

PROMISES AND CFENGINE

A WORKING SPECIFICATION FOR CFENGINE 3

Mark Burgess and Eileen Frisch

November 29, 2005

Chapter 1

Aims of this document

1.1 Introduction

This is a working document whose function is that of a Request For Comment.

This document is a self-contained discussion about cfengine, its structure, operation and syntax. It is a critical document, based on a theoretical model called promise theory. Its aim is to discover a rational framework for discussing what has been successful and unsuccessful about cfengine 1 and 2, with an eye to the redesign cfengine in version 3. Versions 1 and 2 of cfengine have appealed to a number theoretical results as they have evolved, in an *ad hoc* way, but there has never been a unifying theory to appeal to at a general level.

The document addresses the intelligent reader and tries to be sufficiently pedagogical in its applications of theory so that the widest possible audience can understand and participate in the discussion. However, it demands some work on the part of the reader to understand the material.

The document is a working document, so it will be subject to frequent revision. The ultimate aim is to have this document define cfengine 3, as a standard, in appropriate detail for implementation.

1.2 Some theoretical results related to cfengine

The following results are known, prior to promise theory.

1.2.1 Primary theoretical foundations

It is important, when considering cfengine 3, to not lose focus on the fundamental theoretical constructs and foundations that have made the current version useful and effective (see, e.g., refs. [1, 2, 3] and especially the recent discussion in ref. [4]). Of these, the most crucial are the notions of convergence and autonomous agents.

Convergence is central to the design and functioning of cfengine. Cfengine's goal is to bring the system to a state of stable equilibrium and to thereafter maintain it in that state. Used appropriately, cfengine achieves this goal by carrying out a set of convergent operations derived from the statements in its configuration files and applied iteratively as required (recall that such an operation is one that can be repeated without further effect in/from the equilibrium state). Ref. [4] demonstrates that this approach is theoretically rigorous and (mathematically) well-defined.

Cfengine is used within a network of *autonomous agents*: the project provides a distributed framework for achieving convergence on disparate systems. Central to this approach is the notion that policies are applied locally within individual systems, and policy decisions—more precisely, (technical) decisions guided by policy—are made locally, rather than being imposed from outside.

While cfengine can—and is—used in ways which are slightly to completely at odds with these ideas and precepts, such uses (inevitably) run “against the grain” with respect to accepted best practices in computer management and maintenance. Future versions of cfengine, and cfengine 3 in particular, will attempt to provide system administrators with a framework and tool suite which will both enable and encourage designing and applying/implementing policy via convergent operations and a local-centered focus.

1.2.2 Closures

Couch and coworkers have introduced and begun to elaborate on a system configuration paradigm revolving around their concept of em closures (refs. [5, 6, 7]). A closure is an automated configuration management systems. Closures have hierarchical structures consisting of both a high-level language for describing desired behavior, a hardware or software based restraint infrastructure (entities) capable of both enforcing configuration consistency and policy adherence and preventing modification to subsystems by any mechanisms other than those it provides. As the latter implies, closures also include software tools for interacting with such subsystems at a very high, (almost) merely conceptual level. At the present time, closures are add-ons which attempt to assume total control over an existing system facility (although, theoretically, entirely new closures could be built from scratch).

In simple terms, a closure for a simple traditional UNIX service governed in the usual way by a single text-based configuration file is a system which completely encompasses all of the aspects of configuring and managing that daemon. A closure will handle all interaction with the daemon process and with its configuration file. It will provide a tool which a system administrator must use in order to make modifications to the configuration file which ensures that changes are both correct and compliant with policy. The closure will also have the ability to start and stop the daemon when appropriate, signal it to reload its configuration file, monitor its status and restart it when it dies or hangs, and the like.

Clearly, such an approach is very different from the one employed by cfengine. While there are several points at which a critique of closures is appropriate (most notably, in the assumptions that configuration determines system behavior in toto and the work’s current focus on static systems), such a discussion is beyond the scope of this paper. In any case, despite such reservations, closures are an appropriate approach to many sorts of system administration tasks, and cfengine can be used to build them. In fact, future versions ought to make doing so easy and reliable.

1.2.3 Monitoring and change

In ref. [6], Couch et. al. show that consistency and closure of system changes can only be achieved if one does not mix monitoring and scheduling of changes. Observations should not be interleaved with changes (generally preceding them within a given convergence iteration). This has implications for the way a configuration tool operates, and closures are formed/built from its basic elements.

1.2.4 Operator ordering

Discussions of ordering of operations have been presented in, for example, refs. [8, 9]. The most important result from this ongoing area of intensive research is that ordering of operations is not important provided that operations are orthogonal. When this is the case,

simple iteration results in ordering conflict resolution provided that the individual operators are themselves convergent.

A simple solution is to adopt a “normal ordering” in which all creation operators are placed to the left of (earlier than) change operators or deletion operators. While this might be a disadvantage from a resource perspective, as it is often judicious to clear space for further operations or neutralize a threatening object quickly, correctness suggests normal ordering.

1.3 Problems with the current cfengine syntax

There are many inconsistencies in cfengine syntax that must also be addressed in cfengine 3. There also features which are not present in the current version but which important or valuable or merely helpful to current and future users. Finally, there are features which are probably no longer needed. This section considers such items.

1.3.1 Current syntax inconsistencies

The most problematical of the current inconsistencies in cfengine syntax is that the object of change is not always the primary object of syntax, e.g.:

Example 1

```
files:  
  
    /file/object  mode=644  action=fixall
```

In this example, the type of promise is “files:”, the primary object is “/file/object” and various constraints on the object’s attributes, and operations actions are specified after it. Similarly

Example 2

```
links:  
  
    /link/object -> /object/pointed/to
```

give primacy to the object being changed. The syntax thus has the form of a lambda calculus:

(type object) arguments/attributes

However, now consider copy:

Example 3

```
copy:  
  
    /source/object  dest=/destination/object
```

In this case, the object that is being constrained is, in fact, the destination object. A more consistent approach would have been to reverse this syntax:

Example 4

better_copy:

```
/destination/object source=/source/object
```

Cfengine functions, called “methods” also have different syntax because they accept parameters. Normally the response function has no parameters, or implicit parameters.

Example 5

methods:

```
BackupLapTop("/home/mark/LapTop",  
             "/iu/nexus/ud/mark/ZZBackup",  
             $(global.ipv4[eth0]), "mark")
```

```
action=cf.laptopbackup  
server=nexus.iu.hio.no
```

```
returnclasses=done  
returnvars=ret  
ifelapsed=240  
expireafter=480
```

Once again, it is not the object of primary interest that is the focus of this operation. One could perhaps have written this as

files:

```
/home/mark/LapTop
```

```
action=Backup($this,/iu/nexus/ud/mark/ZZBackup,  
              $(global.ipv4[eth0]), "mark")
```

Finally, the syntax for editfiles and filtering introduces completely different ideas about conditional execution, iteration etc.

Example 6

```
{ /etc/ssh/sshd_config  
  
  AppendIfNoSuchLine "Banner /etc/ssh/banner"  
  
  ReplaceAll "X11Forwarding.*no" With "X11Forwarding yes"  
  
  BeginGroupIfNoLineMatching "bla.*"  
  
    AppendLine "bla xxx"  
  
  EndGroup  
  
  DefineClasses "sshd_hup"  
}
```

The promise-test-act cycle of cfengine is normally synonymous with rule-classification-action. Here we have “BeginIfClass” instead of “class::”. The use of a begin-end parenthesis is itself a break with the state-based transition form of the remainder of syntax.

Currently cfengine mixes the setting of control attributes with the definition of user variables. There is no clean understanding or distinction between lists, scalars and arrays.

1.3.2 New capabilities

Cfengine is a child of its time. While it provides a very rich set of operations which can address a very large variety of system administration functions and issues, there are newer, important areas which are currently outside of its scope. For example:

- System services understood as such (rather than as merely generic processes).
- Entries (records) within databases/directories (e.g., LDAP-based directories, the Windows registry, and the like).
- Devices such as printers, again understood as such.
- Expansion of anomaly detection entities to include items such as remote service availability as well as the current threshold-based performance parameters.

In addition, there are desirable changes which would give cfengine 3 all of the capabilities of other, legacy software, giving the system administrator the option of replacing them with cfengine (“one stop shopping”). For example, computing and storing multiple cryptographic signatures would strength this intrusion detection feature (cf. Tripwire ==need refs==). Similarly, cfengine’s file rotation feature should be reconsidered to ensure that it has all of the functionality found in logrotate (actually, even in its present form, it is in some ways already superior to the latter).

1.3.3 Vestigial features

Certain features found in cfengine 2 are probably no longer needed and ought to be removed. For example, the file system mounting actions are seldom used, and the multitude of different variations could be combined or even eliminated.

1.3.4 Other syntax issues

Functions are not presently allowed in class expressions. This restriction can be lifted, with some caveats on semantics about evaluation (e.g., at parse-time).

Example 7

```
any::  
    <rule>  
  
constant1|constant2::  
    <rule>  
  
function("arg")|constant::  
    <rule>
```

1.3.5 Encouraging convergent use of cfengine

Whenever possible, cfengine syntax should encourage and foster convergent use (and thinking). Suggestions along these lines will appear in the proposed syntax section later in this document.

Chapter 2

Basic promise theory

Promise theory was invented to discuss the issues surrounding autonomous operation, and voluntary cooperation. Unlike other modeling techniques, like Petri Nets or UML, promise theory is not about the stepwise development of a device. It is not about protocol modeling, rather it is about *equilibria*, i.e. how to describe steady state behavior that has some underlying dynamics.

Promise theory begins with the idea of completely autonomous agents that interact through the promises they make to one another. It is therefore particularly well-suited to modeling cfengine.

Cfengine maintains the principle of *host autonomy*. Many existing systems and descriptions of configuration management assume a centralized authority to change all the parts of the system. This has a number of disadvantages. Our aim here is to begin in the opposite manner, assuming *no centralization* of authority, and then seeing how collaborative structures emerge by free choice.

2.1 Motivation

The way to secure a clear and consistent picture of policy, in complex environments, is through the use of formal methods. But what formalism do we have to express the necessary issues?

Example 8 Consider a set of files and directories, which we want to have certain permissions. One has a notion of policy as a specification of the permission attributes of these files. Policy suggests that we should group items by their attributes. The existing system has its own idea of grouping structures: directories. File attributes flow from those of their parent directory in the absence of specific overrides.

A simple example of this is the following:

ACL1:	ACL2:
1. READ-WRITE /directory	1. READ-ONLY /directory/file
2. READ-ONLY /directory/file	2. READ-WRITE /directory

Without clear semantics (e.g., first rule wins), there is an ordering ambiguity. The two rules overlap in their application to “file” because we have used a description based on overriding the collection of objects implicitly in “directory.”

In a real system, a directory grouping is the simplest way to refer to this collection of objects. However, this is not the correct classification of the attributes: there is a conflict of interest.

Analogous issues arise from incomplete policy specifications. Consider the list of nameservers in `/etc/resolv.conf`. If there is a policy which ensures that two specific nameservers are always the first two listed in this file, an ambiguous situation arises in the (presumably rare) circumstance when both of these are unavailable. Will no nameserver be accessible in this case? Alternatively, will whatever legacy data existed in the configuration file prior to its (policy-based) modification enable an essentially random choice? If the latter, and the specified system is in fact a nameserver, how will the return to the “proper” nameserver be accomplished?

