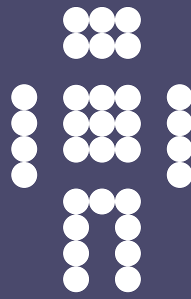
CFEngine



CFEngine 3 Concept Guide

A CFEngine AS workbook

CFEngine AS

Copyright © 2011 CFEngine AS

Table of Contents

1 Introduction - System automation 3
  1.1 Managing diverse and challenging environments seamlessly and invisibly 3
  1.2 Managing expectations - a theory of promises 3
  1.3 Why automation? 4
  1.4 How do *you* view CFEngine? 4

2 The components of CFEngine 7
  2.1 The players 7
  2.2 About the CFEngine architecture 8
  2.3 The policy decision flow 9

3 Bodies and bundles 11
  3.1 Bodies 11
    3.1.1 Body parts 11
    3.1.2 Control bodies 13
  3.2 Bundles 13
    3.2.1 Bundle scope 14
  3.3 A simple syntax pattern 14

4 A simple crash course in concepts 17
  4.1 Rules are promises 17
  4.2 Control promises 18
  4.3 Variables 20
    4.3.1 Scalar variables 20
    4.3.2 List variables 20
    4.3.3 Associative arrays 21
  4.4 Decisions 22
  4.5 Loops 25
  4.6 The main promise types 26
  4.7 Test a promise? 26

5 Knowledge Management 27
  5.1 Promises and Knowledge 27
  5.2 The basics of knowledge 28
  5.3 Annotating promises 28
  5.4 A promise model of topic maps 29
  5.5 What topic maps offer 30
  5.6 The nuts and bolts of topic maps 31
    5.6.1 Topic map definitions 31
  5.7 Example of topics promises 32
    5.7.1 Analysing and indexing the policy 33

5.7.2 `cf-know` 34
5.8 Modelling configuration promises as topic maps 35

6 More... 37

This document is an abreviated version of the CFEngine tutorial (http://cfengine.com/manuals/cf3-tutorial.html).

1 Introduction - System automation

1.1 Managing diverse and challenging environments seamlessly and invisibly

CFEngine was designed to enable scalable configuration management, for the whole system life-cycle, in any kind of environment. Almost every other system for configuration assumes that there will be a reliable network in place and that changes will be pushed out top-down from an authoritative node. Those systems are useless in environments like

- Mobile systems with partial or unreliable connectivity (e.g. a submarine).
- Systems where bandwidths are very low (e.g. a satellite or space probe).
- Systems where computing power is very low (e.g. ad hoc sensors or kitchen appliances).

CFEngine does not need reliable infrastructure. It works opportunistically in almost any environment, using few resources. It has few software dependencies. So, not only does it work in all of the traditional fixed-plan scenarios, but it is capable of working in totally ad hoc deployment: an temporary incident room, a submarine drifting on and off line, a satellite or a robot explorer.

One could argue 'well I don't need that kind of system, because my network is reliable'. However, your network is not as reliable as you think, and mobility is an increasingly important topic. Even with a very strong redundant network, the services that support the network can be paralyzed by any of a number of failed dependencies or mishaps. It is crucial in a modern pervasive environment that systems remain available, fault tolerant and as far as possible independent of external requirements. This is how to build scalable and reliable services.

> CFEngine works in all the places you think it should, and all the new places you haven't even thought of yet. How do we know? Because it is based on almost 20 years of careful research and experience.

1.2 Managing expectations - a theory of promises

One of the hardest things in management is to make everyone aware of their roles and tasks, and to be able to rely on others to do the same. *Trust* is an economic time-saver. If you can't trust you have to verify, and that is expensive.

To improve trust we make promises. A promise is the documentation of an intention to act or behave in some manner. This is what we need to learn to trust systems, no matter whether they are machines or humans.

One CFEngine user once said to me, that the thing that had helped him the most in deploying CFEngine was its design based around voluntary cooperation. "Our main problems were not technical but political – getting everyone to agree in all of our departments around the world". This was because, for all the technology, it is people who make the decisions and people need to feel that the system is empowering rather than disempowering them.

CFEngine works on a simple notion of promises. Everything in CFEngine can be thought of as a promise to be kept by different resources in the system.

Combining promises with patterns to describe where and when promises should apply is what CFEngine is all about.

1.3 Why automation?

Humans are good at making decisions and awful at reliable implementation. Machines are pitiful at making decisions and very good at reliable implementation. It makes sense to let each side do the job that they are good at.

The main problem in managing systems is a loss of self-discipline. Discipline does not imply that order have to be barked from a central command. It only requires that every part of the system knows its job and carries is out seamlessly and flawlessly.

Skilled workers tend to think that it is enough to be smart. In fact this is wrong: smart people tend to be problem solvers and will happily solve the same problem many times, wasting time and effort. Moreover, human intervention is often based on panic and lack of understanding so every time someone logs onto a system by hand, they jeopardize everyone's understanding of the system. Only the self-discipline of stable procedures leads to predictability.

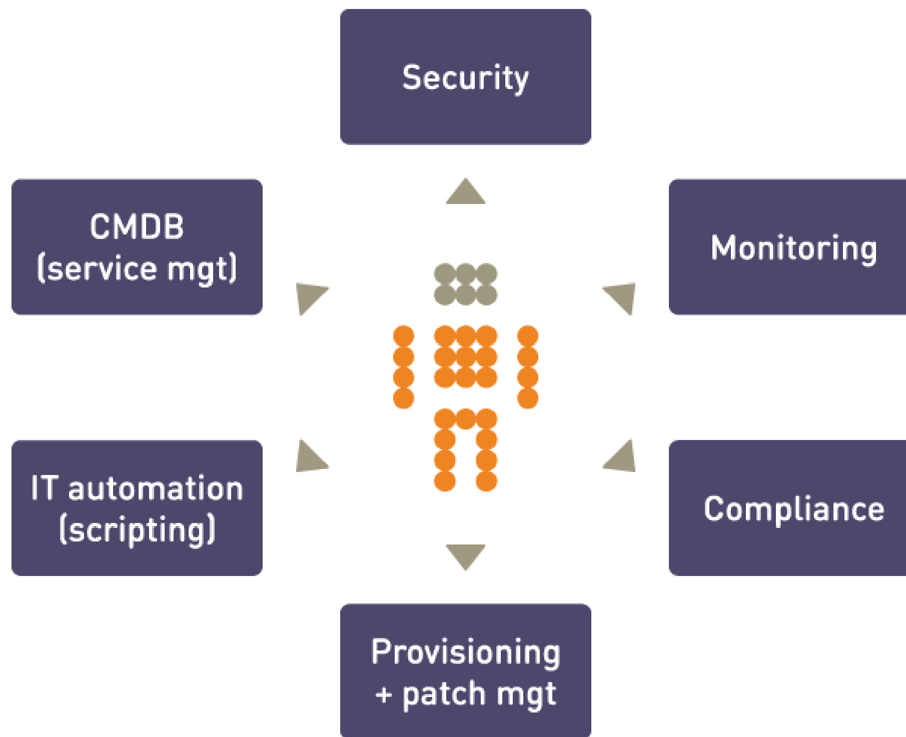
Ad hoc changes are bad because:

