # cfengine installs itself as a cron job - sneaky! :)
    #

    { /var/spool/cron/crontabs/root
```

```
    AppendIfNoSuchLine "0 * * * * $(cfbin)/cfwrap $(cfbin)/cfhourly"
    }

   FTPserver::

      { /etc/shells

      AppendIfNoSuchLine "/bin/tcsh"
      AppendIfNoSuchLine "/local/gnu/bin/bash"
      }


   XBootServer::

      { /etc/inetd.conf

      AppendIfNoSuchLine
          "bootp dgram udp wait root /local/bin/bootpd bootpd -i -d"
      }

   nexus::

      { /iu/nexus/ua/robot/.rhosts

      AppendIfNoSuchLine "borg"
      AppendIfNoSuchLine "borg.iu.hioslo.no"
      AppendIfNoSuchLine "aud4"
      AppendIfNoSuchLine "aud4.iu.hioslo.no"
      }

   dax::

      { /etc/system

      AppendIfNoSuchLine "set pt_cnt=128"
      }

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

required:

   #
   # Any host must have a /local, /usr/local fs. Check that
   # it exists and looks sensible. (i.e. not empty)
   #
```

```
   /$(site)/$(binserver)/local

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

copy:

   #
   # NIS seems broken at IU, so here we use NFS to fudge
   # a file distribution as a temporary solution. Actually
   # this makes the system work faster without NIS!
   #

      $(nisfiles)/services dest=/etc/services o=root g=other mode=0644
      $(nisfiles)/hosts.deny dest=/etc/hosts.deny o=root mode=0644

   !debian::

      $(nisfiles)/hosts    dest=/etc/hosts o=root g=other mode=0644

   PasswdServer::

      /etc/passwd dest=$(nisfiles)/passwd o=root g=other mode=0644
      /etc/shadow dest=$(nisfiles)/shadow o=root g=other mode=0644

   nexus::

      /local/iu/etc/dfstab dest=/etc/dfs/dfstab  o=root  mode=0744

   solaris.!PasswdServer::

      $(nisfiles)/passwd dest=/etc/passwd o=root g=other mode=0644
      $(nisfiles)/shadow dest=/etc/shadow o=root g=other mode=0600
      $(nisfiles)/group.solaris dest=/etc/group o=root g=other mode=0644

   linux::

      $(nisfiles)/passwd.linux dest=/etc/passwd o=root g=other mode=0644
      $(nisfiles)/group.linux dest=/etc/group o=root g=other mode=0644

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

processes:

      "eggdrop"                            signal=kill
      "irc"                                signal=kill
      "ping"                               signal=kill
      "NetXRay"                            signal=kill
```

```
      "netxray"                       signal=kill
      "ypserv"                        signal=kill
      "ypbind"                        signal=kill
      "rarpd"                         signal=kill
      "rpc.boot"                      signal=kill
      "README"                        signal=kill # You don't sh README !

   !XBootServer::

      "bootp"                         signal=kill

   #
   # These processes are not killed every hour, but once a day
   # when cfengine runs at night. Note that there are often
   # hanging pine and elm processes. These programs crash and
   # go berserk, using hundreds of hours of CPU time.
   #

   Hr00::

      "cron"                signal=hup  # HUP these to update their config
      "inetd"               signal=hup

      "/local/sdt/sdt/bin"  signal=term # For those elektro dudes who forget
                                        # to log out
      "netscape"            signal=kill
      "pine"                signal=kill
      "elm"                 signal=kill


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

12.5 cf.motd

```
###################################################################
#
# cf.motd
#
# This file is used to set the message of the day file on
# each host
#
###################################################################
```

```
 #####
 #
 # BEGIN cf.motd
 #
 #####


control:

   #
   # This points to the file containing general text
   #

   masterfile      = ( /iu/nexus/local/iu/etc/motd-master )
   local_message   = ( /etc/motd.local )

editfiles:

      { /etc/motd

      BeginGroupIfFileIsNewer "$(masterfile)"
        EmptyEntireFilePlease
        InsertFile "$(masterfile)"
        InsertFile "$(local_message)"
        PrependIfNoSuchLine "This system is running $(class):$(arch)"
      EndGroup
      }

 #####
 #
 # BEGIN cf.motd
 #
 #####
```

12.6 cf.users

Whether or not you perform any special services for users, with or without their consent is entirely a matter of local policy. In a school or college situation, users are often uncooperative and some are even irresponsible. This file shows you what you could do in an environment with inexperienced users, but please don't feel as though you have to be this totalitarian.

```
#################################################################
#
# cf.users - for iu.hioslo.no
#
# This file contains user specific actions
#
```