Finally, consider what could be called a “one-sided policy.” Suppose a server is configured to stream backup data to a tape drive on a backup server, and that the backup server accepts such data. It is possible for the sending server to either exceed the incoming bandwidth of the backup server or to degrade performance for other processes/subsystems on the backup server by virtue of its actions. While the sender server is clearly operating as an autonomous agent in this case, the lack of reciprocity in this policy has a detrimental effect on the backup server. Clearly, something more than mere acceptance or rejection is required on the part of the backup server.

How can we solve these kinds of problems?

2.2 Policy with autonomy

By a policy we mean the ability to assert arbitrary constraints of the behavior of objects and agents in a system. The most general kind of system one can construct is a collection of objects, each with its own attributes, and each with its own policy. A policy can also be quite general: e.g. policy about behavior, policy about configuration, or policy about interactions with others.

In a network of *autonomous* systems, an agent is only concerned with assertions about its own policy; no external agent can tell it what to do, without its consent. This is the crucial difference between autonomy and centralized management, and it will be the starting point here (imagine privately owned devices wandering around a shopping mall).

Definition 1 (Autonomy) *No agent can force any other agent to accept or transmit information, alter its state, or otherwise change its behavior.*

It is harder because one must then put back all of that complexity, by hand, to show how such individual agents can form collaborative structures, free of conflict.

2.3 Promises

A promise is a general and abstract unit of intent. Promises, between agents, can deal with things like quality of service, quality of behavior, specifications of state, etc. Policies of various types have been identified. For instance, in the Ponder model[10], one has authorizations (promises to grant access) and obligations (promises to follow up on a different promise) or dependency, etc. These can all be translated into the notion of promises.

Consider, then, a set of autonomous agents or objects represented as nodes $\mathcal{N} = \{n_1, n_2, \dots, n_N\}$ in a graph.

Definition 2 (Promise) *A promise is a labeled directed edge (link) that connects two nodes. The promise label represents a specifically intended range of behavior χ from within a domain of possible behaviors. i.e. $n_1 \xrightarrow{\chi} n_2$. A promise is thus made by a node n_1 to a node n_2 . A promise is assumed to be always kept.*

Notice that, in the definition, the agent-nodes, between which promises are made, are kept separate from the constraints between them. This is important for avoiding the kinds of ordering ambiguities alluded to in the example above.

Example 9 (Service Level Agreement (SLA)) *Agent n_1 promises agent n_2 to provide service of type ‘database access in time q ’, τ is the type domain $q \in [0, \infty]$ and the constraint $\chi(q) : 0 < q < 10\text{ms}$.*

The formulation of a promise, above, has obvious characteristics of a directed graph.

Two special types of promise will be identified below, in order to build/rebuild conventional structures from these basic atoms.

- A promise to agree to behave like another.
- A promise to utilize the promise of another.

The first of these is essential for defining groups, roles and social structures with consensus behavior. The latter is crucial for client-server interactions, dependencies and access control. The rest of this paper is about logically combining individual promises into collective and consistent policies that allow cooperation between autonomous agents.

2.4 What is an inconsistency?

In this elementary scheme, the only inconsistency that can occur is if an agent promised two contradictory things to a second agent.

Definition 3 (Broken promise) *A promise of χ_1 and type τ from agent n_1 to agent n_2 is said to be broken if there exists another promise from n_1 to n_2 , of χ_2 and type τ , in which $\chi_1 \neq \chi_2$.*

It says that an agent can only break its own promises: if an agent promises two different things, it has broken both of its promises. Two promises of different types do not contradict one another, but two promises about the same thing are clearly inconsistent. Put another way, the intersection of any χ_i, χ_j of the same type must be the null set for all i and j .

2.5 Uniformity through collaboration

What is required is a construction based on freely given promises that allows independent agents to come together and form structures spanning several nodes; i.e. by *voluntary cooperation*. Such an agreement has to be made between every pair of nodes involved in the cooperative structure. We summarize this with the following:

Definition 4 (Cooperative promise rule) *For two agents to guarantee the same promise, one requires a special type of promise: the promise to cooperate with neighboring agent-nodes, about basic promise themes.*

A complete structure looks like this:

- n_1 promises p to n_3 .
- n_2 promises n_1 to collaborate about p (denote this as a promise $C(p)$).
- n_1 promises n_2 to collaborate about p (denote this as a promise $C(p)$).
- n_2 promises p to n_3

By measuring p from both n_1 and n_2 , n_3 acts as a judge of their compliance with the mutual agreements between them (see fig. 2.1). This allows the basis of a theory of measurement, by third party monitors, in collaborative networks. It also shows how to properly define structures in the file-directory example (see fig 2.1).

If only one of the agents promises to do as the other, then they will both do the same, but one agent assumes a position of control over the other. The agent which has not made a promise can lead the other by the nose, making it follow its every whim. Only if both agents promise bilaterally to cooperate, can they obtain a promise that is a stable fixed point.

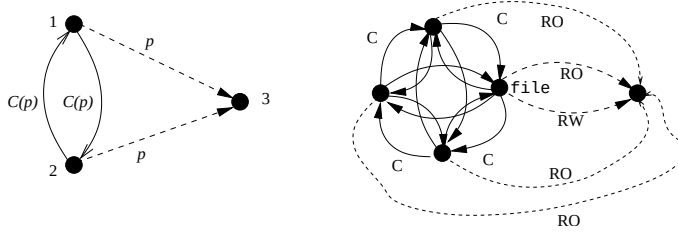


Figure 2.1: (Left) Cooperation and the use of third parties to measure the equivalence of agent-nodes in a region. Agents form groups and roles by agreeing to cooperate about policy. (Right) This is how the overlapping file-in-directory rule problem appears in terms of promises to an external agent. An explicit broken promise is asserted by file, in spite of agreements to form a cooperative structure (i.e., in spite of an implicit agreement among the nodes at the left to act as a group).

2.5.1 Dependencies and handshakes

Even networks of autonomous agents have to collaborate and delegate tasks, depending on one another to fulfill promised services. We must find a way of expressing dependency relationships without violating the primary assumption of autonomy.

Consider three agents n_1, n_2, n_3 , a database server, a web server and a client. We imagine that the client obtains a web service from the web server, which, in turn, gets its data from a database. Define propositions and validities:

- p_1 = “will send database data in less than 5 ms,” $n_1 \rightarrow n_2$.
- p_2 = “will send web data in less than 10 ms,” $n_2 \rightarrow n_3$.

These two promises might, at first, appear to define a collaboration between the two servers to provide a promise of service to the client, but they do not.

The promise to serve data from $n_1 \rightarrow n_2$ is in no way connected to the promise to deliver data from $n_2 \rightarrow n_3$:

- n_2 has no obligation to use the data promised by n_1 .
- n_2 promises its web service regardless of what n_1 promises.
- Neither n_1 nor n_3 can force n_2 to act as a conduit for database and client. (Indeed, in the case of n_3 , this is not even a reasonable expectation as any interaction between n_2 and n_1 ought to be transparent to it.)

What is needed is a structure that complete the backwards promises to *utilize* promised services – promises that completes a *handshake* between the autonomous agents. We require:

- A promise to uphold p_1 from $n_1 \rightarrow n_2$.
- An acceptance promise, to use the promised data from $n_2 \rightarrow n_1$. In many cases, this is analogous/equivalent to an acknowledgment.

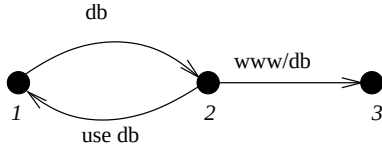


Figure 2.2: Turning a conditional dependency into a real promise. The necessary structure is shown in graphical form.

- A conditional promise from $n_2 \rightarrow n_3$ to uphold p_2 iff p_1 is both present and accepted.

Thus, three components are required to make a dependent promise (see fig. 2.2). This requirement cannot be derived logically; rather, we must specify it as part of the semantics of autonomy.

Assumption 1 (Acceptance/usage promise rule) *Autonomy requires an agent to explicitly accept a promise that has been made, when it will be used to derive a dependent promise.*

One thus identifies a second special type of promise: the “usage” or “acceptance” promise.

2.6 Autonomous, voluntary cooperation

What use is this construction? First, it advances the manifesto of making all policy decisions explicit. In the example in fig. 2.2, it shows explicitly the roles and responsibilities of each of the agents in the diagram. Furthermore, the graphical representation of these promises is quite intuitive and easy to understand. The construction has two implications:

1. The component atoms (promises) are all visible, so the inconsistencies of a larger policy can be determined by the presence or absence of a specific link in the labeled graph of promises, according to the rules.
2. One can provide basic recipes (handshakes etc.) for building consensus and agent “societies,” without hiding assumptions. This is important in pervasive computing, where agents truly are politically autonomous and every promise must be explicit.

The one issue that we have not discussed is the question of how cooperative agreements are arrived at. This is a question that has been discussed in the context of cooperative game theory[11, 12], and will be elaborated on in a future paper[13]. Once again, it has to do with the human aspect of collaboration. The reader can exercise imagination in introducing fictitious, intermediate agents to deal with issues such as shared memory and resources.

2.7 Interlopers: transference of responsibility

One of the difficult problems of policy consistency is in transferring responsibilities from one agent to another: when an agent acts as a conduit or interloper for another. Consider agents a , b and c , and suppose that b has a resource B which it can promise to others. How might b express to a : “You may have access to B , but do not pass it on to c ”?

The difficulty in this promise is that the promise itself refers to a third party, and this mixes link-worlds with constraints. As a single promise, this desire is not implementable in the proposed scheme:

- It refers to B , which a has no access to, or prior knowledge of.
- If refers to a potential promise from a to c , which is unspecified.

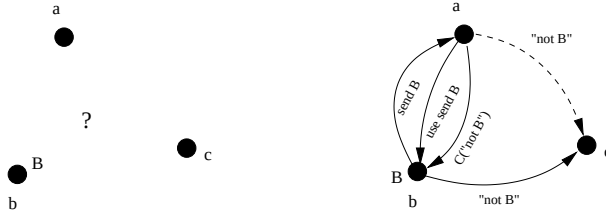


Figure 2.3: Transference of responsibility.

- It preempts a promise from a to b to never give B along $a \rightarrow c$.

There is a straightforward resolution that maintains the autonomy of the nodes, the principle of separation between nodes and constraints, and which makes the roles of the three parties explicit. We note that node b cannot order node a to do anything. Rather, the agents must set up an agreement about their wishes. This also reveals that fact that the original promise is vague and inconsistent, in the first instance, since b never promises that it will not give B to c itself. The solution requires a cooperative agreement (see fig. 2.3).

- First we must give a access to B by setting up the handshake promises: i) from $b \rightarrow a$, “send B ,” ii) from $a \rightarrow b$, accept/use “send B .”
- Then b must make a consistent promise not to send B from $b \rightarrow c$, by promising “not B ” along this link.
- Finally, a promises b to cooperate with b ’s promises about “not B ,” by promising to cooperate with “not B ” along $a \rightarrow b$. This implies the dotted line in the figure that it will obey an equivalent promise “not B ” from $a \rightarrow c$, which could also be made explicit.

At first glance, this might seem like a lot of work for express a simple sentence. The benefit of the construction, however, is that it preserves the basic principles of make every promise explicit, and separating agents-nodes from their intentions. This will be crucial to avoiding the contradictions and ambiguities of other schemes.

2.7.1 Promises with strings attached

Constrained promises are a seemingly similar situation to transferred responsibility when the constraint refers to a third party. For example, consider the previous scenario again. Agent a desires B from b , but b is willing to provide it only if a agrees not to pass it on to c . In this case, suppose that b is the authoritative source for B , making c equivalent to the rest of the world.

In this case, a must promise b not to pass on the information to c , but b makes no such promise itself (quite the contrary, in fact).