- Others have no idea what happened.
- There is no record of changes or intentions.
- A scar is left from the change.

People often rile against automation saying that it dehumanizes their work. In fact the opposite is true: forcing humans to do the work of machines, in repetitive and reliable ways is what dehumanizes people. The only way to make progress with a bad habit is to recognize it and be willing to abandon the habit.

1.4 How do *you* view CFEngine?

CFEngine is a framework. It is not so complex, but it is certainly extensive. Often when trying to describe CFEngine, it seems that there is too much to tell and it is hard to convey in a simple way what the software can do. The picture below shows a few ways in which you can think of CFEngine.



For many users, CFEngine is simply a configuration tool – i.e. software for deploying and patching systems according to a policy. Policy is described using promises – indeed, every statement in CFEngine 3 is a promise to be kept at some time or location. More than this, however, CFEngine is not like most automation tools that 'roll out' an image of some software once and hope for the best. Every promise that you make in CFEngine is continuously verified and maintained. It is not a one-off operation, but an encapsulated process that repairs itself should anything deviate from the policy.

That clearly places CFEngine in the realm of automation, which often begs the question: so it's just another scripting language? Certainly CFEngine contains a powerful scripting language, but it is not like any other. CFEngine is not a low level language like Perl, Python or Ruby; it is a language of promises, in which you express very high level intentions about the system and the inner details figure out the algorithms needed to implement the result.

Above all, CFEngine is aimed to promote human understanding of complex processes. Its promises are easily documentable using comments that the system remembers and reminds us about in error reporting. It hides irrelevant and transitory details of implementation so that the *intentions* behind the promises are highlighted for all to see. This means that the knowledge of your organization can be encoded into the CFEngine language.

WHY DOES KNOWLEDGE MATTER? 1. Technical descriptions are hard to remember. You might understand your configuration decisions when you are writing them, but a few months later when something goes wrong, you will probably have forgotten what you were thinking. That costs you time and effort to diagnose. 2. Organizations are fragile to the loss of those individuals who code policy. If they leave, often there is no one left who can understand or fix the system. Only with proper documentation is it possible to immunize against loss.

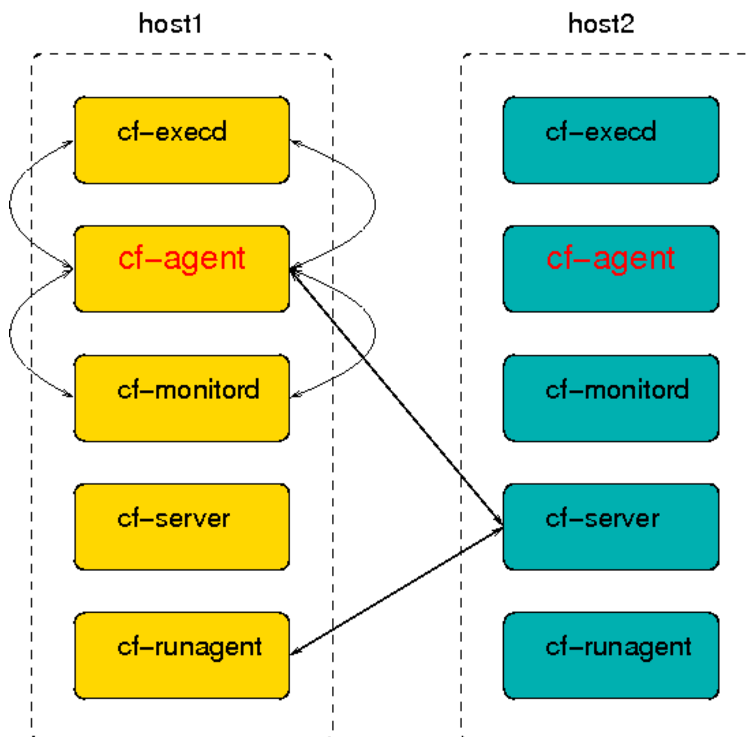
2 The components of CFEngine

CFEngine comprises a number of components. In this chapter we'll consider how to build them and what they are for.

2.1 The players

A CFEngine system is something like an orchestra. It is composed of any number of computers (players), each of which has its own copy of the music and knows what to play. It might or might not have a conductor to help coordinate the individual parts – that's up to you.

CFEngine's software agents run on each individual computer but can communicate if they need to, as depicted the figure below. This means you don't have to arrange risky login credentials to run your network – and if something goes wrong with the communications network, CFEngine is where it needs to be to repair or protect the system during the outage.



If the network is not working, CFEngine just skips these parts and continues with what it can do. It is fault tolerant and opportunistic.

cf-promises
: The promise verifier and compiler. This is used to pre-check a set of configuration promises before attempting to execute.

cf-agent
: This is the instigator of change. The agent is the part of CFEngine that manipulates system resources.

cf-serverd
: The server is able to share files and receive requests to execute existing policy on an individual machine. It is not possible to send (push) new information to CFEngine from outside.

cf-execd
: This is a scheduling daemon (which can either supplement or replace `cron`). It also works as a wrapper, executing and collecting the output of `cf-agent` and E-mailing it if necessary to a system account.

cf-runagent
: This is a helper program that can talk to `cf-serverd` and request that it execute `cf-agent` with its existing policy. It can thus be used to simulate a push of changes to CFEngine hosts, if their policy includes that they check for updates.

cf-report
: This generates summary and other reports in a variety of formats for export or integration with other systems.

cf-know
: This agent can generate an ISO standard Topic Map from a number of promises about system knowledge. It is used for rendering documentation as a 'semantic web'.

