# Configurable immunity for evolving human-computer systems

Mark Burgess
Oslo University College,
Cort Adelers Gate 30, N-0254 Oslo, Norway

November 29, 2003

**Abstract**

The immunity model, as used in the GNU cfengine project, is a distributed framework for performing policy conformant system administration, used on hundreds of thousands of Unix-like and Windows systems. This paper describes the idealized approach to policy-guided maintenance, that is approximated by cfengine, building on the notion of 'convergent' operations, i.e. those that reach stable equilibrium. Agents gravitate towards a policy determined configurations, through the repeated application of unintelligent 'anti-body' operations or discrete, coded countermeasures. The distributed agents turn passive discovery of state into active strategy for 'curing' systems of policy transgressions. Keywords: autonomous computer management, cfengine, immunity model.

## 1 Introduction

A central problem in system administration is the construction of a secure and scalable scheme for maintaining configuration integrity of a computer system over the short term, while allowing configuration to evolve gradually over the longer term. The importance of policy-based configuration management, to this task, has been expounded since the early 1990's[1, 2, 3, 4]. Policy is an important tool for ensuring the security and consistency of configured systems. It involves a specification of essentially arbitrary decisions associated with a configuration, using a simple descriptive language; the interpretation of configuration can then be built into dedicated agents. If this language can be compressed into digital (symbolic) strings with high-level interpretations, then it can be implemented as an artificial immune system, i.e. by preprogrammed autonomous agents following 'dumb' algorithms that combine with collective intelligence[5].

The conventional idea of policy-based management is to formulate an expression of what can be done on a computer system, in which situations, and by whom. In other words, it is about using the authorization and implementation of actions, combined with some kind of role-based access control. The present

paper is about a set of principles, collectively referred to as the *immunity model*, whose aim is to make policy-based configuration consistent and implementable by distributed, autonomous agents, even in the face of environmental change. These principles have been explored and developed in a project called GNU cfengine for the past ten years.

Cfengine couches management in dynamical terms, as a competition between forces which tend to disorder systems (sickness), and forces which re-order them (countermeasures). In this respect, the approach is similar to those expounded in other controller-based approaches to system management[6, 7, 8]. Policy schemes generally assume that formal management decisions are sufficient to keep systems in a predetermined state indefinitely, once implemented; they do not take account of the effect of random errors which accrue through usage or by intrusion. The immunity model expects change to occur both through planned revisions of policy and through unplanned events (noise), such as misunderstandings between humans, undisciplined maintenance and even through regular usage. Change which does not conform to policy is defined as sickness and is 'attacked'. The result is an approach to system management based on continuous regulation of system state, somewhat analogous to Shannon's problem of error correction on a noisy channel[9].

# 2 Cfengine

Cfengine is an agent-oriented system for site configuration management, with a language interface, developed at Oslo since the early 1990's[4]. From a recent survey, it is known that cfengine is now installed on hundreds of thousands of Unix and NT systems around the world[10]. Cfengine contains a high level declarative language, which is used to express policy for network and system administration. The policy is an expression of what the immunity model will *tolerate*.

In order to construct policy, cfengine uses a scheme of classification of the possible states in which resources can find themselves. Classes are used to identify necessary actions or responses that direct the system to the attainment of the ideal state. In other words, classes define a healthy state as a finite state machine. Each primitive action that codes a symbol is required to have 'idempotent' behaviour[4, 11, 12, 13], meaning that each action can be repeated an arbitrary number of times and will always terminate with the same result. Sequences of such actions are required to be 'convergent', i.e. they should (through idempotent actions) bring the system always closer to the policy-conformant state.

What makes cfengine different from such similar approaches to configuration management[1, 3] is that it does not assume that transitions between states of its model occur only at the instigation of an operator, or at at the behest of a protocol; cfengine imagines that changes of state occur unpredictably, as symbol errors, at any time and must be dealt with quickly in order to maintain the average state. It then automatically knows how to converge to the correct state,

by virtue of its policy defintion, where this defintion is expressed of convergent operations.

# 3 Policy and its tolerance

A cfengine agent is installed on each host, along with a scheduler which ensures that the agent is run periodically, and a copy of the complete configuration policy. The agent on a given host determines, on the basis of its environment, which rules apply to it and implements them. No matter how often the agent is run, the configuration of any host is only altered if it does not conform to the specifications laid down in the policy rules. A certain number of users will conform to policy, and a certain number will not. Similarly, random errors will occur with a certain probability. There is thus a certain probability that a host will obey policy at a given time, and hence a probabilistic interpretation to system behaviour.

To describe the formal properties of the cfengine model in detail, would require a lengthy discourse. Here we note some key properties of the agent.