This case is structurally simpler than the previous one in that a and b make a pair of promises to one another, and sending/using B is contingent on a ’s not disseminating B any further (i.e., to any c):

- First, b promises that only it can give access to B , and a must promise to cooperate with b .
- Then we can give a access to B by setting up the handshake promises: i) from $b \rightarrow a$, “send B ,” ii) from $a \rightarrow b$, accept/use “send B .”

This is expressed with the “&&” notation in fig. 2.4. Note that no promises involve c directly.

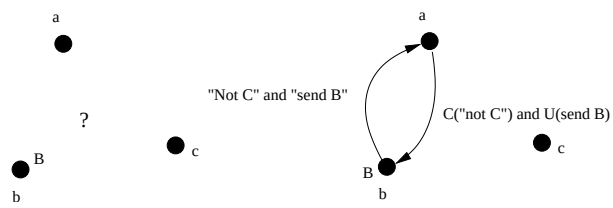


Figure 2.4: A constrained promise.

Chapter 3

Applying promise theory to cfengine

Promise theory is about autonomous agents. How does this idea apply to cfengine? It has always been a principle of cfengine that hosts should be autonomous in a management sense, but promise theory is stronger than that.

Promise theory is essentially a model for generalized *services*. The challenge, then, is to see cfengine as a service carried out by autonomous agents.

3.1 Agents and constraints

Cfengine deals with objects such as files, processes, disks, interfaces, etc. Each of these will be represented by an autonomous agent in the sense of promise theory. This might seem counter-intuitive, since files cannot really play an active role in anything, but this we can show that this is just a formality, and the modeling is appropriate.

Because cfagent's function is to *alter* the configuration attributes of an object, when they are incorrect, we need to think of two distinct phases or mappings that are involved in each cfengine statement.

There is a set of possible configurations for an object of a given type. We can denote the type as τ , e.g. file attributes, web process configuration.

By making a policy, we promise to restrict the behavior of our system to a subset of that possible realm ρ_τ , which we denote $\mathcal{P}$ (for constraint). Finally, if the system falls outside of the allowed constraint, we must identify a unique state $q \in \mathcal{P}$ to set it to, by an operation $\hat{O}_q$. The choice of q is arbitrary, as long as it is within the policy set $\mathcal{P}$.

Consider the diagram in fig. 3.1.

- **Typed Promises Π :**

A promise is a description of what subdomain of values $\mathcal{P}$ a state of type τ may occupy, according to policy. The set ρ_τ represents a domain of possible values for the state. The subdomain $\mathcal{P} \subset \rho_\tau$ could be represented as a regular expression, or a list, etc.

A promise is thus a surjection from ρ_τ to $\mathcal{P}$.

- **Typed Operations $\hat{O}$:**

For each type of promise there is an operation that brings an arbitrary state of this type back into a promised range. An operation is a many-to-one function that is convergent.

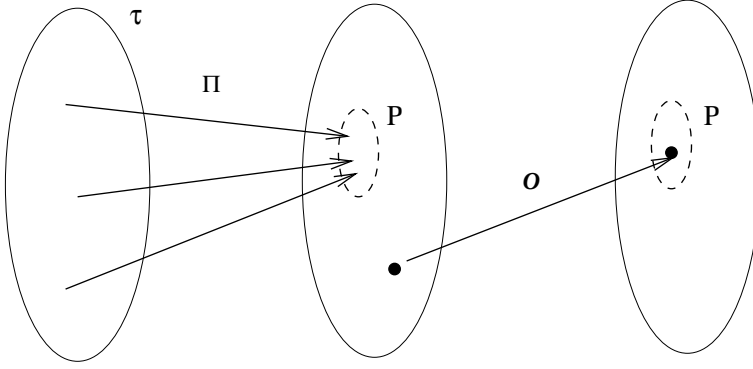


Figure 3.1: The mappings in policy from the space of possibilities to the policy-allowed subset, to a representative element of the policy allowed set: $\tau \rightarrow \mathcal{P} \rightarrow q \in \mathcal{P}$

An operation $\hat{O}$ is applied to any state $q \in (\rho_\tau - \mathcal{P})$, i.e. any state outside of policy. It takes all such states to a single representative value q , which is called the *ideal state*. An operation is an idempotent surjection from $\rho_\tau - \mathcal{P}$ to $\mathcal{P}$.

3.2 Convergent operations imply promises

A special feature of cfengine is that it expressed policy implicitly through the asserted operations. It does not traditionally separate promise from repair-operation.

Because of the similarities between the promise and the maintenance operations, cfengine tries to simplify the situation by hiding the distinction between them.

3.3 A service model of operations

The individual agents, in the sense of promise theory, are processes, files, and the resources of the system; in addition, cf agent (as a process) is itself an agent.

We might want to imagine that policy means that cfagent promises to make everything on the system “right,” but this is the wrong way to define a promise for autonomous agents, because the resources control their own attributes. Cfagent should not promise something about the state of attributes that do not belong to it (even though its privileged position in Unix often allows it).

Instead, we must turn things around and imagine cfagent to be the observer of everyone else’s promises. Policy is not something that is centrally defined, it is the sum of many individual promises. (The fact that most agents share the same cfagent.conf file is not yet relevant.)

Definition 5 (cfagent as Promise Observer) *Cfagent functions as a observer of promises. In this way, the reciprocal nature of promises (promise pairs) are ensured. In addition, this monitoring allows resources to (conceptually) make promises within the cfengine framework.*

3.3.1 Between whom are promises made?

For the purpose of verification, cfagent has the natural status of an observer. We define promises from every configuration object to cfagent, for instance:

$$\text{/etc/passwd} \xrightarrow{\pi} \text{cfagent} \quad (3.1)$$

where π stands for $\{ \text{file mode} = 644, \text{owner} = \text{root} \}$.

But, we would also like to have the cfagent promise to respond if the resource's promise is inadequate. Consider an autonomous agent that has the task of configuring a resource somewhere else. What are the promises required to sustain and maintain policy between these independent parts of the system?

Let us consider this situation in two stages of increasing sophistication. We shall think of the behavior of the prototype configuration agent cfagent which requires that a host configure a resource within a constraint such that a measured value q should lie in a set of acceptable policy values $\mathcal{P}$.

We shall pick a particular τ and stay with it so that we may simplify the notation by eliminating ρ_τ . The resource promises the agent that it will obey $\mathcal{P}$ and also that it will obey the agent's wishes; the agent promises to set the value of q to a policy conformant value q_p (see fig. 3.2).

$$\text{resource} \xrightarrow{q \in \mathcal{P}} \text{agent} \quad (3.2)$$

and cfagent promises to set the value of q to a specific value in $\mathcal{P}$. These promises are

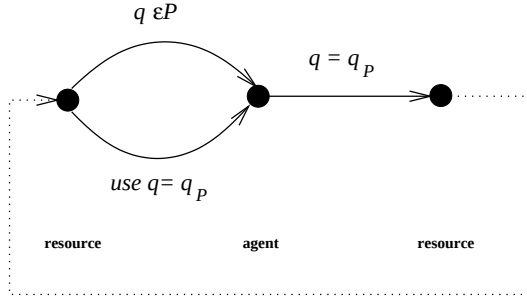


Figure 3.2: The simplest agreement between a configuration agent and a resource.

consistent but they do not capture the usual convergent semantics of the promises made by configuration agents (e.g. cfengine [9]). Note that the role of the ‘use’ promise is to authorize the agent to act: it obliges the resource to comply with the agent's promise. This is the glue that makes a constraint into an action (operation).

In the second version (see fig. 3.3), we add a conditional promise that gives the promises convergent semantics: if we denote the use promise as an operator $\hat{O}_{\text{use}}$, we may write:

$$\begin{aligned} \hat{O}_{\text{use}} q &= q_p, \quad q_p \in \mathcal{P} \\ \hat{O}_{\text{use}} q_p &= q_p, \quad \forall q_p \in \mathcal{P}. \end{aligned} \quad (3.3)$$

We denote the complement of $\mathcal{P}$ by $\neg\mathcal{P}$. In other words, the agent promises:

$$\text{agent} \xrightarrow{q=q_p/q \in \neg\mathcal{P}} \text{resource} \quad (3.4)$$

The agent associates an implicit action item to this promise, i.e. it will set a fixed value from the policy set if the actual value is found to lie outside the policy set. For example, if the resource R is a file and the policy set is the set of strings $\{A, B.C\}$, the agent A might choose to set the contents of the file, i.e. the value $q_p = B$.

In the third and final version of this policy agreement, we change the perspective and place the agent in the driving seat, making it the source and instigator of policy rather than merely a police force for a pre-decided rule-set.

$$\left(A \xrightarrow{q=q_p/q \in \neg\mathcal{P}} R \otimes R \xrightarrow{\text{use}} A \right) \otimes R \xrightarrow{q \in \mathcal{P}/(q=q_p/q \in \neg\mathcal{P})} A \quad (3.5)$$

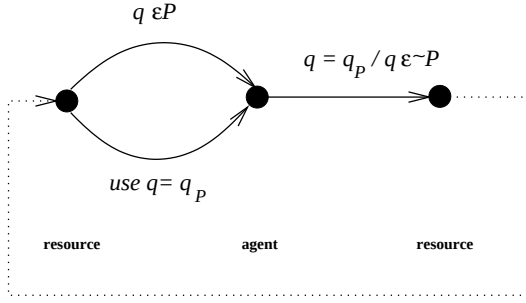


Figure 3.3: An agreement between a configuration agent and a resource with a conditional promise.

This is a circular promise, and we may show that it is self-consistent by applying the composition rule:

$$\pi' = \begin{cases} q \in \mathcal{P} & \text{if } (q = q_p / q \in \neg \mathcal{P}) \subseteq (q = q_p / q \in \neg \mathcal{P}) \text{ (which is true)} \\ \emptyset & \text{if } ((q = q_p / q \in \neg \mathcal{P}) \cap (q = q_p / q \in \neg \mathcal{P}) = \emptyset) \text{ (which is false)} \end{cases} \quad (3.6)$$

In other words, setting a value $q \rightarrow q_p$ is consistent with q and q_p being elements of the policy set $\mathcal{P}$:

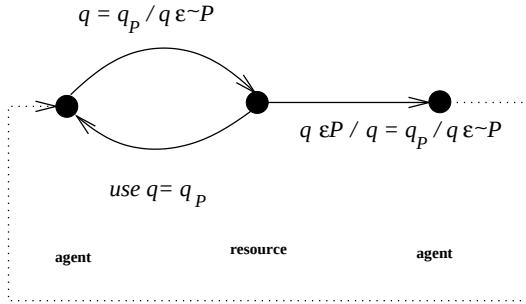


Figure 3.4: A comprehensive agreement between a configuration agent and a resource with a conditional promise. Note the reversal of node-roles.

Note the interesting result that it is unnecessary for the resource to promise that it will have only a single ‘correct’ value, i.e. $q = q_p$, it is sufficient to allow a more general fuzzy range $q \in \mathcal{P}$. No inconsistency or indeterminacy of action arises.

3.3.2 What is an operator?

An operator is easily identified as a bilateral service agreement between two parties. The agent promises to inform the resource that about the correct value, and the resource promises to use that value.

$$\begin{aligned} \hat{O}_{q_p} &\equiv \text{resource} \xrightarrow{\text{use } q_p} \text{agent} \otimes \text{agent} \xrightarrow{\text{please set } q_p} \text{resource} \\ &\equiv \left(\text{resource} \xrightarrow{\text{use } q_p} \text{agent} \right) \xleftarrow{q_p} \end{aligned} \quad (3.7)$$

3.3.3 How can the resource promise something?

If the resource is a file, how can it promise to adopt a state proposed to it by cfagent? This is a formality that we can easily deal with by splitting cfagent into a passive part and an

active part. We can associate the active part of the agent with every resource, to allow the resource to effectively change its value. This is allowed as long as the active and passive parts of cfagent do not communicate with one another, except by the explicit promise arrows we draw.