2.2 About the CFEngine architecture

This section explains how CFEngine will operate autonomously in a network, under your guidance. If your site is large (thousands of servers) you should spend some time discussing with CFEngine experts how to tune this description to your environment as *scale* requires you to have more infrastructure, and a potentially more complicated configuration. The essence of any CFEngine deployment is the same.

There are four commonly cited phases in managing systems, summarized as follows:

- Build
- Deploy
- Manage
- Audit

These separate phases originate with a model of system management based on transactional changes. CFEngine's conception of management is somewhat different, as transaction processing is not a good model for system management, but we can use this template to see how CFEngine works differently.

Build
: A system is based on a number of decisions and resources that need to be 'built' before they can be implemented. Building the trusted foundations of a system is the key to guiding its development. You don't need to decide every detail, just enough to build trust and predictability into your system.

: In CFEngine, what you build is a template of proposed promises for the machines in an organization such that, if the machines all make and keep these promises,

the system will function seamlessly as planned. This is how it works in a human organization, and this is how is works for computers too.

Deploy Deploying really means implementing the policy that was already decided. In transaction systems, one tries to push out changes one by one, hence 'deploying' the decision. In CFEngine you simply publish your policy (in CFEngine parlance these are 'promise proposals') and the machines see the new proposals and can adjust accordingly. Each machine runs an agent that is capable of implementing policies and maintaining them over time without further assistance.

Manage Once a decision is made, unplanned events will occur. Such incidents traditionally set off alarms and humans rush to make new transactions to repair them. In CFEngine, the autonomous agent manages the system, and you only have to deal with rare events that cannot be dealt with automatically.

Audit In traditional configuration systems, the outcome is far from clear after a one-shot transaction, so one audits the system to determine to discover what actually happened. In CFEngine, changes are not just initiated once, but locally audited and maintained. Decision outcomes are assured by design in CFEngine and maintained automatically, so the main worry is managing conflicting intentions. Users can sit back and examine regular reports of compliance generated by the agents, without having to arrange for new 'roll out' transactions.

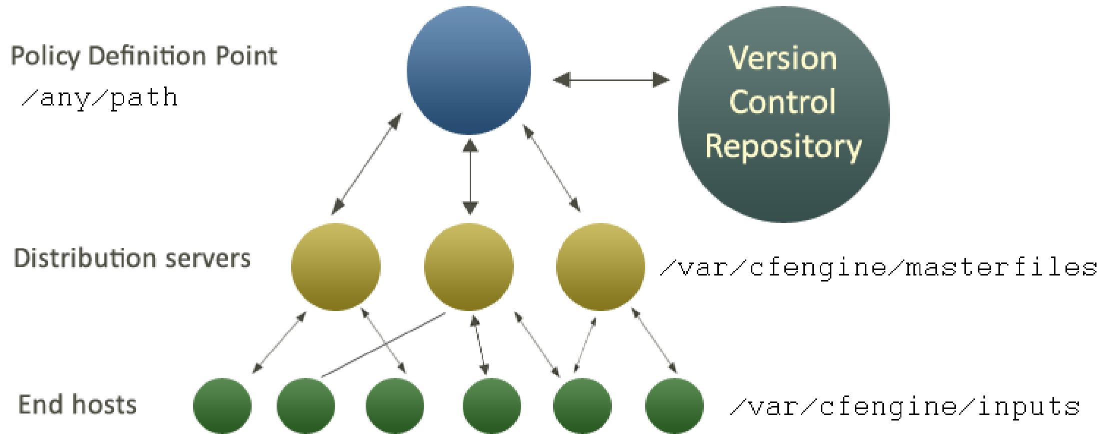
ROLL-OUT and ROLL-BACK? You should not think of CFEngine as a roll-out system, i.e. one that attempts to force out absolute changes and perhaps reverse them in case of error. Roll-out and roll-back are theoretically flawed concepts that only sometimes work in practice. With CFEngine, you publish a sequences of policy revisions, always moving forward (because like it or not, time only goes in one direction). All of the desired-state changes are managed locally by each individual computer, and continuously repaired to ensure on-going compliance with policy.

2.3 The policy decision flow

CFEngine does not make absolute choices for you, like other tools. Almost everything about its behaviour is matter of policy and can be changed. However, a structure for use, like the following, is recommended (see the following figure).

In order to keep operations as simple as possible, CFEngine maintains a private working directory on each machine referred to in documentation as WORKDIR and in policy by the variable `$(sys.workdir)`. By default, this is located at '`/var/cfengine`' or '`C:\var\CFEngine`'. It contains everything CFEngine needs to run.

The figure below shows how decisions flow through the parts of a system.



- It makes sense to have a single point of coordination. Decisions are therefore usually made in a single location (the Policy Definition Point). The history of decisions and changes can be tracked by a version control system of your choice (e.g. Subversion, CVS, etc.).
- Decisions are made by editing CFEngine's policy file '`promises.cf`' (or one of its included sub-files). This process is carried out off-line.
- Once decisions have been formalized and coded, this new policy is copied *manually* (a human decision) to a *decision distribution point*, which by default is located in the directory '`/var/cfengine/masterfiles`' on all policy distribution servers.

  In this introduction, we shall assume that there is only one central policy distribution server, a specially-appointed server which is referred to simple as the `policy server`.
- Every client machine contacts the policy server and downloads these updates. The policy server can be replicated if the number of clients is very large, but we shall assume here that there is only one policy server.

Once a client machine has a copy of the policy, it extracts only those promise proposals that are relevant to it, and implements any changes without human assistance. This is how CFEngine manages change.

WHY DO THIS? CFEngine tries to minimize dependencies by decoupling processes. By following this pull-based architecture, CFEngine will tolerate network outages and will recover from deployment errors easily. By placing the burden of responsibility for decision at the top, and for implementation at the bottom, we avoid needless fragility and keep two independent quality assurance processes apart.

3 Bodies and bundles

To emphasize the fact that CFEngine is not an imperative programming language, and to keep closely to the nomenclature of Promise Theory, CFEngine uses two concepts throughout: bundles and bodies.

3.1 Bodies

Promises are the fundamental statements in CFEngine. Promises are the policy atoms. If there is no promise, nothing happens.