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

###
#
# BEGIN cf.users
#
###

ignore:

    robot

tidy:

   longjob::

     #
     # Some users just don't understand what they are doing
     # and this is safest, allbeit totalitarian
     #

     home                 pat=.rhosts                       age=0

     #
     # Tidy up users' home dirs
     #

     home                 pat=core             r=inf        age=0
     home                 pat=a.out            r=inf        age=1
     home                 p=*%                 r=inf        age=2
     home                 p=*~                 r=inf        age=2
     home                 p=#*                 r=inf        age=1
     home                 p=*.dvi              r=inf        age=14   type=ctime
     home                 p=*.log              r=inf        age=2
     home                 p=Log.*              r=inf        age=3
     home                 p=CKP                r=inf        age=1
     home                 p=BAK                r=inf        age=1
     home                 p=log                r=inf        age=0
     home                 p=*.o                r=inf        age=0
     home                 p=*.aux              r=inf        age=3
     home                 p=*.zip              r=inf        age=7
     home/.deleted        p=*                  r=inf        age=0
     home/.wastebacket    p=*                  r=inf        age=14
     home/www             p=*~                 r=inf        age=1

     #
     # Clear the big cache files netscape creates
     #
```

```
     home/.netscape-cache  p=cache????*         r=inf       age=0
     home/.MCOM-cache      p=cache????*         r=inf       age=0
     home/.netscape/cache  p=*                  r=inf       age=0


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

files:

   AllHomeServers.longjob.rest::

     #
     # Check users files are not writable to the world
     # and there are no stale links (pointing nowhere)
     #

     home mode=o-w recurse=inf action=fixall # links=tidy

     home/.xsession mode=755 action=fixall
     home/.cshrc    mode=755 action=fixall

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

copy:

   Hr00.longjob::

   #
   # Make sure each user has an up to date standard
   # setup.  Cshrc just sources in a big standard file
   # which is kept in ~user/../.setupfiles/cshrc
   # to reduce disk wastage
   #

   $(masterfiles)/lib/Cshrc   dest=home/.cshrc
   $(masterfiles)/lib/tkgrc   dest=home/.tkgrc
   $(masterfiles)/lib/fvwm2rc dest=home/.fvwm2rc


###
#
# END cf.users
#
###
```

12.7 cf.solaris

```
#################################################################
#
# cf.solaris - for iu.hioslo.no
#
# This file contains solaris specific patches
#
#################################################################

###
#
# BEGIN cf.solaris
#
###

directories:

     #
     # httpd/netscape want this to exist for some bizarre reason
     #

      /usr/lib/X11/nls

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

tidy:

     /var/log  pattern=syslog.* age=0

   MailHub::

     /var/mail pattern=lp       age=0

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

files:

  #
  # If this doesn't exist fork will not work and the
  # system will not even be able to run the /etc/rc
  # scripts at boottime
  #