The point of having this construction is to make all promises (information transfers) explicit, thus as long as this is the case, the particular model of abstraction is not important. The usefulness of this construction is in converting an implicit logical problem into an explicit graphical one, with simple rules.

3.3.4 Distributed promises

Notice that, the passive cfagent plays the role of a central monitor. There is nothing in promise theory that distinguishes the location of the agents. They might be co-located on the same host (as with cfengine), or on different hosts (as with SNMP). The only assumption is that promises will be securely transmitted.

3.3.5 Cfengine file syntax

The syntax of cfengine has the form

$$\begin{aligned} \text{object} \quad \text{promise} &= \pi_1 \\ \text{promise} &= \pi_2 \dots \\ \text{promise} &= \pi_n \end{aligned} \tag{3.8}$$

which is simply a pragmatic rewrite of

$$\text{object} \xrightarrow{\pi_1, \pi_2, \dots, \pi_n} \text{cfagent} \tag{3.9}$$

3.3.6 Conditional promises and cfengine class triggers

If one promise is made or used, it can trigger a conditional promise (see fig. 3.5). In cfengine quasi-syntax:

```
<object-0> promise=constrain-0 define=yes
yes::
    <object-1> promise=constrain-1
!yes::
    <object-2> promise=constrain-2
```

We might at first expect this scenario to look like the dependency chain in fig. 3.5:

However, this picture is not correct, since all promises are made between objects and cfagent. Instead, we have the picture in fig. 3.6.

Once again, this structure indicates that the monitoring cfagent needs to communicate back information to the scheduling cfagent. This occurs through classes, which themselves are formally a service provided by the monitor to the operational scheduler.

Note that we suppress the bilateral promises from each of the objects, as in fig. 3.2, for clarity.

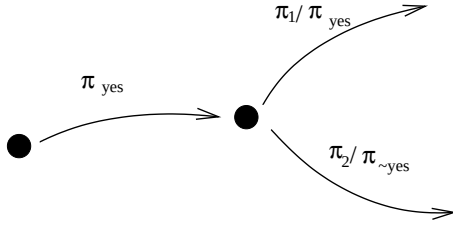


Figure 3.5: We might expect the conditional structure above to look like this diagram, but we must remember that all configuration promises are between objects and cfagent.

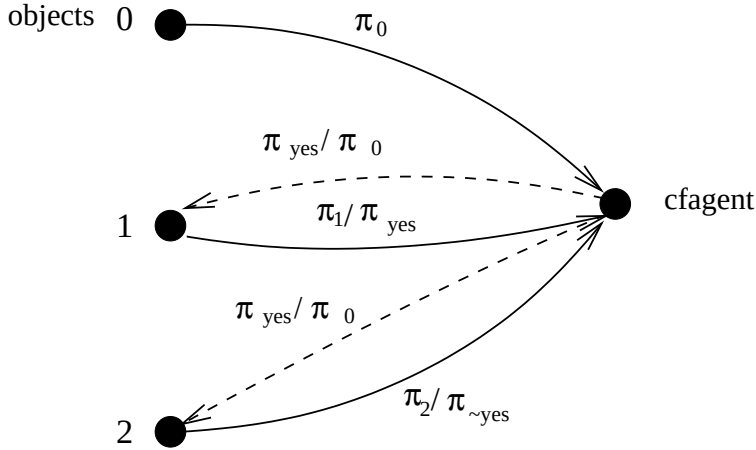


Figure 3.6: The initial object makes a promise. The agent then reports back to each object if the promise is kept a values of “yes” with a broadcast promise, to each object, of the value of a variable “yes.” The two lower objects then make their own conditional promises based on whether the promise is “yes” or “!yes” (not yes).

3.3.7 Cooperative promises

Cooperative promises arise when several objects promise the *same* behavior to a third party (in this case the cfagent monitor).

By grouping them together in a syntax of the form,

$$\begin{aligned}
 \text{compound-object promise} &= \pi_1 \\
 \text{promise} &= \pi_2... \\
 \text{promise} &= \pi_n
 \end{aligned} \tag{3.10}$$

one can imagine implicit contracts between the objects in the compound object. The compound object can be represented by a regular expression or simply as a directory or process container, as in fig. 2.1.

3.4 Cfengine constructions as promises

3.4.1 Most promises

Most cfengine actions are of the simple form of a single promise about state from an object (file, interface, process etc) to the cfagent node.

3.4.2 Classes

Classes are determined by observation. The cfagent monitor determines information about objects that are not represented in the graph (e.g. hard classes like operating system type). This information must be considered internal to the cfagent node. Any node in the graph can, in principle make observations about the environment, but in cfengine all such information is passed via the cfagent node.

Class information is passed to objects as a promise or class-service from the agent, and they promise to use that information by being defined in an appropriate class, as in fig. 3.5 and section 3.3.6:

```
classes:

    observed_class = ( Observation )

action:

    observed_class::

        <promise>
```

3.4.3 Copy files

A copy operation is a straightforward service provision in which the service promises the contents of the source file and the result is transferred as a conditional promise to cfagent.

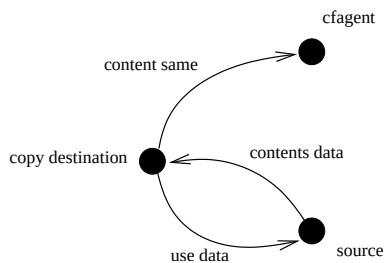


Figure 3.7: A copy action, represented as promises.

3.4.4 Link files

A link operation is a straightforward service provision in which the service promises the name or identity of the source file and the result is transferred as a conditional promise to cfagent.

3.4.5 Methods

Methods are cfengine's subroutines or function calls. They can take parameter substitutions, classes and can return both variables and classes to the calling method or function.

Modeling this behavior presents a number of subtleties in the context of promise theory, but we learn a lot by doing so.

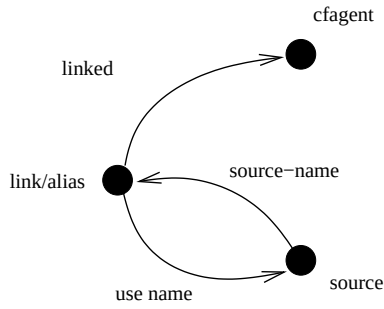


Figure 3.8: A link action represented as promises.

Arguments represent the values of data that can be substituted in parameterized rules. An argument or parameter could be

- The name of an object.
- The scope of the promise's constraint.

The only channel of communication is in fact the promises, so arguments must be communicated as a set of promises to relay these values by the local cfagent.

Each method call is bound by a use-promise to a caller that supplies the argument service, and relays the data to the object agents within the cluster. Since each object can read argument information authorized to it from the local cfagent, data can be passed to the correct agent by a promise.

Agents make promises, so they are the repository of the constraint information. (We write this information on the arrow to indicate which it applies to, but the information originates from the pointing object agent. Thus there is no difficulty about communicating variable data that parameterizes constraints.)

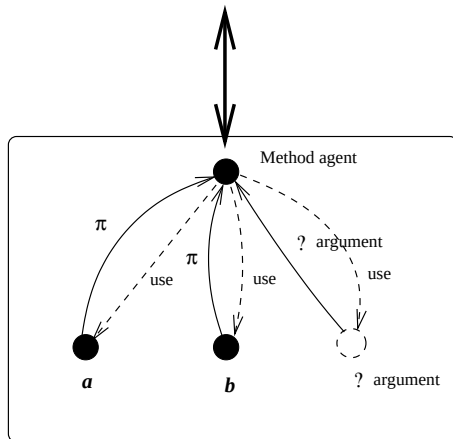


Figure 3.9: A method is a self-contained cluster of promises to a “local agent.” Each method must have its own representative cfagent, to whom promises are made. This agent is also needed to relay communications between the methods, its objects and the calling method.

Return values are passed back as conditional promises from the local cfagent node to the calling agent. On receipt of a confirmation of action promise, the local cfagent promises to return a value or a boolean class to the caller.

Chapter 4

A consistent syntax

In this chapter, we begin to consider how cfengine 3 syntax will embody promises and convergent operations.

4.1 Lessons: What does promise theory tell us about cfengine?

The preceding discussion has led to the following key conclusions with respect to applying promise theory to the design and functioning of cfengine:

1. Cfengine provides/makes possible a framework of intercommunicating independent agents working together by voluntary cooperation. These are responsible for managing the resources.
2. It must include capabilities which are able to monitor (cfenvd), transmit data (cfservd or local), and act to alter state (cfagent).
3. Parameters can be passed within this framework as conditional promises.

Since all promises are mediated through cfagent, cfagent is the recipient of all policy promises, except when promises are used `define` or `elsedefine` to trigger conditional promises. A basic promise therefore consists of a resource object and a number of constraints.

1. There is a resource object with a privileged significance.
2. There are different types (or categories) of promise. Enumerating the distinct, orthogonal types must be one of the objectives in designing cfengine 3.
3. Resource objects should not overlap—e.g. `/etc/*` and `/etc/passwd`—at least with respect to the same promise. Ideally, no overlapping of objects would be permitted. However, enforcing this would make operations like `tidy` unmanageable since `tidy` must be able to traverse and examine paths which are cited as objects elsewhere.
4. Boolean results can be used as conditionals (cfengine classes)
5. There must be a constraint (promise) about the object's attributes and/or characteristics, for each type of promise about the object. However, not all constraints need be embodied in a `Constraint` stanza (see below). Constraints can be defined formally as such, but they are also implicit in the options used in every object specification.

6. Methods are not necessarily closures, in the sense of object orientation, for the resource objects. They are simply named clusters of promises, possibly parametrized as patterns. However, if there are no broken promises, every promise is a closure with respect to its data.
7. As each method has its own local agent with its own state and *beliefs* about the environment, belonging to a specific method has the potential for prejudicing the state with respect to which the parametrized promises are made. Care is required to avoid this pitfall.

4.2 A data model

4.2.1 Object types

The following object types are proposed:

- File system objects: i.e., files and directories. The corresponding action item is `files-and-dirs`.
- Processes: processes whose existence should be verified, that should be signaled, and so on. The corresponding action item is `processes`.
- Command: external command to be run. The corresponding action item is `commands`.
- Devices: Devices to be validated, configured, etc. Currently, the defined devices would be network printers and network interfaces. The corresponding action item is `devices`.
- Database records: specific entries within LDAP directories, the Windows registry and potentially other directories and databases. The corresponding action item is `records`.
- File systems: local disk partitions and remote shares which can be mounted, unmounted, shared or whose status should be validated. The corresponding action item is `filesystems`.
- Misc. system configuration: configuring `/etc/resolv.conf` and setting the time zone. The corresponding action item is `sysconfig`.
- Packages: files to be installed on or removed from the system. The corresponding action item is `packages`.
- Methods: user defined cfengine “functions” which can be parametrized and reused. The corresponding action item is `methods`.

Note that all other current action sequence items are removed. Their functionality remains, but appears in a more consistent and (hopefully) convergent way in the syntax.

4.2.2 Constraints

Making templates for the many command attributes in cfengine 2 has been suggested as a way of rationalizing syntax. An alternative is to separate the definition of constraints entirely from the place where they are used. This approach is already used in “filters” and “acls,” which are merely constraints already. It seems natural to generalize this idea. An `editfile` block on a file is also a constraint on a file attribute (its contents).