However, promises can become quite complicated and readability becomes an issue, so it is useful to have a way of breaking them down into independent components. The structure of a promise is this:

Promiser This is the object that formally makes the promise. It is always the *affected object*, since objects can only make promises about their own state or behaviour in CFEngine.

Promisee (optional)
This is a possible stakeholder, someone who is interested in the outcome of the promise. It is used as documentation, and it is used for reasoning in the commercial CFEngine product.

Promise body
Everything else about a promise is defined in the body of the promise. We use this word in the sense of 'body of a contract' or the 'body of a document' (like `<body>`) tags in HTML, for example.

A promise body is a list of declarations of the following form:

```
CFEngine_attribute_type => user_defined_value or template
```

3.1.1 Body parts

The CFEngine reserved word `body` is used to define *parameterized templates* for bodies to hide the details of complex promise specifications. For complex body lists, you must fill in a body declaration as an 'attachment' to the promise, e.g.

```
files:

  "/tmp/promiser"        # Promiser

    perms => myexample;  # The body is just one line,
                         # but needs an attachment
```

The attachment is declared like this, with a 'type' that matches the left hand side of the declaration in the promise:

```
body perms myexample
{
mode => "644";
owners => { "mark", "sarah", "angel" };
```

```
groups => { "users", "sysadmins", "mythical_beasts" };
}
```

The structure is this:

```
 promiser

   LVALUE => RVALUE

..

body LVALUE RVALUE
{
LVALUE => RVALUE;
LVALUE => RVALUE;
}
```

Another way of looking at it is this:

```
 promiser

   CFEngine_word => user_defined_value

..

 body CFEngine_word user_defined_value
 {
 CFEngine_word => user_defined_value;
 CFEngine_word => user_defined_value;
 ...
 }
```

Body attachments are required items. You cannot choose to put the attachments in-line. This is a lesson that was learned from CFEngine 2. Readability is quickly lost if too many details are placed in-line.

Blocks: bundles and bodies

- **Bundles** are collections of promises under a single name – like a “subroutine”
- **Bodies** are template macros for simplifying complex sets of promise attributes.
- Declaration:

```
bundle agent myname()
{
files:
..
   perms => yourname,
}
```

```
body perms yourname()
{
mode => 0644

}
```

3.1.2 Control bodies

Some promises in CFEngine are implicit. They are hard-coded into the program. For example, the fact that CFEngine looks for a number of files to read and will execute them in a sequence is hard coded. You cannot change this. However, you can change the behaviour of such promises by setting control parameters. These are formally parts of the ‘promise body’ for these hard-coded promises, so we use the body structure to set them. Each agent has a special body whose name is `control`; e.g.

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

3.2 Bundles

A bundle is a simple concept. A bundle is merely a collection of promises in a ‘sub-routine-like’ container. The purpose of bundles is to allow you greater flexibility to break down the contents of your policies and give them names. Bundles also allow you to re-use promise code by parameterizing it.

Like bodies, bundles also have ‘types’. Bundles belong to the agent that is used to keep the promises in the bundle. So `cf-agent` has bundles declared as

```
bundle agent my_name
{
}
```

The `cf-serverd` program has bundles declared as:

```
bundle server my_name
{
}
```

and so on.

3.2.1 Bundle scope

Variables and classes defined inside bundles are not directly visible outside. All variables in CFEngine are globally accessible, however if you refer to a variable by '`$(unqualified)`', then it is assumed to belong to the current bundle. To access any other (scalar) variable, you must qualify the name using the name of the bundle in which it is defined: '`$(bundle_name.qualified)`'.

Some promise types, like `var`, `classes` may be made by any agent. These are called `common` promises. Bundles of type `common` are special. They may contain common promises. Classes defined in common bundles have global scope.

3.3 A simple syntax pattern

The syntax of CFEngine follows a simple pattern in all cases and has a few simple rules:

- CFEngine built-in words, and identifiers of your choosing (the names of variables, bundles, body templates and classes) may only contain the usual alphanumeric and underscore characters ('`a-zA-Z0-9_`').
- All other 'literal' data must be quoted.
- Declarations of promise bundles in the form:

  ```
  bundle agent-type identifier
  {
  ...
  }
  ```

- Declarations of promise body-parts in the form:

  ```
  body constraint_type template_identifier
  {
  ...
  }
  ```

  matching and expanding on a reference inside a promise of the form '`constraint_type => template_identifier`'.
- CFEngine uses many 'constraint expressions' as part of the body of a promise. These take the form: left-hand-side (cfengine word) '`=>`' right-hand-side (user defined data). This can take several forms:

  ```
  cfengine_word => user_defined_template(parameters)
                   user_defined_template
                   builtin_function()
                   "quoted literal scalar"
                   { list }
  ```

  In each of these cases, the right hand side is a user choice.

Once you have learned this pattern, it will make sense anywhere in the program. The figure below illustrates this pattern. Some words are reserved by CFEngine, and are used as types or

categories for talking about promises. Other words (in blue) are to be defined by you. Look at the examples and try to identify these patterns yourself.

The pattern

Cfengine word		User defined word
What is it?	What's it for?	What's its name?
bundle	agent server	myname
body	perms signals	yourname

4 A simple crash course in concepts

4.1 Rules are promises

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

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

```
type:

classes::

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

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

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

> The promiser is always the object affected by the promise.

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

```
reports:

"hello world";
```

And the simplest commands promise looks like this

```
commands:

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

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

```
# Promise type
files:

# promisor                    -> promisee (no curly braces needed if only one)
"/home/mark/tmp/test_plain" -> "system blue team",
```

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

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

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

4.2 Control promises

Certain promises that CFEngine components make are hard-wired into their code. For example, the promise to email output to an appropriate address, or the promise to wait until a certain time has elapsed before checking a promise again (`ifelapsed`). Although these promises are hard-wired, their behaviour can be changed. In CFEngine, behaviour is always constrained by the promise body. Thus hard-wired behaviour is altered by changing the control body for each. You can find these alterable parameters in the reference manual.