  /etc/system     o=root g=root m=644 action=touch

  /var/log/syslog o=root        m=666 action=touch

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

```
copy:

   #
   # Some standard setup files, can't link because
   # machine won't boot if their not on / partition.
   #

   /local/bin/tcsh dest=/bin/tcsh mode=755

   /local/iu/etc/nsswitch.standalone dest=/etc/nsswitch.conf

  #
  # Our named server uses a newer BIND
  # Put this here so that it will be preserved under
  # solaris reinstallation
  #

  NameServers::

   /local/iu/sbin/in.named         dest=/usr/sbin/in.named         mode=555
   /local/iu/sbin/in.named.reload  dest=/usr/sbin/in.named.reload  mode=555
   /local/iu/sbin/in.named.restart dest=/usr/sbin/in.named.restart mode=555
   /local/iu/sbin/in.ndc           dest=/usr/sbin/in.ndc           mode=555
   /local/iu/sbin/named-xfer       dest=/usr/sbin/named-xfer       mode=555
   /local/iu/lib/nslookup.help     dest=/usr/lib/nslookup.help     mode=444

  any::
   /local/iu/lib/libresolv.a        dest=/usr/lib/libresolv.a      mode=444
   /local/iu/lib/libresolv.so.2     dest=/usr/lib/libresolv.so.2   mode=444
   /local/bin/nslookup              dest=/usr/sbin/nslookup        mode=444

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

editfiles:

      { /etc/netmasks

      AppendIfNoSuchLine "128.39  255.255.255.0"
      }

      { /etc/defaultrouter

      AppendIfNoSuchLine "128.39.89.1"
      }

      { /usr/openwin/lib/app-defaults/XConsole

      AppendIfNoSuchLine "XConsole.autoRaise: on"
```

```
      }

   #
   # CERT security patch for vold vulnerability
   #

   { /etc/rmmount.conf

   HashCommentLinesContaining "action cdrom"
   HashCommentLinesContaining "action floppy"
   }

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

disable:

    /etc/.login  type=file
    /etc/aliases

   #
   # These files are ENORMOUS, don't let them fill the disk
   #

   Wednesday::

      /var/lp/logs/lpsched rotate=empty

      /var/adm/wtmpx       rotate=empty
      /var/adm/wtmp        rotate=empty

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

files:

    /etc/passwd        m=0644 o=root g=other action=fixplain
    /etc/shadow        m=0600 o=root g=other action=fixplain
    /etc/defaultrouter m=0644 o=root g=other action=touch
    /var/adm/wtmpx     m=0664 o=adm  g=adm   action=touch
    /var/adm/wtmp      m=0644 o=root g=adm   action=touch
    /var/adm/utmp      m=0644 o=root g=adm   action=fixplain
    /var/adm/utmpx     m=0664 o=adm  g=adm   action=fixplain

    /tmp m=1777                              action=fixdirs

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

disable:
```

```
   #
   # CERT security patch
   #

   /usr/openwin/bin/kcms_calibrate
   /usr/openwin/bin/kcms_configure
   /usr/bin/admintool

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

shellcommands:

   AllBinaryServers.Saturday.longjob.Hr00::

      #
      # Make sure the man -k / apropos data are up to date
      #

      "/usr/bin/catman  -M /local/man"
      "/usr/bin/catman  -M /local/X11R5/man"
      "/usr/bin/catman  -M /usr/man"
      "/usr/bin/catman  -M /local/gnu/man"
      "/usr/bin/catman  -M /usr/openwin/share/man"
      "/usr/bin/catman  -M /local/X11R5/man"
      "/usr/bin/catman  -M /usr/share/man"

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

editfiles:


      #
      # A painless way to add an rc.local script to the rc files
      # under solaris without having to fight though inittab
      #

      { /etc/rc3.d/S15nfs.server

      AppendIfNoSuchLine "sh /local/iu/etc/rc.local"
      }

      #
      # umask defined when inetd starts is inherited by all subprocesses
      # including ftpd which saves with mode 666 (!) unless we do this
      #

      { /etc/rc2.d/S72inetsvc
```

```
      PrependIfNoSuchLine "umask 022"
      }


###
#
# END cf.solaris
#
###
```

12.8 cf.linux

```
#################################################################
#
# cf.linux - for iu.hioslo.no
#
# This file contains debian linux specific patches
#
#################################################################

###
#
# BEGIN cf.linux
#
###

files:

      /etc/printcap m=644 o=root action=fixplain

     #
     # Cert advisories
     #

      /bin/mount         m=755 o=root action=fixall
      /bin/umount        m=755 o=root action=fixall

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

  disable:

     #
     # Cert advisories
     #

      /sbin/dip-3.3.7n
```

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

links:

    /local/bin/tcsh   ->  /bin/tcsh

    /local/lib/mail   ->  /$(site)/$(main_server)/local/lib/mail

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

editfiles:

  #
  # Samba default mode needs to be set...
  #

   { /etc/smb.conf

   ReplaceAll "700" With "644"
   }

  #
  # Linux date is very stupid and needs a very careful
  # TZ definition, otherwise it loses
  #

   { /etc/csh.cshrc

   AppendIfNoSuchLine "setenv TZ 'MET-1MET DST-2,M3.5.0/2,M10.5.0/3'"
   }

  #
  # resolv+ ordering
  #

   { /etc/host.conf

   PrependIfNoSuchLine "order bind"
   }


  #
  # Should have been configured already (!)
  #

   { /etc/ld.so.conf

   AppendIfNoSuchLine "/usr/X11R6/lib"
```

```
   }

  #
  # Kill annoying messages
  #

   { /etc/cron.daily/standard

   HashCommentLinesContaining "security"
   }

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

shellcommands:

  Hr00::

     #
     # Find/locate database
     #

     "/usr/bin/updatedb"

###
#
# END cf.linux
#
###
```

12.9 cf.freebsd / cf.netbsd

FreeBSD and NetBSD are sufficiently similar to have a single file for both.