Cfengine 3 constraints—or promises—are a syntactical mechanism for grouping a set of attributes into a named, reusable unit. In keeping with the new theoretical underpinnings, constraints can also be referred to as promises (and both keywords are supported).

Multiple constraints can be applied to a given object. However, it is very important to understand at the outset that any constraints must be designed/defined to be orthogonal. In addition, since some items that can appear within constraints can also appear as settings attached to the item itself, the same considerations apply. (In fact, the item specification itself is also a constraint.) Cfengine 3 will always flag multiple specifications of the same setting as an error regardless of the source(s) of the duplication except in the case of overrides of global defaults (set in the `defaults` constraint).

Constraints must also be constructed and applied so that only items which are applicable to the latter are included. If there are options within the constraint which are invalid in the usage context, an error will result.

Constraints may be grouped using the `group` constraint type. However, any “circular” nesting is an error.

Constraint types

The following types of constraints can be defined:

- **perms**: sets of permissions specifications, including traditional UNIX file ownership and modes, ACLs, and other related items.
- **whatfiles**: File system object inclusion, exclusion and/or ignore patterns (regular expressions).
- **copy**: Settings related to file copying, including the source locations for files, comparison criteria when deciding whether to update an existing file, and the like.
- **execution**: Settings related to command execution.
- **transaction**: Settings related to the application frequency and outcome of cfengine specifications (e.g., `ifelapsed`, `expireafter`, `define`, `inform`, `log`, and so on).
- **editfile**: File editing specifications.
- **rotate**: File rotation specifications and conditions.
- **checksum**: Specifies file checksum comparison parameters.
- **spool**: Specifies parameters related to tidying up spool directories.
- **depth**: Settings related to recursion and link-following.
- **filter**: Current cfengine 2 filters.
- **defaults**: Sets default values for various parameters. Can be overridden with the `constraints=nodefault` option in any item.
- **group**: A named set of constraints which can be applied as a unit. *Note that overuse of this feature will result in unreadable configuration files and is thus strongly discouraged.*

See the examples in the next chapter for more details.

Open question: what would constraint parameters look like?

4.2.3 Assignments

Currently cfengine mixes the setting of control attributes with the definition of user-variables. There is no clean understanding or distinction between lists, scalars and arrays. These should be separated.

4.2.4 Object rules

It is proposed, first of all, to terminate all statements with semi-colons in order to define syntax more clearly. This also makes it easier to parse.

A policy statement needs

- A promise (constraint).
- A maintenance response (convergent operation).
- Transaction attributes (administration).

For example, something like this:

Example 10

```
object_type:

  classes::

    <resource object> constraints=template1, template2, ...
                        setting = value, ... ;
```

A resource object is then a set of one or more objects (as defined above), specified as a single item, a list, a regular expression, and so on.

4.3 Handling functions

What should the handling functions (really callbacks, or “functors” or function objects) look like? How should they behave?

We would like to:

- Make the language extensible
- Discourage the use of arbitrary, non-convergent functions.

My feeling is that we have to be very careful about this. The goal is not to create as much freedom as possible, but to balance flexibility with constraint.

Methods will have the following general structure:

```
method name(param1, param2, ...)
{
  control:
    method-specific control directives
    [premethodcommand = command]
    [postmethodcommand = command]
    actionsequence = list

  parameters:
    type param1 [default=value] [valid=range-or-test]
```

```

    type param2 [default=value] [valid=range-or-test]
    sendclasses = list-of-classes

returns:
    list-of-variables
    returnclasses= list-of-classes

variables:
    local-variable-declarations

action-sequence-sections
...
}

```

Methods are invoked in the `methods` stanza (see the examples).

Methods may call other methods (including themselves?).

Local variables and positional parameters are dereferenced with a dollar sign as usual: `$param1`. Parameter default values allow arguments to be omitted when the method is invoked. Without a default value, a missing parameter will result in an error.

Note that the outermost “level” of every `cfengine 3` configuration is inherently a method. This is indicated using the reserved method name `cfengine` (which takes no arguments). This results in the following general configuration file structure:

```

method cfengine
{
control:

probes:

action items ...
}
constraint perms sysfiles
{
...
}
additional constraints and methods ...

```

4.4 Type-inference

4.5 Iteration and arrays

One usually wants to iterate over objects rather than attributes, i.e. many objects can have common attributes, but a single object cannot have many different sets of attributes. Thus, implicit iteration over a promise-set is easily accounted for.

The semantics of iteration must be defined in each context. For instance, iteration over a remote server could mean fail over: try until success. Iteration over a file object would mean do all these things.

Iteration over the following:

```

/$(list1)/$(list2)

```

must be also defined. The most obvious meaning would be the ordered combinations of the items in the two lists (please consider all control structures in the this subsection as pseudo-code and not as proposed syntax):

```
foreach i in list1
{
  foreach j in list 2
  {
    /$(i)/$(j)
  }
}
```

In an array context, an intuitive interpretation is to use consecutive elements of two arrays, thereby stepping through the them sequentially:

```
# requires sizeof(array2) >= sizeof(array1)
foreach i = 1 .. sizeof(array1)
{
  /$(array1[i])/$(array2[i])           # or i -> j
}
```

In this second case, consistency is obviously contingent on the cardinality of the arrays.

Finally, for associative arrays, iteration logically should consist of using elements with matching keys:

```
foreach k in keys( $(list1[]) )
{
  /$(list1[k])/$(list2[k]) if $(list2[k]) != NULL
}
```

This latter formulation should exclude elements from either array which are not present in both arrays.

4.5.1 Recursion

In languages like ML, one has “tail end” recursion, in which a recursive function can read off a stack in order to process a list. Is this a desirable property of cfengine rules, once parameters are allowed?

4.6 Compliance with standards

Common criteria

Service management

Configuration management

4.6.1 Anomaly detection additions

- alerts here
- disk free space
- CPU usage
- Free memory
- Reachable network connection

Chapter 5

Specification through examples

This section provides examples of configuration file entries for all of the cfengine 3 capabilities. Be aware that we do not concern ourselves with classes in this discussion in order to focus on the constraints and operations themselves. In addition, we assume familiarity with the functioning and features of cfengine 2, so the explanations of individual examples are somewhat terse.

5.1 Constraints

Constraints have the following general structure:

```
Constraint type name
{
    option1 = value1 ;
    option2=  value2 ;
    ...
}
```

In the interest of regularizing the syntax, equals signs have been used here are elsewhere. However, colons are also acceptable (as in cfengine 2 filter specifications).

5.1.1 perms: Permissions, ownerships and other attributes

This is the most complex constraint type. It includes the ACL definitions found in cfengine 2 (several distinct types), as well as standard UNIX ownerships and file modes, BSD flags, and permissions for other entities such as database records and exported file systems.

Multiple permission types can be mixed within a single **perms** constraint. However, errors will result from both internal inconsistencies (duplication?) between the items (e.g., both an ACL specification and a UNIX file mode) or applying a constraint containing items which do not apply to that specific context (e.g., including BSD flags and having it be invoked on a Linux system).

What follows are sample constraints each containing a single item which serve to illustrate the latter.

The following is an example constraint defining UNIX file ownerships and permissions:

```
Constraint perms sysbinaries
{
    permtype = unix ;
    owner = root ;
    group = 0 ;
```

```
mode = 755 ;
}
```

A unix perm constraint must contain at least one of owner, group, and mode.

The following constraint defines a POSIX ACL:

Constraint perms chem

```
{
  permtype = posixacl ;
  method = append ;                # or overwrite

  user::rwx ;
  group::rw ;
  other::noaccess ;
  mask::rwx ;

  user:mark:rx,w ;
  user:aeleen:remove ;            # currently the keyword is default
  group:chem:rw ;
}
```

If you wanted to set the ownership of the file(s) as well, then you would create a constraint with both a unix and a posixacl item, but the unix item would need to omit the mode.

Here is an example, which also illustrates constraint grouping:

Constraint perms rootbin

```
{
  permtype=unix ;
  owner=root ;
  group=bin ;
}
```

Constraint group chemperms

```
{
  constraints=rootbin,chemacl ;
}
```

The following constraint defines a Windows NT/2000/XP/2003 ACL:

Constraint perms winchem

```
{
  permtype = win2kac1 ;
  method = append ;
  group:Everyone:remove ;

  user:mark:full:allow ;          # one can use either predefined permission
  user:aeleen:modify:allow ;      # set names or atomic permission codes
  group:chem:rxwdp:allow ;        # update to post-Win2K: rxwdabtuispmo
}
```

The following constraint defines an ACL for Windows share permissions:

Constraint perms xpshares

```
{
```

```

permtype = winshareacl ;

group:Everyone:remove ;
group:Authenticated Users:full:allow ; # full or change or read
group:Chem:read:allow ;
}

```

An analogous form can be used for permissions on Windows registry entries (`permtype = winregacl`), with the defined permissions letters modified to support those that are valid in that context.

The following constraint defines additional file attributes provided by BSD-based operating systems:

```

Constraint perms nodelete
{
  type = bsdflags ;
  flags = uchg, uunlnk ;
}

```

5.1.2 whatfiles: Inclusion, exclusion and ignore lists

The following constraint illustrates this constraint type, which specifies inclusion, exclusion, and/or ignore lists for various contexts:

```

Constraint whatfiles fpfiles
{
  include = "\.dbx$" ;           # regular expressions
  include = "\.prg$" ;
  include = "\.idx$" ;
  exclude = "\.*junk" ;
  exclude = nodefault ;         # don't use the global exclude list
  ignoredirs = /tmp ;
}

```

5.1.3 execution: Command execution

Here is an example constraint for a shell command or process:

```

Constraint execution normal
{
  useshell = false ;
  chdir = / ;
  chroot = none ;
  background = true ;
  umask = 022 ;
  uid = 0 ;
  gid = 0 ;
}

```

The `uid` and `gid` options refer to the process execution context. Ownership and permissions settings for files that may be created are specified via a separate `unixperms` constraint.

5.1.4 transaction: Action control and outcome

This constraint type includes items which specify frequency, outcome classes and logging of actions cfengine may take. Here is a simple example which sets `inform` to true, specifies logging to an external file and to syslog and specifies that detected errors/inconsistencies are to be repaired:

Constraint transaction fixverbose

```
{
  inform = true ;
  log = file, syslog ;
  action = repair ;
}
```

This example constraint specifies classes to be defined and removed when an action is and is not taken (the fact that all four possibilities are present here should *not* be construed as good practice; it is merely illustrative in this instance):

Constraint transaction defines

```
{
  define = hupinetd ;
  elsedefine = killinetd ;
  undefine = killsyslog ;           # remove class(es) on success ...
  elseundefine = hupsyslog ;       # ... or failure
}
```

`ifelapsed` and `expireafter` may also appear in `transaction` constraints.

5.1.5 editfile: Modifying file contents

This constraint type is the most problematic and the most difficult to use in a convergent manner. Indeed, some/most existing `editfiles` editing directives are (inherently) non-convergent, or at the very least are quite difficult to use in a reliably convergent manner (unless they follow `EmptyEntireFilePlease`).