The most important bundle is the `common` bundle, that is read by all components of CFEngine. It contains the list of promise bundles that should be read in and examined for promise suggestions. From the '`promises.cf`' file:

```
body common control
{
bundlesequence  => {
                 "update",
                 "garbage_collection",
                 "main",
                 "cfengine"
                 };

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

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

body agent control
{
# if default runtime is 5 mins we need this for long jobs
```

```
ifelapsed => "15";
}

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

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

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

body executor control

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

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

body reporter control

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

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

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

}

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

body server control
```

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

# Make updates and runs happen in one

cfruncommand          => "$(sys.workdir)/bin/cf-agent -f failsafe.cf &&
$(sys.workdir)/bin/cf-agent";
allowusers            => { "root" };
}
```

4.3 Variables

Variables (or "variable definitions") are also promises – the promise to represent their values. We can write these in any promise bundle. CFEngine recognizes two object types: scalars and lists (lists contain 0 or more objects), as well as three data-types (string, integer and real). Typing in CFEngine is dynamic, as in Perl and other scripting languages. Thus variables of any data-type may be used as strings.

4.3.1 Scalar variables

Scalar variables hold a single value. The are declared as follows:

```
bundle <type> name
{
vars:

"my_scalar" string => "String contents...";
    "my_int" int    => "1234";
   "my_real" real   => "567.89";

}
```

The '`<type>`' indicates that any kind of bundle applies here. Scalar variables are referenced by '`$(name)`' (or '`${name}`') and they represent a single value at a time.

- Scalars that are written without a context, e.g. '`$(myvar)`' are local to the current bundle.
- Scalars are globally available everywhere provided one uses the context to verify them e.g. '`$(context.myvar)`' may be written to access the variable 'myvar' in bundle 'context'.

4.3.2 List variables

List variables hold several values. The are declared as follows:

```
bundle <type> name
{
vars:

   "my_slist" slist => { "list", "of", "strings" };
   "my_ilist" ilist => { "1234", "5678" };
   "my_rlist" rlist => { "567.89" };
```

```
}
```

An entire list is referred to with the at symbol '@', but it does not usually make sense to use this reference in a string. For instance

```
reports:

  cfengine_3::

     "My list is @(my_slist)";
```

means nothing and cannot be expanded (it does not generate an error, but instead inserts the text @(my_slist) into the string); but if we use the scalar reference to a list variable, CFEngine will iterate over the values in the list essentially making this into a list of promises.
To summarize:

- Scalar references to *local* list variables imply iteration, e.g. suppose we have local list variable '`@(list)`', then the scalar '`$(list)`' implies an iteration over every value of the list.
- Lists can be passed in their entirety in any context where a list is expected as '`@(list)`'., e.g.

  ```
  vars:

  "longlist" slist => { @(shortlist), "plus", "plus" };

  "shortlist" slist => { "you", "me" };
  ```

  The declaration order does not matter – CFEngine will execute the promise to assign the variable '`@(shortlist)`' before the promise to assign the variable '`@(longlist)`'.
- Only local lists can be expanded directly. Thus '`$(list)`' can be expanded but not '`$(context.list)`'. Global list references have to be mapped into a local context if you want to use them for iteration. See the reference manual for more information.

4.3.3 Associative arrays

Associative array variables also hold several values. The are declared as follows:

```
bundle <type> name
{
vars:

   "switches[mellow]" int => "1";
   "switches[relaxed]" int => "1";
   "off_keys" slist => { "red", "grouchy", "coarse", "febrile" };
   "switches[$(off_keys)]" int => "0";

}
```

See the reference manual for information on the '`getindices`' function and other details of associative arrays.

4.4 Decisions

CFEngine decisions are made behind the scenes and the results of certain true/false propositions are cached in Booleans referred to as 'classes'. There are no if-then-else statements in CFEngine; all decisions are made with classes.

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

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

- The name of an operating system architecture e.g. `ultrix`, `sun4`, etc.
- The unqualified name of a particular host. If your system returns a fully qualified domain name for your host, CFEngine truncates it at the first dot. Note: `www.sales.company.com` and `www.research.company.com` have the same unqualified name – `www`.
- 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, current time zone (in the form `Hr00, Hr01 ... Hr23`).
- An hour of the day GMT (in the form `GMT_Hr00, GMT_Hr01 ... GMT_Hr23`). This is consistent the world over, in case you need virtual simultaneity of change coordination.
- 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`)
- The quarter-hour (in the form `Q1, Q2, Q3, Q4`).
- A day of the month (in the form `Day1, Day2, ... Day31`).
- A month (in the form `January, February, ... December`).
- A year (in the form `Yr1997, Yr2004`).
- A shift in `Night,Morning,Afternoon,Evening`, which fall into six hour blocks starting at 00:00 hours.
- A 'lifecycle index', which is the year number modulo 3 (used in long term resource memory).
- An arbitrary user-defined string.
- The IP address octets of any active interface (in the form `ipv4_192_0_0_1`, `ipv4_192_0_0`, `ipv4_192_0`, `ipv4_192`).

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

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

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

User-defined or soft classes are defined in bundles. Bundles of type `common` yield classes that are global in scope, whereas in all other bundle types classes are local. Soft classes are

evaluated when the bundle is evaluated. They can be based on test functions or simply from other classes:

```
bundle agent myclasses
{
classes:

"solinus" expression => "linux||solaris";

# List form useful for including functions

"alt_class" or => { "linux", "solaris", fileexists("/etc/fstab") };

"oth_class" and => { fileexists("/etc/shadow"), fileexists("/etc/
passwd") };

reports:

alt_class::

   # This will only report "Boo!" on linux, solaris, or any system
   # on which the file /etc/fstab exists
   "Boo!";
}
```

Classes may be combined with the operators listed here in order from highest to lowest precedence:

'()' The parenthesis group operator.

'!' The NOT operator.

'.' The AND operator.

'&' The AND operator (alternative).

'|' The OR operator.

'||' The OR operator (alternative).

So the following expression would be only true on Mondays or Wednesdays from 2:00pm to 2:59pm on Windows XP systems:

```
(Monday|Wednesday).Hr14.WinXP::
```

Consider the following more advanced example. Promises in bundles of type 'common' are global in scope – all other promises are local to the scope of their bundle.