```
#################################################################
#
# cf.bsd - for iu.hioslo.no
#
# This file contains bsd specific patches
#
#################################################################

###
#
# BEGIN cf.bsd
#
###
```

```
links:

    /usr/spool        ->  /var/spool
    /local/bin/tcsh   ->  /bin/tcsh
    /local/bin/perl   ->  /usr/bin/perl
    /usr/lib/sendmail ->  /usr/sbin/sendmail

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

files:

   /usr/tmp mode=1777 owner=root action=fixall

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

editfiles:

   #
   # Comment out all lines to shut up this annoying cfengine-like
   # script, which sends mail every day!!!
   #

   { /etc/crontab

   HashCommentLinesContaining "daily"
   HashCommentLinesContaining "weekly"
   HashCommentLinesContaining "monthly"
   }

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

copy:

      $(masterfiles)/etc/printcap.client      dest=/etc/printcap mode=0644

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

shellcommands:

  Hr00::

    "/usr/libexec/locate.updatedb"
    "/usr/bin/makewhatis /usr/share/man:/usr/X11R6/man"

###
#
# END cf.bsd
#
```

```
###
```

12.10 cfd.conf

```
#########################################################
#
# This is a cfd config file
#
# The access control here follows after any tcpd
# control in /etc/hosts.allow and /etc/hosts.deny
#
#########################################################

 #
 # Could import cf.groups here and use a structure like
 # in cfengine.conf, cf.main, cf.groups
 #

control:

  public = ( /usr/local/publicfiles )

  almost_public = ( /usr/local/almostpublicfiles )

  cfrunCommand = ( /iu/nexus/ud/mark/comp/Tests/cfrun-command )

  MaxConnections = ( 10 )

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

admit:   # or grant:

     $(public) *

     $(almost_public) *.iu.hioslo.no *.gnu.ai.mit.edu

     /etc/passwd *.iu.hioslo.no

     #
     # Who can exec cfengine remotely?
     #

     $(cfrunCommand) *.iu.hioslo.no

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

```
deny:

     $(public)/special *.moneyworld.com
```

Appendix A Runtime Options

Note that GNU long options are available with the syntax `--longoption`. The long names are given in brackets.

'`-a`' (`--sysadm`) Print only the name of the system administrator then quit.

'`-A`' (`--auto`) Can be used to signify an automatic run of cfengine, as opposed to a manual run. The distinction is not predetermined. Use of this option currently causes cfengine to ignore locks. This option is reserved for future development.

'`-c`' (`--no-check-files`) Do not check file systems for ownership / permissions etc.

'`-C`' (`--no-check-mounts`) Check mount points for consistency. If this option is specified then directories which lie in the "mount point" area are checked to see whether there is anything mounted on them. Normally this is *off* since not all machines use mounted file systems in the same way. e.g. HPUX does not generally operate with partitions, but nevertheless one might wish to mimick a partition-like environment there, but it would be irritating to be informed that nothing was mounted on the mount point.

'`-d`' (`--debug`) Enable debugging output. Normally you will want to send this to a file using the shell script command or a pipe. -d1 shows only parsing output. -d2 shows only runtime action output. -d0 shows both levels. Debugging ouput is intended mainly for the author's convenience and is not a supported feature. The details of this output may change at any time.

'`-D`' (`--define`) Define a compound class symbol of the form *alpha.beta.gamma*.

'`-e`' (`--no-edits`) Suppress file editing.

'`-E`' (`--enforce-links`) Globally force links to be created where plain files or links already exist. Since this option is a big hammer, you have to use it in interactive mode and answer a yes/no query before cfengine will run like this.

'`-f`' (`--file`) Parse filename after this switch. By default cfengine looks for a file called *cfengine.conf* in the current directory.

'`-h`' (`--help`) Help information. Display version banner and options summary.

'`-H`' (`--no-hard-classes`). Prevents cfengine from generating any internal class name information. Can be used for emulation purposes.

'`-i`' (`--no-ifconfig`) Do not attempt to configure the local area network interface.

'`-I`' (`--inform`) Switches on the inform output level, whereby cfengine reports everything it changes..

'`-k`' (`--no-copy`) Do not copy/image any files.

'`-K`' (`--no-lock`) Ignore locks when running.

'`-l`' (`--traverse-links`) Normally cfengine does not follow symbolic links when recursively parsing directories. This option will force it to do so.