On the other hand, the following directives are almost always convergent (at least in the absence of bugs in cfengine itself). Another way of saying this is that it is very challenging to create closure using directives editing other than those in the following list. The cautious/prudent system administrator might consider limiting `editfile` constraints to these:

- `AppendIfNoSuchLine`, `AppendIfNoLineMatching`, `AppendIfNoSuchLineFromFile` and their `Prepend` variants.
- `AppendToLineIfNotContains`
- `CommentLinesContaining`, `CommentLinesMatching`, and their `Hash`, `Percent`, `Slash` and `Un` prefixed variants. Note that the `Hash`, `Percent`, and `Slash` versions are removed from cfengine 3; use the `SetCommentStart` and `SetCommentEnd` directives to set the desired commenting method.
- `DeleteLinesContaining`, `DeleteLinesMatching`, `DeleteLinesStarting`, `DeleteLinesContainingFileItems`, and their `Not` variants
- `EmptyEntireFilePlease`: note that virtually any directive(s) following this one will be convergent.
- `FixEndofLine`

- All of the Warn directives.
- ReplaceAll
- SetCommentStart, SetCommentEnd, SplitOn, UMask, AutoCreate, and Begin ... End blocks.

Here are two example `editfile` constraints. The first one comments out lines beginning with “rs” with a hash (#) character:

```
Constraint editfile hashrs
{
  SetCommentStart = "#" ;
  SetCommentEnd = "" ;
  CommentLinesStarting "rs" ;
}
```

This constraint uncomments any lines containing the specified path string, and then adds such a line to a file if it is not already present somewhere within it:

```
Constraint editfile addmotd
{
  UncommentLinesMatching    ".*usr/local/bin/motd.pl.*" ;
  AppendIfNoLineMatching    " */usr/local/bin/motd.pl[ $]" "/usr/local/bin/motd."
}
```

The `editfilesize` and `editbinaryfilesize` settings can also be specified in an `editfile` constraint.

5.1.6 rotate: File rotation settings

The constraint type specifies settings for rotation of log files and other files. Here is an example, which rotates files which are more than 500 MB in size or are 30 days or more old, compresses the old copies using, gzip, keeping up to 10 old copies, and using numerical suffixes to name the saved copies:

```
Constraint rotate varlog
{
  rotate = 10 ;
  compress = gzip ;           # others: compress, bzip2
  suffix = number ;          # or timestamp
  age = 30 ;
  size => 500MB ;
  test = AND ;               # could be OR
}
```

These features would make cfengine 3 essentially equivalent to facilities like logrotate.

5.1.7 checksum: The checksum database

This constraint specifies the location of the checksum database as well as the validate vs. update mode for cfagent:

```
Constraint checksum system
{
  checksumdatabase = /var/cfengine/database ;
}
```

```

checksumpurge = off ;
checksumupdate = off ;
checksumtypes = "md5, sha" ;      # can store >1 checksum like Tripwire
}

```

5.1.8 spool: Cleaning spool directories

The `spool` constraint is used to specify parameters for cleaning up spool directories (see `files-and-dirs` below for an example use):

```

Constraint spool print1
{
    deletenonuserfiles = on ;
    deletenonownerfiles = off ;
    warnnonuserfiles = off ;
    warnnonownerfiles = on ;
}

```

5.1.9 depth: Recursion and link traversing

These constraints specify recursion depths, whether device boundaries should be crossed and whether links should be followed. Here is an example:

```

Constraint depth alltheway
{
    recurse = on ;
    xdev = off ;
    links = follow ;
}

```

5.1.10 filter: Specify complex file filtering criteria

The current filtering mechanisms are, in fact, constraints, and should be incorporated into `cfengine 3`.

The following simple example selects regular files with the `setuid` bit set:

```

Constraint filter setuid
{
    mode = +2000 ;
    type = reg ;
    result = mode.type ;
}

```

5.1.11 copy: Settings related to file copying/updating

This constraint type specifies options related to file copying for `files-and-dirs: op=copy`. Here is an example which specifies file comparisons based on modification times, no file purging, and a list of three potential locations for the source files:

```

Constraint copy updates
{
    type = mtime ;
    sources = norway:/depot, sweden:/new, denmark: ;
    srctype = tree ;
}

```

```

    purge = false ;
}

```

5.1.12 defaults: Default values for various parameters

This constraint type sets default values for many parameters. Previously, these were set in the `control` section. Note that this constraint is automatically applied in every context, regardless of whether any other constraints are used (i.e., `constraints=defaults` is always assumed—see below). However, it may be overridden via the `nodefaults` keyword in the constraints list for an entry.

Constraint defaults

```

{
    binarypaddingcharacter = "\0" ;
    defaultcopytype = mtime ;
    defaultpkgmgr = rpm ;
    dpkginstallcommand = "/usr/bin/pkgmgr %s" ;
    excludecopy = regexp(s) ;
    excludelink = regexp(s) ;
    expireafter = 1 ;
    ifelapsed = 120 ;
    inform = off ;
    fileextensions = list ;
    linkcopies = list ;
    repchar = "_" ;
    repository = path ;
    rpmcommand = command ;
    rpminstallcommand = command ;
    sensiblecount = 2 ;
    sensiblesize = 1000 bytes;
    split = ":" ;
    suninstallcommand = command ;
    suspiciousnames = list ;
}

```

Note that this constraint type does *not* accept a name.

5.2 Action sequence stanzas

5.2.1 files-and-dirs

This item specifies configuration and maintenance for file system objects: files and directories. It encompasses the functionality of the `copy`, `disable`, `editfiles`, `files`, `links`, `directories`, and `tidy` action sequence items of cfengine 2.

Here is the general syntax:

files-and-dirs:

```

class::

    fs-object
        general settings ...
        op = copy | disable | rotate | editfile | tidy | validate
        operation specific settings ... ;

```

The `op` setting specifies what sort of operation is used to configure or maintain that object or set of objects. Only one operation may be specified. In addition, the set of file system objects specified within the various stanzas in this section ought to be exclusive: an file or directory should ideally be present only once (after class processing), either by being listed explicitly or via wildcards (the one possible exception to this may be with `op=tidy` where directories that are mentioned elsewhere might again be traversed). If you need to perform multiple operations on a single file or directory, then you probably want to encapsulate the actions within a method (for an example, see the section on methods later in this document),

Examples for each operation type follow, beginning with the simplest.

tidy

Here is an example use of the `tidy` operation, which deletes files that are 5 days old and older from a list of directories:

```
constraint transaction
{
    name = verbose
    syslog = on
    inform = on
}
...

files-and-dirs:
    /$(dirs)      op=tidy constraints=verbose,nodefault
                  recurse=inf age=5 type=mtime xdev=on
                  links=keep rmdirs=false ;
```

This example removes links which point nowhere from user home directories:

```
files-and-dirs:
    home          op=tidy recurse=inf links=tidy xdev=on ;
```

The link tidying options present in many places in cfengine 2 are thus all gathered here in the new scheme.

The following example tidys the system spool directories:

```
files-and-dirs:
    /$(spooldirs) op=tidy recurse=inf constraint=myspool ;
```

The presence of a `spool` constraint indicates that this invokes the special spool directory cleaning facility. In the interest of readability, we might want to add an `action=cleanspool` or similar option.

Other options for this operation are the normal inclusion/exclusion options (`include`, `exclude`, `ignore` and `filter`), along with the remaining `transaction` constraint options not used here (the four `define` options plus `ifelapsed` and `expireafter`). Note that the latter group are valid for every operation type and so will be mentioned only briefly from here on.

disable

Here are some examples of the `disable` operation:

```

files-and-dirs:
    /etc/hosts.equiv    op=disable ;                # => hosts.equiv.cfengine
    home/.rhosts        op=disable dest=.BadBoy ;
    /etc/gshadow        op=disable repository=/usr/local/depot ;
    /etc/inetd.conf     op=disable suffix=old ;      # => inetd.conf.old
    /lib/testlib.s  o   op=rename dest=reallib.so ;

```

As the final example illustrates, `rename` is a synonym for `disable`.

This operation also accepts the transaction options as well as `type` and `action`.

rotate

The file rotation functionality is separated from the disabling/rename operation in cfengine

3. Here is an example:

```

constraint rotate
{
    # rotate every 30 days and when size > 500 MB
    name = logfiles
    rotate = 5                # keep 5 files
    size => 500MB
    suffix = numbers
    age = 30
    compress = bzip2
    test = or
}

```

```

files-and-dirs:
    /var/log/*              op=rotate constraint=logfiles type=file ;
    /usr/local/log/sqllog   op=rotate rotate=2 suffix=timestamp age=7 ;

```

The first example shows the use of a constraint to specify the rotation parameters. The second uses only explicit options.

The transaction-related options are also valid with the `rotate` operation, as is `repository`.

validate

Validation is the bread-and-butter operation for files and directories. This operation tells cfengine to bring such objects into line with the desired specifications. It is one of the key components of convergent system maintenance.

Here are some examples:

```

promise unixperms
{
    name = sysperms
    owner = root
    group = bin
    mode = 755
}

```

...

```

files-and-dirs:

    /$(sysdirs)            op=validate constraints=sysperms recurse=inf ;

```

```

/$(admdirs)      op=validate owner=root group=0 mode=550 recurse=inf ;

home            op=validate constraints=suspectfiles,nodefault
                mode=u-s,g-s,go-w  # turn off set{u,g}id, non-owner w
                recurse=inf xdev=off links=traverse ;

```

These examples descend into directory trees and verify the ownership and permissions on the files, correcting them if necessary. The first two examples are quite similar, with one using a promise/constraint and the other using explicit options. The third example uses the `suspectfiles` constraint (not shown) to employ inclusion and exclusion rules, and also deactivates any default inclusion, exclusion and/or ignore lists that have been defined.

These examples verify the presence of important files and directories, creating them if necessary:

`files-and-dirs:`

```

/var/log/messages op=validate constraints=sysperms action=create ;
/scratch          op=validate recurse=off mode=777 action=create ;

```

The current version of cfengine also provides an `action=touch` option. This option is almost impossible to use in a convergent way for existing file (i.e., when it is not equivalent to `action=create`). We should consider removing this “touch of evil.”

This operation can also be used to verify the presence of symbolic links, creating them if necessary, as in these examples:

`files-and-dirs:`

```

/bin -> /usr/bin      op=validate constraints=sysperms ;
/sw/bin +>! /usr/bin  op=validate type=absolute recurse=off ; # Mac fink
/g03 -> /mf/src/g03   op=validate nofile=force # ok to point nowhere

```

The transaction-related and file inclusion/exclusion/ignore options are valid with this operation.

Note that the file copying capabilities present in the cfengine 2 “link” stanzas are not present here. They are incorporated into `op=copy`.

copy

The functions of the cfengine 2 “copy” action item become part of `files-and-dirs op=copy`. The “copy first” option in cfengine 2 becomes a standard part of this facility whenever more than one server appears in the `servers` list.

Consider these examples of copies from local and remote systems:

`# note that the local target file is now the object!`

`files-and-dirs:`

```

/etc/hosts.allow    op=copy type=mtime srctype=file
                    source=/depot/hosts.allow ;

/etc/hosts          op=copy type=mtime
                    source=norway:/depot srctype=tree ;

```

```

/etc/issue                op=copy source=norway:,denmark:,sweden: ;

/usr/local/rpms/*.rpm      op=copy type=checksum recurse=off
                           source=norway:/depot/rpms,
                           (sweden,denmark):/pub/rpms,
                           france: (/usr/share/rpms,/opt/rpms),
                           (a,b): (/c,/d) ;

/usr/local/patches        op=copy source=norway:/etc/depot/patches/*.rpm
                           recurse=off purge=true ;

```

The first example is a simple local file copy from `/depot/hosts.allow` to the file of the same name in `/etc` if the modification time of the former is later.

The next example is a similar remote copy. It attempts to copy a file named `hosts` in `/depot/etc` on server `norway`. If there is such a file, and its modification time is later than that of the local file, the copy will be made.