```
body common control
{
```

```
bundlesequence => { "g","ls_1", "ls_2" };
}

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

bundle common g
{
classes:

# The promise "zero" is always satisfied , and is global in scope
"zero" expression => "any";

}

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

bundle agent ls_1
{
classes:

# The promise "one" is always satisfied , and is local in scope to ls_1
"one" expression => "any";
}

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

bundle agent ls_2
{
classes:

# The promise "two" is always satisfied , and is local in scope to ls_2
"two" expression => "any";

reports:

zero.!one.two::

  # This report @b{will} be generated
  "Success";
}
```

Here we see that class 'zero' is global while classes 'one' and 'two' are local. The report 'Success' result is therefore true because only 'zero' and 'two' are in scope in the 'ls_2' bundle (and the class expression for bundle 'ls_2' requires that both 'zero' and 'two' be true and that 'one' not be true).

4.5 Loops

If you are looking for loops in CFEngine then we need to reprogram you a little, as you are thinking like a programmer! CFEngine is not a programming language that is meant to give you low level control, but rather a set of declarations that embody processes. It's the difference between the gears on a bicycle and the automated transmission in a transporter.

Loops are executed implicitly in CFEngine, but there is no visible mechanism for it – because that would steal attention from the intention of the promises. The way to express them is through lists.

Loops are really a way to iterate a variable over a list. Try the following.

```
body common control

{
bundlesequence  => { "example" };
}

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

bundle agent example

{
vars:

# This is a list

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

# This is an associative array

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

reports:

cfengine_3::

"$(component) is $(array[$(component)])";

}
```

The output looks something like this:

```
/usr/local/sbin/cf-agent -f ./unit_loops.cf -K

R: cf-monitord is The monitor
```

```
R: cf-serverd is The server
R: cf-execd is The executor, not executionist
```

You see from this that, if we refer to a list variable using the scalar reference operator '$()', CFEngine interprets this to mean "please iterate over all values of the list". Thus, we have effectively a 'foreach' loop, without the attendant syntax.

4.6 The main promise types

The following promise types may be used in any bundle:

`vars` A promise to be a variable, representing a value.

`classes` A promise to be a class representing a state of the system.

`reports` A promise to report a message.

These additional promise types may be used only in agent bundles

`commands` A promise to execute a command.

`databases`
A promise to configure a database.

`files` A promise to configure a file, including its existence, attributes and contents.

`interfaces`
A promise to configure a network interface.

`methods` A promise to take on a whole bundle of other promises.

`packages` A promise to install a package.

`storage` A promise to verify attached storage.

These promise types belong to other components:

`access` A promise to grant or deny access to file objects in `cf-serverd`.

`measurements`
A promise to measure or sample data from the system, for monitoring or reporting in `cf-monitord` (CFEngine Nova and above).

`roles` A promise to allow certain users to activate certain classes when executing `cf-agent` remotely, in `cf-serverd`.

`topics` A promise to associate knowledge with a name, and possibly other topics, in `cf-know`.

`occurrences`
A promise to point or refer to a knowledge resource, in `cf-know`.

4.7 Test a promise?

If you are impatient to get hands-on experience, now might be a good time to take a break from Concepts and have a look at the getting started guide (http://cfengine.com/manuals/cf3-getstarted.html. Still, since knowledge management is an integral part of CFEngine, we strongly recommend to read the following section on this very issue sooner rather than later.

5 Knowledge Management

A unique aspect of CFEngine, that is fully developed in the commercial editions of the software, its ability to enable integrated knowledge management as part of an automation process, and to use its configuration technology as a 'semantic' documentation engine.

Knowledge management is the challenge of our times. Organizations frequently waste significant effort re-learning old lessons because they have not been documented and entered into posterity. Now you can alleviate this problem with some simple rules of thumb and even build sophisticated index-databases of documents.

5.1 Promises and Knowledge

The learning curve for configuration management systems has been the brunt of frequent criticism over the years. Users are expected to either confront the informational complexity of systems at a detailed level, or abandon the idea of fine control altogether. This has led either to information overload or over-simplification. The ability to cope with information complexity is therefore fundamental to IT management

CFEngine introduced the *promise model* for configuration in order to flatten out this learning curve. It can lead to simplifications in use, because a lot of the thinking has been done already and is encapsulated into the model. One of its special properties is that it is both a model for system behaviour and a model for knowledge representation (this is what declarative languages seek to be, of course). More specifically, it incorporated a subset of the ISO standard for 'Topic Maps', an open technology for semantic indexing of information resources. By bringing together these two technologies (which are highly compatible), we end up with a seamless front-end for sewing together and browsing system information.

Knowledge management is a field of research in its own right, and it covers a multitude of issues both human and technological. Most would agree that knowledge is composed of facts and relationships and that there is a need both for clear definitions and semantic context to interpret knowledge properly; but how do we attach *meaning* to raw information without ambiguity?

Knowledge has quite a lot in common with configuration: what after all is knowledge but a configuration of ideas in our minds, or on some representation medium (paper, silicon etc). It is a coded pattern, preferably one that we can agree on and share with others. Both knowledge and configuration management are about describing patterns. A simple knowledge model can be used to represent a policy or configuration; conversely, a simple model of policy configuration can manufacture a knowledge structure just as it might manufacture a filesystem or a set of services.

5.2 The basics of knowledge

Knowledge only truly begins when we write things down:

- The act of formulating something in writing brings a discipline of thought than often lends clarity to an idea.
- You never confront an idea fully until you try to put it into language.
- Any written record that is kept allows others to read it and pass on the knowledge.

The trouble is that writing is something people don't like to do, and few are very good at. To an engineer, it can feel like a waste of time, especially during a busy day, to break off from the doing to write about the doing. Also, writing requires a spurt of creative thinking and engineers are often more comfortable with manipulating technical patterns and notations than writing fluent linguistic formulations that seem overtly long-winded.

CFEngine tries to bridge this gap by making documentation simple and part of the technical configuration. CFEngine's knowledge agent then uses AI and network science algorithms to construct a readable documentation from these technical annotations. It can do this because a lot of thought has already gone into the meaning of the promise model.

5.3 Annotating promises

The beginning of knowledge is to annotate the technical specifications. Remember that the point of a promise is to convey an *intention*. When writing promises, get into the habit of giving every promise a comment that explains its intention. Also, expect to give special promises *handles*, or helpful labels that can be used to refer to them by in other promise statements. A handle could be something dumb like 'xyz', but you should try to use more meaningful titles to help make references clear.

```
files:

"/var/cfengine/inputs"

     handle => "update_policy",
    comment => "Update the CFEngine input files from the policy server",
      perms => system("600"),
  copy_from => rcp("$(master_location)","$(policy_server)"),
depth_search => recurse("inf"),
file_select => input_files,
     action => immediate;
```

If a promise affects another promise in some way, you can make the affected one promise one of the promisees, like this:

```
access:

"/master/CFEngine/inputs" -> { "update_policy", "other_promisee" },

handle  => "serve_updates",
  admit   => { "217.77.34.*" };
```

Conversely, if a promise might depend on another in some (even indirect) way, document this too.