**Property 1** *Centralized policy-based specification, with environmental adaptation, using a symbolic language, independent of operating system.*

**Property 2** *Distributed agent-based action: each host node is responsible for its own maintenance.*

**Property 3** *Cfengine actions have the properties of* idempotence *and* convergence*, i.e. the repeated application of a rule leads ends in a stable and predictable state. Once this state has been reached, the agent becomes passive or quiescent until the next measurable deviation arises. This is sometimes referred to as homeostasis.*

The immunity model shares several features with the homoeostatic security model proposed in refs. [14, 11]. In ref. [15] it was shown that a complete specification of policy determines a configuration only on average, over time. There are fundamental limits to the tolerances one can expect a system to satisfy with respect to policy compliance. In the immunity model, the interaction between the forces for change and stability may be seen as a contest, seeking an equilibrium. Drawing on the game-theoretical ideas introduced in ref. [15], cfengine's class-predicated actions may be interpreted as pure strategies in a two-person zero-sum game between the system and the agents. A mixed strategy is a statistical distribution over pure strategies, or cfengine rules. It may be shown from Von Neumann's minimax theorem[16], or from the Nash equilibrium theorem[17], that any two-person zero-sum game has a solution in terms of, at best, a mixed strategy. Cfengine treats system administration as if it were a game in normal or strategic form.

If the average behaviour of a system can be described by a number of discrete symbols, then anomalous events are easily determined and corrected. This is the

lesson of evolutionary stability. Cfengine digitizes policy by introducing action primitives that behave like 'alleles' for configuration.

**Definition 1** *An* action *is an operation executed by an agent. Actions are directed operations which point the system in the direction of the ideal state, from any starting state. An action is carried out by an operator $\hat{O}$, which in turn is constructed from a set of primitive 'transition operators' $\{T_a\}$ (see below) and the latin index a runs over the set of independent primitives (e.g. copy, set attribute1,set set attribute2 etc). The set of all operators $\mathcal{O} \equiv \{T_a\}^*$, contains all sequences (denoted by asterisk) of primitive transition operators.*

As we shall see, generic operators $\hat{O} \in \mathcal{O}$ have too few restrictions to allow predictable behaviour in general. We would like to replace these with a new set C of contrained, convergent operators that lead to more predictable behaviour.

**Definition 2** *A* class *is a label for a set of hosts, unified by a common property. A class becomes an attribute of a particular cfengine invocation, that selects a variation in policy. Some classes are persistent, while others are ephemeral or transient. Classes bind actions to environmental conditions in a context-sensitive way.*

*If $\mathcal{P}$ represents the set of all identifying host properties (like fingerprints or characteristics) with members $p$, $\mathcal{H}$ represents a domain of hosts (with members $h$) to be configured and $\mathcal{O}$ represents all possible operations (with members $\hat{O}$), then the class $\chi$ may be thought of as a function:*

$$\chi : \mathcal{H} \times \mathcal{P} \rightarrow \{\text{True}, \text{False}\}. \tag{1}$$

*Thus a class instance $\chi(h, p)$ is true if host $h$ has property $p$.*

We may further introduce the set of all possible rules $\mathcal{R}$ (with members $r$) that map classes to actions. This describes the set of possible policies. A single policy, from this set, is thus a function:

$$r : \{\chi_1, \chi_2, \ldots, \chi_n\} \rightarrow \mathcal{O}, \tag{2}$$

that maps $n$ classes representing the host properties into a set of actions to be performed. Suppose there are $m$ such actions; then we may write:

$$r(\chi_1, \chi_2, \ldots, \chi_n) = \{\hat{O}_1, \hat{O}_2, \ldots, \hat{O}_m\}. \tag{3}$$

Thus the notion of a ruleset implicitly binds pairs of hosts and their configuration requirements $(h, \hat{O})$ to certain environmental conditions, expressed by the classes. A class is implemented on each host $h \in H$ as the proposition $\chi(h, p)$ is true, in which case it is appropriate to apply the operation $\hat{O}$ to host $H$. In practice, most of the mappings in $\chi$ are 'false' and the small number of 'true' mappings that lead to actions in $r \in \mathcal{R}$ constitutes policy. Examples of classes are given in ref. [4].

We would now like to reserve the term *policy* for rulesets that are composed only of idempotent, convergent operations.

**Definition 3** *A convergent policy $\pi \in \Pi$ is a specification of the desired system configuration over an interval of time which is useful for users of the system[15]. We shall further restrict policy $\pi(h, p, \hat{O})$ by insisting that it be expressed in terms of convergent actions $\{\hat{C}_a\} = C(\{T_a\})$. This subtlety is described in the remainder of the paper.*

The general aim here is to begin with all primitive change operators $T_a$, that can be combined in linear combinations to form $\hat{O}$, and then whittle away this set with constraints until we end up with $\hat{C}_a$, a set of operators that obey policy and have idempotent and convergent behaviour. Note, the objects $\hat{C}_a$ depend on policy and state; we shall often suppress this dependency in the notation to avoid typographical complexity.