The `src` option is needed to disambiguate where on the server `cfagent` is to look; it tells how to interpret any path specified in the source list. The first example uses the option `file`, which means that the source will give the full path to the desired file; in other words, it forces the source path to be interpreted as a file path and not as a directory. The other options are `directory`, which says to look for a file of the same name in the specified directory, and `tree`, which says to look for a file with the same path *relative* to the specified directory. The default is `directory` when no wildcards are present in the source specification and `file` when wildcards are present.

If no path is specified with the server name, then the same path as the target object is assumed.

It is possible to have lists of sources: servers and/or directories. The third example will try to update the `/etc/issue` file if necessary. It will search the `/etc/` directory on servers `norway`, `denmark` and `sweden` in turn, stopping at the first found instance of the file. The test as to whether to copy the file (here, using the default criteria of creation time) is performed for the first match *only*.

The next (fourth) example illustrates several points. A wildcard is used in the object specification, which means that every file in the specified directory with the extension `.rpm` is a candidate for updating. There is a substantial list of remote directories to check. The parentheses allow lists to be applied on either side of the colon. Thus, `"(a,b):/c"` expands to `"a:/c, b:/c"`; parentheses on both sides result in all ordered combinations: `"(a,b):(/c,/d)"` means `"a:/c, a:/d, b:/c, b:/d."` Lists iterate from left to right.

When there are multiple target object and multiple source locations, then the search is conducted for each item individually. In other words, each item is looked for starting from the start of the (fully expanded) source list, regardless of where the previous item was located (`cfengine` optimizes searching internally). The first located version of each individual file is used.

In the course of such searches, nonexistent directories are simply ignored.

The final example illustrates using a wildcard in the source list with a directory as the object. In this case, all of the matching files in the remote directory are candidates for copying. What is being updated here is the specified local directory. It is being synchronized with the remote directory. In this case, all `.rpm` files on the remote server which are not present in the local directory or a newer than the version in the local directory are copied. In addition, any files in the local directory which are not present on the remote server will be deleted (`purge=true`), regardless of their extension (see the next example).

Legally, you can have wildcards in both locations:

`files-and-dirs:`

```
/pub/*.txt          op=copy purge=true  
                    source=norway:/text/*.txt,sweden:/data/*.txt ;
```

This will update all of the .txt files in /pub from either norway or sweden, as appropriate. It will also add any files present in either location which are not in the local directory. Finally it will remove any .txt files from /pub which were not found on either server. Using the .txt extension for both remote locations effectively limits the scope of the purge operation to those files.

Note that there is a potential pitfall here. If there are files on both norway and sweden (but not on the local system already), and those on sweden are newer, they will probably not get updated. To do so would be to add to the original file list dynamically, which is not a good idea (IMHO). Such use is therefore *not* recommended.

This operation also accepts the file inclusion, file permissions, filter, transaction, and depth options, as well as the current `stealth`, `repository`, `timestamps`, `typecheck`, `force`, `forcedirs`, `forceipv4`, `size`, `trustkey`, `encrypt`, `verify` and `findertype` options. The functionality of the link-related options is available in `op=validate`.

The `include`, `exclude` and `ignore` options also now take the keyword `noglobal` in addition to patterns; it says to deactivate any global list defined in the `defaults` constraint.

editfile

This operation performs file editing. Note that all of the “real work” happens in the constraint. Here is a simple example (which uses a constraint we defined earlier):

files-and-dirs:

```
home/.login          op=editfile constraint=addmotd ;
```

The recursion, file inclusion, filters and transaction options can also be used with `editfile`.

But I Want to Do More Than One Thing to a File!

Write a method (see below).

5.2.2 processes

Process handling is little changed from cfengine 2. The `transaction`, `execution`, `filter` and `whatfiles` constraints may be used (although in the latter case, it is process names rather than file system objects, which are matched), as can the existing `restart`, `useshell`, `signal`, `matches` and `action` options.

5.2.3 shellcommands

Like `processes`, this action item is essentially unchanged. The `transaction`, `execution`, `filter` and `whatfiles` constraints may be used (although in the latter case, it is process names rather than file system objects, which are matched), as can the existing `timeout`, `preview`, `noabspath`, `useshell`, `action` and `background` options. A `perms` constraint can also be used to specify the permissions for any files created as a result of the command execution.

5.2.4 devices

This section is used to configure and validate devices. The defined devices at the moment are network interfaces and LPD-based network printers.

Here is are some examples:

devices:

```
eth0          type=netiface hostname=gudinne ipv4=192.168.0.100
              netmask=255.255.255.0 broadcast=ones
              defaultroute=192.168.0.200 ;

ibsen         type=printer protocol=lpd                  # could also support SMB?
              address=nexus                             # IP address or hostname
              netmask=255.255.255.0                     # if different from local
#             name of the queue on the specified server--omit to use default
              queue=lp ;
```

The transaction options also apply here.

5.2.5 records

This section checks for the presence and settings of database and directory entries. Here are some example entries from this section:

records:

```
"uid=aeleen,ou=users,dc=ahania,dc=com" dbtype=ldap server=ldapmaster
                                         attribute=gidNumber value=589
                                         action=warn ;

"uid=*,cn=users,dc=ahania,dc=com"       dbtype=ldap action=fix
                                         binddn=$(ldapadmin) password=xxx
                                         server=ldapmaster port=636 # SSL
                                         attribute=loginShell
                                         value=/bin/tcsh ;

"$HKEY_CLASSES_ROOT/.tex"               dbtype=winreg server=seattle
                                         attribute="PerceivedType"
                                         datatype=REG_SZ value=text
                                         action=create ;
```

The first example checks the value of the gidNumber attribute in the LDAP record with the specified dn, warning if it is different.

The second example sets the value of the loginShell attribute to /bin/tcsh for all users in a Mac OS X domain if it is not already this value. The wildcard in the dn string causes all matching dn's to be considered. This example also illustrates additional options for locating and binding to the LDAP server.

The third example checks for the specified record in the Windows registry on the specified server, creating it if necessary, and ensuring that the setting is as specified (in Windows parlance, the key is \$HKEY_CLASSES_ROOT, the value is .tex and the data is text). Note that action=create implies also action=fix.

SQL databases offer more challenges since they are relational rather than hierarchical. Presumably, cfengine 3 would have to use some ODBC-based facility for interacting with them.

Specifying the object(s) of interest is challenging. These thoughts should be considered preliminary. Here, we look at the contents of a single cell, the value of the column “state” in the row where “id” is 123456 in table “addresses” in database “employees”:

records:

```
"state from employees.dbo.addresses where id=123456"
    odbc=myconnection value="CT" action=validate ;
```

You could also use `action=update` to modify the value. Note that `from` and `where` are keywords separating the field, database and where clause.

Open questions about SQL databases:

- Is any of this really a good idea?
- Should multiple fields be allowed, with a value list of the same length to compare with?
- How complex a where clause should be allowed? What about joins? What about field lists containing fields from different tables?
- Do we want to just have the user put in the whole SQL query? If so, what can be done with the results?

5.2.6 filesystems

This action is designed to check for the presence of important local and remote file systems, as in these examples:

```
filesystems:
    /home                device=/dev/hdb1 type=ext3 ;

    /usr/src/g03         device=norway:/mf/src/g03 type=nfs ;

    /office              device=/seattle/office type=smb
                        options="user=aeleen,password=xxx" ;    # for mount -o
```

Unwanted file systems can also be unmounted:

```
filesystems:
    /junk                action=umount force=true ;    # default action=mount
```

File systems can also be exported/shared using this action item:

```
filesystems:
    /home2               action=export type=nfs
                        writelist=$(nfsallow) ;

    /openoffice          action=export type=smb sharename=oo
                        readlist="+users" writelist="+chem,+phys" ;
```

These entries validate the presence of the specified file system among the export/share lists for the local system. Signaling/restarting servers must be handled separately.

The transaction options are also valid here, as is `scanarrivals`.

5.2.7 sysconfig

This section is used to configure `/etc/resolv.conf` and the time zone:

```
sysconfig:
# this action type uses keywords as objects
    resolve      domain=mydomain.net nameserver=$(serverlist)
                  current=keep ;

    timezone      zone=EST5EDT ;
```

The `resolve` object can take the `search` option instead of `domain`. Whichever one is chosen determines which keyword appears in the first line of the file (i.e., “domain” or “search”). The other value that the `current` option can take is `purge`, which removes any nameserver entries currently in the `/etc/resolv.conf` file.

The transaction-related options also apply here.

5.2.8 packages

This is pretty much the same as the current one, with one additional option:

```
packages:
    openssl      pkgmgr=rpm cmp=ge version=0.7 pkgdir=path ;
```

The `pkgdir` option specifies a list of directories in which to search for package files when the desired package level is not installed.

The transaction-related options also apply here.

5.2.9 probes: Probe system state and set/unset classes

The `probes` section is used to define and launch probes: items which determine current system state. Regardless of this section’s placement, probes are always done before all actions. Thus, “probes” is not a valid `actionsequence` keyword.

Probes are structurally similar to `filter` constraints in structure. However, they are used to probe the current system state prior to `cfagent` taking any action. They incorporate similar tests and functions as filters, but they define and/or undefine classes depending on the results.

This probe checks whether the SSL package (RPM) is installed:

```
Probe has_ssl
{
    ExecResult = "/usr/bin/rpm -q openssl (not installed)" ;
    test = ExecResult ;
    define = needssl ;          # also, elsedefine
    undefine = havessl ;       # also, elseundefine
}
```

5.2.10 methods

This section is used to invoke methods. See the separate section below.

5.2.11 Moved and deleted action sequence items

The following table lists action sequence items which have been removed along with the item providing the same functionality.

Cfengine 2 item	Cfengine 3 item
addmounts	filesystems
binserver	files-and-dirs
disks	files-and-dirs
mountall	filesystems
mountinfo	filesystems
unmount	filesystems
required	filesystems
broadcast	devices
defaultroute	devices
netconfig	devices
interfaces	devices
editfiles	files-and-dirs: op=editfile
copy	files-and-dirs: op=copy
disable	files-and-dirs: op=disable — rotate
files	files-and-dirs: op=validate
directories	files-and-dirs: op=validate
links	files-and-dirs: op=validate
tidy	files-and-dirs: op=tidy
checktimezone	sysconfig
resolve	sysconfig
ignore	whatfiles constraint
mailcheck	removed
homeservers	removed

5.3 The control section

Based on the previous discussions, it is clear that the `control` section has been greatly reduced. It now functions more in accordance with what its name literally implies: to specify the overall functioning of cfagent.

The following items remain in the `control` section:

- `access`: allowed users
- `actionsequence`: action list
- `allowredefinitionof`: variables which can change value
- `bindtointerface`: outgoing network interface
- `childlibpath`: library path environment variable value for child processes
- `domain`: local domain
- `dryrun`: just say what we would have done
- `exclamation`: control shrieking
- `homepattern`: where home directories are
- `interfacename`: nonstandard network interface name
- `lastseen`: remember/forget when hosts last connected

- `moduledirectory`: path to methods files
- `schedule`: set schedule for running cfagent
- `secureinput`: do/don't import files not owned by process' owner
- `showactions`: Add info to output prefixes
- `site` (synonym `faculty`): Name ourselves
- `skipidentify`: do/don't assume DNS registration
- `smtpserver`: server for outgoing mail
- `splaytime`: interval for load sharing
- `sysadm`: system administrator's email address
- `syslog` and `syslogfacility`: use/don't use syslog and specify syslog facility (respectively)
- `timezone`: time zone in use on local system
- `timeout`: connection timeout period
- `verbose`: output verbosity
- `warnings`: whether to output warnings

The following items are new:

- Nothing so far ...