```
files:

"/var/cfengine/inputs"

      handle => "update_policy",
     comment => "Update the CFEngine input files from the policy
server",
  depends_on => { "serve_updates" },
       perms => system("600"),
   copy_from => rcp("$(master_location)","$(policy_server)"),
depth_search => recurse("inf"),
file_select => input_files,
      action => immediate;
```

This use of annotation is the first level of documentation in CFEngine. The annotations are used internally by CFEngine to provide meaningful error messages with context and to compute dependencies that reveal the existence of process chains. These can be turned into a topic map for browsing the policy relationships is a web browser, using `cf-know`.

> The CFEngine Knowledge Map is only available in commercial editions of the software, where the necessary support to set up and maintain this technology can be provided.

5.4 A promise model of topic maps

CFEngine's model of promises can also be used to promise information and its relevance in different contexts. The Knowledge agent `cf-know` understands three kinds of promise.

`topics:` A topic is merely a string that can be associated with another string. It represents a 'subject to be talked about'. Like other promise types, you can use contexts, which are formed from other topics expressions to limit the scope of the current topic promise.

`things:` Things are a simplified interface to topics, that were introduced to make it easier for users to contribute knowledge about more concrete 'things', or less abstract

ideas. A challenge with knowledge management is the abstract and technical nature of the models one must use to represent it. Things attempt to make that task easier.

`occurrences:`
An occurrence is a reference to a document or a piece of text that actually represents knowledge content about the topic concerned. Occurrences are generally URLs or strings explaining things or topics.

5.5 What topic maps offer

CFEngine is capable of automating the documentation of a policy, using basic annotations provided above, as a knowledge map. They require very little effort from the user. If you are using the Community Edition of CFEngine, you can develop a topic map, but we do not support the backend technology without a commercial license. In either case, once you become familiar with the use of Topic Maps, you will want to extend your knowledge manually to incorporate things like:

- Local (high level) policy documents
- Related databases, such as CMDBs

So let us spend a while showing how to encode knowledge in topic maps using `cf-know`.

The kind of result you can expect is shown in the pictures below. The example figures show typical pages generated by the knowledge agent `cf-know`. The first of these shows how we use the technology to power the web knowledge base in the commercial CFEngine product.

In this use, all of the data are based on documentation for the CFEngine software, and most of the relationships are manually entered.

For a second example, consider how CFEngine can generate such a knowledge map analysis of its own configuration (self-analysis). The data in the images below describe the CFEngine configuration promises. One such page is generated, for instance, for each policy promise, and pages are generated for reports from different computers etc. You can also create you own 'topic pages' for any local (enterprise) information that you have.

In this example, the promise has been given the promise-handle `update_policy`, and the associations and the lower graph shows how this promise relates to other promises through its documented dependencies (these are documented from the promisees and `depends_on` attributes of other promises.).

The example page shows two figures, one above the other. The upper figure shows the thirty nearest topics (of any kind) that are related to this one. Here the relationships are unspecific. This diagram can reveal pathways to related information that are often unexpected, and illustrates relationships that broaden one's understanding of the place the current promise occupies within the whole.

Although the graphical illustrations are just renderings of semantic associations shown more fully in text, they are useful for visualizing several levels of depth in the associative network. This can be surprisingly useful for brainstorming and reasoning alike. In particular, one can see the other promises that could be affected if we were to make a change to the current promise. Such impact analyses can be crucial to planning change and release management of policy.

A knowledge base is a slightly improved implementation of a Topic Map which is an ISO standard technology. A topic map works like an index that can point to many different kinds of external resources, and may contain simple text and images internally. So you use it to bind together documents of any kind. A CFEngine knowledge base is not a new document format, it is an overlay map that joins ideas and resources together, and displays relationships.

5.6 The nuts and bolts of topic maps

5.6.1 Topic map definitions

Topic maps are really electronic indices, but they form and work like webs. A topic is the technical representation of a 'subject', i.e. anything you might want to discuss, abstract or physical e.g. an item of 'abstract knowledge', which probably has a number of concrete exemplars. It might be a person, a machine, a quality, etc.

Topics can be classified into boxes called *topic-types* so that related things can be collated and unrelated things can be separated, e.g. types allow us to distinguish between `rmdir` the Unix utility and `rmdir` the Unix system-call.

Each typed topic can further point to a number of references or exemplars called *occurrences*. For instance, an occurrence of the topic 'computer' might include books, web documents, database entries, physical manifestations, or any other information. An occurrence is a reference that exemplifies the abstract topic. Occurrence references are like the page numbers in an index.

A book index typically has 'see also' references which point from one topic to another. Topic Maps allow one to define any kind of *association* between topics. Unlike an ordinary index, a topic map has a rich (potentially infinite) variety of cross reference types. For instance,

```
topic_1 ‘‘is a kind of’’ topic_2
topic_1 ‘‘is improved by’’ topic 2
topic_1 ‘‘solves the problem of’’ topic_2
```

The topic map model thus has three levels of containers:

Contexts
: The box into which we classify a topic to disambiguate different topics with the same name ('in the context of')[1].

Topics/Things
: The representation of a subject (an index term).

Occurrence Types
: A term that explains how an actual document occurrence relates to the topic is claims to say something about. e.g. (tutorial, manual, or example, definition, photo-album etc).

Occurrences
: Specific information resources: these are pointers to the actual documents that we want to read (like page numbers in an index).

Contexts map conveniently into CFEngine classes. Topics map conveniently into promisers. Occurrences also map to promisers of a different type. These three label different levels of

[1] Here, CFEngine differs from the topic map standard in allowing contexts to be overlapping sets, rather than mutually exclusive 'types'. CFEngine is guided by Promise Theory in this respect in order to enable distribtued cooperation and the development of a free and emergent ontology.

granularity of meaning. Contexts represent a set of topics that might be relevant, which in turn encompass a set of occurrences of resources that contain actual information about the topics in that context. The primacy of topics in this stems from their ability to form networks by *association*.

The classic approach to information modelling is to build a hierarchical decomposition of non-overlapping objects. Data are manipulated into non-overlapping containers which often prove to be overly restrictive. Topic maps allow us to avoid the kinds of mistakes that have led to monstrosities like the Common Information Model (CIM) with its *thousands* of strictly non-overlapping type categories.

Each topic allows us to effectively 'shine a light' onto the occurrences of information that highlight the concepts pertinent to the topic somehow.

5.7 Example of topics promises

You can use `cf-know` to render a topic map either as text (for command line use) or as HTML (for web rendering). We begin with the text rendering as it requires less infrastructure. You will just need a database.

Try typing in the following knowledge promises:

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

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

bundle knowledge tm
{
topics:


"server" comment => "Common name for a computer in a desktop";
"desktop" comment => "Common name for a computer for end users";

programs:: # context of programs

"httpd" comment => "A web service process";
"named" comment => "A name service process";

services::

"WWW" comment => "World Wide Web service",
      association => a("is implemented by",
                       "programs::httpd",
                       "implements");

 # if we don't specificy a context, it is "any"

"WWW" association => a("looks up addresses with",
                       "named",
                       "serves addresses to");

occurrences:
```

```
httpd::
   "http://www.apache.org"
     represents => { "website" };

}

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

body association a(f,name,b)

{
forward_relationship => "$(f)";
backward_relationship => "$(b)";
associates => { $(name) };
}
```