'`-L`' (`--delete-stale-links`) Delete links which do not point to existing files (except in user home directories, which are not touched).

'-m' (--no-mount) Do not attempt to mount file systems or edit the filesystem table.

'-n' (--recon,--dry-run,--just-print) No action. Only print what has to be done without actually doing it.

'-N' (--negate,--undefine) Cancel a set of classes, or undefine (set value to *false*) a compound class of the form *alpha.beta.gamma*.

'-p' (--parse-only) Parse file and then stop. Used for checking the syntax of a program. You do not have to be superuser to use this option.

'-q' (--no-splay) Switch off host splaying (sleeping).

'-s' (--no-commands) Do not execute scripts or shell commands.

'-S' (--silent) Silence run time warnings.

'-t' (--no-tidy) Do not tidy file systems.

'-u' (--use-env) Causes cfengine to generate an environment variable 'CFALLCLASSES' which can be read by child processes (scripts). This variable contains a summary of all the currently defined classes at any given time. This option causes some system 5 systems to generate a Bus Error or segmentation fault. The same information is available from the cfengine internal variable `$(allclasses)` and can be passed as a parameter to scripts.

'-U' (--underscore-classes). When this option is set, cfengine adds an underscore to the beginning of all hard system classes (like `_sun4`, `_linux` etc.) This can be used to avoid naming conflicts if you are so unjudicious as to name a host by the name of a hard class. Other classes are not affected.

'-v' (--verbose) Verbose mode. Prints detailed information about actions and state.

'-V' (--version) Print only the version string and then quit.

'-x' (--no-preconf) Do not execute the 'cf.preconf' net configuration file.

'-X' (--no-links) Do not execute the `links` section of a program.

'-w' (--no-warn,--quiet) Do not print warning messages.

Appendix B Cfengine network protocol specs

Cfengine uses a simple protocol for communicating via a streams-based tcp connection. This section documents the protocol for anyone who might want to create their own clients or server components to interface with cfengine. Several transfers use a standard buffer size of 4096 bytes. The get-file service uses a character based read interface in which the buffer size is not directly relevant; the size of the get-file buffer is dictated by client-side disk blocksizes.

Each new connection to the remote server daemon must begin with a verification or 'login' string whereby the client identifies itself to the server. This information is used to verify the connection by using a reverse DNS lookup and then a double-reverse lookup. This is the basis of hostname authentication.

The various services are listed below:

Stat file

```
AUTH client-name server-name
SYNCH long-time-string STAT filename

reply with OK: <stat-reply-string>
```

Get file

```
AUTH client-name server-name
GET filename

reply with <stream>, break on zero chars received
or BAD: <message>
```

Opendir Recursive parsing of directory trees poses a technical problem. If cfengine keeps a connection open and send buffers on a need-to-know basis, then the number of daemon connections will mount up and overload the server. If cfengine caches the entire directory on the client side, allowing the server connection to be severed, then the caching could easily fill the memory of the client. The policy chosen is to attempt to cache all names client-side, in spite of the possible memory problem. The reason for this choice is that, even on large filesystems (max size of paritions with 32 bit pointers is 4GB), the sum memory used by every filename is only of the order of a few megabytes, and this is within reasonable modern memory limits. It is assumed that, even 64 bit users will not create filesystems which are much large than this.

```
AUTH client-name server-name
OPENDIR dir-name

reply with <stream>, break on zero chars received
```

In the future it might be useful to stat the file automatically here and cache the value client-side.

Exec

```
AUTH client-name server-name
EXEC option-string
CLASSES stream terminated with --- (CFD_TERMINATE)

reply with <stream>, break on zero chars received
```

Checksum verification

```
MD5 filename 16 byte sequence

reply with CFD_TRUE for no match (copy) or CFD_FALSE for match (no copy)
```

Reply formatting

The format of reply messages, except for stream data, is

`OK:` *message*

`BAD:` *message*

A return prefix of 'BAD' implies a failure server-side and the client-side wrapper functions return -1 in this case.

The server daemon currently runs single threaded for all requests except GetFile. Since cfd uses heavyweight processes for general applicablity this avoids unnecessary forking and context switching which would download the server. An upper limit on the number of forks which may be performed is set in the config file. This is mananged using the SIGCHLD signal and a pair of arrays in the master processes (This approach is used to avoid shared memory and semaphore usage which is not portable to many older BSD derivative systems).

Variable Index

(Index is nonexistent)

Concept Index

(Index is nonexistent)

FAQ Index

(Index is nonexistent)

Table of Contents