5.3.1 Moved and deleted control section items

Cfengine 2 item	Cfengine 3 item
addclasses	removed (unnecessary)
addinstallable	removed (unnecessary)
autodefine	removed
binarypaddingcharacter	defaults constraint
checksumdatabase	checksum constraint
checksumpurge	checksum constraint
checksumupdates	checksum constraint
copylinks	removed
defaultcopytype	defaults constraint
defaultpkgmgr	defaults constraint
deletenonuserfiles	spool constraint
deletenonownerfiles	removed
deletenonusermail	removed
deletenonownermail	removed
dpkginstallcommand	defaults constraint
editbinaryfilesize	editfile constraint
editfilesize	editfile constraint
emptyresolveconf	removed
excludecopy	defaults constraint
excludelink	defaults constraint
expireafter	defaults constraint
hostnamekeys	removed
ifelapsed	defaults constraint
inform	defaults constraint
lastseenexpireafter	alerts
fileextensions	defaults constraint
linkcopies	defaults constraint
logdirectory	removed
logtidyhomefiles	files-and-dirs
mountpattern	removed
netmask	netdevices
nonalphanumfiles	files-and-dirs
nfstype	removed
repchar	defaults constraint
repository	defaults constraint
rpmcommand	defaults constraint
rpminstallcommand	defaults constraint
sensiblecount	defaults constraint
sensiblesize	defaults constraint
singlecopy	files-and-dirs
split	defaults constraint
spooldirectories	spool constraint
suninstallcommand	defaults constraint
suspiciousnames	defaults constraint
warnnonuserfiles	spool constraint
warnnonownerfiles	spool constraint
warnnonusermail	removed
warnnonownermail	removed

5.4 Methods

Here again is the basic structure of a method, this time more heavily annotated:

```
method name(param1, param2, ...)
{
# Can contain the usual control directives, as well as commands
# to be run at the start and end of the method.
control:
    method-specific control directives ;
    [ before = command ; ]
    [ after = command; ]
    [ actionsequence = list ; ]

# Positional parameters; default values make them optional
parameters:
    type param1 [default=value] [valid=test] ;
    type param2 [default=value] [valid=test] ;
# Sendclasses are defined classes which are carried into the local
# class space of the method (others are invisible)
# Built-in classes are always present.
    sendclasses = list-of-classes l

# Returns: both variables, returned as method.name (as in cfengine 2),
# and a list of classes to be set (both are optional)
returns:
    list-of-variables ;
    returnclasses = list-of-classes ;

# Declarations of local variables
variables:
    local-variable-declarations ;

# The heart of the method -- the code ...
action-sequence-sections
...
}
```

Here are some simple initial examples of methods. This first example disables a list of network services controlled by xinetd which is specified as its first parameter by moving the files from /etc/xinetd.d to /usr/local/sbin/xinetd.unused. The suffix “.save” or a user-supplied suffix will also be applied to the moved files. The method stops the xinetd daemon before moving the files and restarts it afterwards.

```
method fix_xinetd(services, suffix)

control:
    before = "/etc/init.d/xinetd stop" ;
    after = "/etc/init.d/xinetd start" ;

parameters:
    list services ;
    string suffix default=".save" ;

files-and-dirs:
```

```

/etc/xinetd.d/${services}    op=disable    suffix=${suffix}
                             repository=/usr/local/sbin/xinetd.unused ;

```

The following method illustrates a way of performing two actions on a file. The method first copies a list of files to a specific directory, then it rotates the same list of files. Note that ordering is handled by the `define` option to the copy operation.

```
method myrotate(indir,outdir,files,days,number)
```

```
parameters:
```

```

    list indir ;
    directory outdir ;
    list files ;
    int days default=7 valid=1:20 ;
    int number default=5 ;

```

```
files-and-dirs:
```

```

    /$(outdir)                op=copy source=$(indirs)/$(files)
                              owner=root group=root mode=550
                              define=do_rotate ;

```

```
do_rotate::
```

```

    $(indir)/$(files)    op=validate action=rotate compress=gzip
                        age=$(days) rotate=$(number) suffix=number ;

```

This example method installs a package for a service if it is not already present and then starts the new service:

```
method newsvc(name,rpmdir,rundir,script)
```

```
{
parameters:
```

```

    string name ;
    dir rpmdir ;
    dir rundir default="/etc/init.d" ;
    string script default=$(name) ;

```

```
packages:
```

```

    $(name)    pkgmgr=rpm pkgdir=$(rpmdir) elsedefine=nopkg ;

```

```
services:
```

```
!nopkg::
```

```

    $(name)    op=start initdir=$(rundir) script=$(script) ;

```

```
}
```

The `elsedefine` option to the `packages` entry ensures that the method will not attempt to start the service if the package installation fails.

Here is a simple method for verifying and managing processes associated with defined system services, using the operating system supplied hooks for doing so (e.g., the control scripts in `/etc/init.d` on traditional Linux/Unix systems). Its workings are based around classes which are activated based on its parameters.

```
method svc_ctrl(pname,type,p2,p3,action,testop)
```

```
{
control:
```

```

        actionsequence = ( processes shellcommands ) ;

parameters:
# process name or regexp to match
  regexp pname ;

# parameter sets class indicating service type
  class type default=svc_unix valid="svc_unix|svc_win|svc_cust" ;

# the next 2 parameters have type-dependent definitions
# type      p2      p3
# unix      script name  script dir
# win       service name  unused
# custom    start cmd    stop cmd
  string p2 default=$(pname) ;
  string p3 default=/etc/init.d ;

# parameter sets class indicating what to do
  class action default=restart valid="verify|start|stop|restart" ;

# process-section option for service verification
  string testop default="matches=>1" ;

returns:
  returnclasses = svc_ok ;

processes:

  verify:: $(pname)  action=warn $(testop)
            define=$(svc_ok) elsedefine=need_restart ;

  svc_cust&restart::
            $(pname)  signal= kill define=need_start ;

shellcommands:

  svc_unix&(need_restart|!verify)::

            "$(p3)/$(p2) $(action)" define=$(svc_ok) ;

  svc_win&(need_restart|!verify)::

            "net stop $(p2) & net start $(p2)" define=$(svc_ok) ;

  svc_cust&(need_start|!verify)::

            "$p2" define=$(svc_ok) ;

  svc_cust&stop::
            "$p3" define=$(svc_ok) ;
}

```

5.4.1 Invoking methods

Methods are invoked in the `methods` stanza, as in these examples:

```
methods:
    fix_xinetd($(badservices),) ;

    myrotate(/var/log,/usr/local/rawlogs,$(logfiles),30,10)
        pre="/etc/init.d/syslog stop"
        post="/etc/init.d/syslog start" ;

    newsvcl("bacula",,"/usr/local/sbin/bacula",) ;
```

The invocation of the `myrotate` method runs specified commands before and after invoking the method, which serve to stop and restart the `syslog` daemon, actions which are necessary in this invocation for a series of `syslog`-based log files, but are not always so (and thus are not included in the method).

Here are some sample uses of the `svc_ctrl` method:

```
methods:
    svc_ctrl(smb,svc_unix,,,verify,,) ;

    svc_ctrl(cron,svc_unix,,,start,,) ;

    svc_ctrl(watchover,svc_unix,wover,/usr/local/etc/init.d,restart,) ;

    svc_ctrl(holster,svc_cust,"/usr/local/sbin/upholster -n3",,
        verify,"number=<3") ;

    svc_ctrl("Event Log",svc_win,,,verify,) ;
```

The first example determines whether the `smb` daemon is running, starting it if it is not. The second example similarly starts the `cron` daemon.

The third example restarts a service called “`watchover`,” using the `wover` script in the specified directory, passing it the keyword “`restart`” as is typical for UNIX initialization scripts.

The fourth example causes the `holster` service to be restarted, using the specified command.

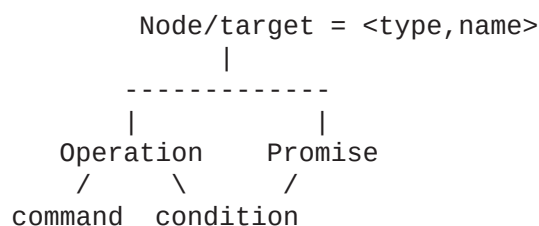
The final example verifies that the `Event Log` service is running on a Windows system, again starting it if it is not.

In all cases, the method defines the class `SVC_OK` if its operations are successful.

5.5 Miscellaneous

A promise usually does not contain an operation, but here we are describing the condition in terms of an operator since they share the same information.

Cfengine does not promise anything that it cannot (at least attempt to) repair by itself.



```

/      of state
attribs      /      \

```

```

    ifelapsed state\_converged

```

Graph operations: ;target name, type, promise_edge;
 ;ifelapsed,expireafter,condition; = bounded in time and state, i.e. if belongs logically
 with the operator and not with the target
 An operator has a ;TYPE,target,template;
 Goals: efficiency, correctness, simplicity
 A Cooperative promise C() is like a shared template.
 A Use promise is a copy or access service U() in response to an access control list

Special:

```

    Undo last

```

Classifiers:

```

    ProbeService::
    ReadData::
    IsFile etc::

```

Search parameters:

```

    filter
    matches=?

```

Target-types:

```

    files:
    processes:
    commands:
    mounts:
    link: (?)

```

```

    interface/service-port:

```

Promises/Operators: { group syntax? }

```

    source (copy/link)
    access (owner,group,mode)|acl
    signal
    execute
    IP-config
    qos

```

To evaluate a rule, you'll need to expand each target rule, assembling all the pieces in a list first, and then expanding variables, before constructing the evaluation context.

Bibliography

- [1] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.
- [2] M. Burgess. Computer immunology. *Proceedings of the Twelfth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.
- [3] M. Burgess. Evaluation of cfengine’s immunity model of system maintenance. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, 2000.
- [4] M. Burgess. Configurable immunity for evolving human-computer systems. *Science of Computer Programming*, 51:197, 2004.
- [5] A. Couch, J. Hart, E.G. Idhaw, and D. Kallas. Seeking closure in an open world: A behavioural agent approach to configuration management. *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII) (USENIX Association: Berkeley, CA)*, page 129, 2003.
- [6] A. Couch and Y. Sun. On observed reproducibility in network configuration management. *Science of Computer Programming*, 53:215–253, 2004.
- [7] A. Couch and S. Schwartzberg. Experience in implementing an http service closure. *Proceedings of the Eighteenth Systems Administration Conference (LISA XVIII) (USENIX Association: Berkeley, CA)*, page 213, 2004.
- [8] S. Traugott. Why order matters: Turing equivalence in automated systems administration. *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI) (USENIX Association: Berkeley, CA)*, page 99, 2002.
- [9] M. Burgess. Cfengine’s immunity model of evolving configuration management. *Science of Computer Programming*, 51:197, 2004.
- [10] N. Damianou, N. Dulay, E.C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.
- [11] S. Fagernes and M. Burgess. The effects of ‘tit for tat’ policy for rejecting ‘spam’ or denial of service floods. In *Proceedings of the 4th System Administration and Network Engineering Conference (SANE 2004)*, 2004.
- [12] R. Axelrod. *The Complexity of Cooperation: Agent-based Models of Competition and Collaboration*. Princeton Studies in Complexity, Princeton, 1997.
- [13] M. Burgess and S. Fagernes. Pervasive computing management ii: Voluntary cooperation. *IEEE eTransactions on Network and Service Management*, page (submitted).