The simplified things interface is similar, but uses fixed relations:

```
bundle knowledge company_knowledge
{
things:
 regions::

   "EMEA"     comment => "Europe, The Middle-East and Africa";
   "APAC"     comment => "Asia and the Pacific countries";

 countries::
   "UK"            synonyms => { "Great Britain" },
              is_located_in => { "EMEA", "Europe" };

   "Netherlands"   synonyms => { "Holland" },
              is_located_in => { "EMEA", "Europe" };

   "Singapore"     is_located_in => { "APAC", "Asia" };

 locations::
   "London_1"    is_located_in => { "London", "UK" };
   "New_Jersey"  is_located_in => { "USA" };

 networks::

  "192.23.45.0/24"         comment => "Secure network, zone 0. Single octet for corporate o
                   is_connected_to => { "oslo-hub-123" };
```

5.7.1 Analysing and indexing the policy

CFEngine can analyze the promises you have made, index and cross reference them using the command:

```
# cf-promises -r
```

Normally, the default policy in Nova or Constellation will perform this command each time the policy is changed.

5.7.2 cf-know

CFEngine's knowledge agent `cf-know` allows you to make promises about knowledge and its inter-relationships. It is not specifically a generic topic map language: rather it provides a powerful configuration language for managing a knowledge base that can be compiled into a topic map.

To build a topic map from a set of knowledge promises in '`knowledge.cf`', you would write:

```
# cf-know -b -f ./knowledge.cf
```

The syntax of this file is hinted at below. The full ISO standard topic map model is too rich to be a useful tool for system knowledge management. However, this is where powerful configuration management can help to simplify the process: encoding a topic map is a complex problem in configuration, which is exactly what CFEngine is for. CFEngine's topic map promises have the following form:

```
bundle knowledge example
{
topics:

topic_type_context::                              # canonical container

"Topic name"                                    # short topic name

      comment => "Use this for a longer description",
  association => a("forward assoc to","Other topic","backward assoc");

  "Other topic";

occurrences:

Topic_name::                                      # Topic

  "http://www.example.org/document.xyz"        # URI to instance

    represents => { "Definition", "Tutorial"}; # sub-types
}
```

The association body templates look like this:

```
body association a(f,name,b)
{
forward_relationship => "$(f)";
backward_relationship => "$(b)";
associates => { $(name) };
}
```

Promise theory adds a clear structure to the topic map ontology, which is highly beneficial as experience shows that weak conceptual models lead to poor knowledge maps.

5.8 Modelling configuration promises as topic maps

We can model topic maps as promises within CFEngine; the question then remains as to how to use topic maps to model configurations so that CFEngine users can navigate the documented promises using a web browser and be able to see all of the relationships between otherwise isolated and fragmentary rules. This will form the basis of a semantic Configuration Management Database (sCMDB) for the CFEngine software. The key to making these ends meet is to see the configuration of the topic map as a number f promises made in the abstract space of topics and the turning each promise into a meta-promise that models the configuration as a topic with attendant associations. Consider the following CFEngine promise.

```
bundle agent update
{
files:

any::

‘‘/var/cfengine/inputs’’ -> { ‘‘policy_team’’, ’’dependent’’ },

          comment => ‘‘Check policy updates from source’’,
            perms => true,
             mode => 600,
        copy_from => true,
      copy_source => /policy/masterfiles,
          compare => digest,
     depth_search => true,
            depth => inf,
         ifelapsed => 1;

}
```

This system configuration promise can be mapped by CFEngine into a number of other promise proposals intended for the `cf-know` agent. Suppressing some of the details, we have:

```
type_files::

"/var/cfengine/inputs"
    association => a("promise made in bundle","update","bundle
contains promise");
"/var/cfengine/inputs"
    association => a("specifies body type","perms","is specified in");
"/var/cfengine/inputs"
    association => a("specifies body type","mode","is specified in");
"/var/cfengine/inputs"
    association => a("specifies body type","copy_from","is specified
in");

# etc ...
```

```
occurrences:

_var_CFEngine_inputs::

  "promise_output_common.html#promise__var_CFEngine_inputs_update_cf_13"
     represents => { "promise definition" };
```

Note that in this mapping, the actual promise (viewed as a real world entity) is an occurrence of the topic 'promise'; at the same time each promise could be discussed as a different topic allowing meta-modelling of the entity-relation model in the real-world data. Conversely the topics themselves become configuration items or 'promisers' in the promise model. The effect is to create a navigable semantic web for traversing the policy; this documents the structure and intention of the policy using a small ontology of standard concepts and can be extended indefinitely by human domain experts.

6 More...

You will find extensive help, examples and documentation as part of the commercial CFEngine support. Visit the website http://www.cfengine.com for more details.

Need help getting started?

- CFEngine Concepts: http://cfengine.com/manuals/cf3-installation.html
- Get started, first promises: http://cfengine.com/manuals/cf3-getstarted.html

For a complete overview:

- Tutorial: http://cfengine.com/manuals/cf3-tutorial.html
- Reference manual: http://cfengine.com/manuals/cf3-reference.html