# 4 Strategy

As an appendix to previous section, one can add that the most general combination of such operations involves a schedule as well as a conditional property constraint. One can, however, easily implement this as time constraints by including time information in the classing scheme. This is the approach used by cfengine. This allows a connection with classical Game Theory and optimization (see ref. [15] and [18]) with the following additional concepts.

**Definition 4** *A pure strategy is a complete sequence of actions in a policy, which provides one possible route to an acceptable ideal configuration. Let us use the index $A$ to denote a member of this equivalence class of $\pi$ that lead to the same configuration. A pure strategy is then a sequence $\hat{C}^*_A(\pi)$, for some $A$, of operations $\hat{C}_a(\pi)$, belonging to policy $\pi \in \Pi$. The $\hat{C}_a(\pi)$ are primitive operations with convergent properties.*

As long as we have equivalent operations, it does not matter which of them we choose to implement policy. In general a probabilistic mixture can be created. This has some potential scheduling and security advantages over use of a single member of the class (see ref. [19]).

**Definition 5** *A mixed strategy $\hat{\sigma}$ (usually denote by $\sigma$ in the game theory literature[20]) is a group of strategies, labelled by different $A$, which may be chosen at random, with probability $p_A$, to achieve the same end as using distinct sequences:*

$$\hat{\sigma} = \sum_A p_A \hat{C}^*_A(\pi), \tag{4}$$

*where $\hat{C}_a(\pi)$ is a convergent, idempotent operation belonging to the policy $\pi \in \Pi$. Each alternative provides an acceptable, but different route to the policy configuration state.*

It is with some subtlety that we define a strategy in terms of the subset of convergent operations, rather than the unconstrained sets. The explanation follows in the remainder of the paper. Mixed strategies are used to schedule maintenance operations optimally or to make the system unpredictable to would-be attackers: this includes users of the system who seek to confound policy through their own actions.

# 5 Proof of convergence to a stable state

To explain the significance of the operators $\hat{C}_a$, let us begin again more systematically and build up the concept in terms of the necessary constraints of the system. These constraints enter formally as the classes $\chi$.

Corrections made to a host configuration should lead towards a definite state and any constructive or counter-measures should terminate after a small number of iterations. The route countermeasures take through state space should be unidirectional. If this were not the case, then contradictions and non-terminating cycles could result. Cfengine addresses such convergence in two ways: by making each successful sequence of actions convergent in a single step (idempotence), and by checking for contradictory sequences. If a single step should fail or be undermined, for what ever reason (crash, interruption, changing conditions, loss of connectivity etc), it can be completed later; this is sufficient to ensure that simple configurations converge.

We now prove how this works, using a linear representation of configuration vectors, acted on by matrix-value operators. If two operations are *orthogonal*, it means that they can be applied independently of order, without affecting the final state of the system (they might not succeed unless an ordering is followed, but we defer this for now). Using vectors and matrices, this is equivalent to requiring the commutativity of countermeasure operations. To formalize these points, we require a notation for the state of the system and for the operations. Let the notation

$$|a, b, c, \ldots\rangle \tag{5}$$

represent the configuration vector of the system, and let $T_a$ be a set of transition operators that may act on this state vector to effect changes. The matrices $T_a$ commute, i.e. their commutator bracket vanishes:

$$[T_a, T_b] \equiv T_a T_b - T_b T_a = 0, \tag{6}$$

and they are linearly independent, i.e.

$$\sum_{a=0}^{\ell} \alpha^a T_a = 0 \ \Rightarrow \alpha^a = 0, \ \forall a, \tag{7}$$

meaning that none of the matrices is equivalent to a combination of any of the others. Indeed, we may always take them to be orthogonal:

$$T_a T_b = 0, \ a \neq b. \tag{8}$$

At the most basic level, the configuration vector is the set of all bits characterizing the memory of the system; however, at the level of the operating system, this memory is coded into higher level structures, such as files and processes which have attributes. At this level of abstraction, the vector may be thought of comprising flags which signify the existence and internal attributes of these objects.

Since the objects comprising a host (files and processes etc.) cannot overlap, they form a clean partitioning of the vector into subsystems. Thus we may write a column vector representation for $|a, b, c, \ldots\rangle$, so that each attribute of each object is a row in this vector:

$$|a, b, c, \ldots\rangle = \begin{pmatrix} a \\ b \\ c \\ \vdots \end{pmatrix} \tag{9}$$

Each independent attribute $a$ has its own linearly-independent operator $T_a$ which can alter the value.

The operators are sparse matrices which 'increment' or 'decrement' the individual attributes. In some cases it is useful to formally separate by sign and orthogonal direction, the operators which are considered to be "do" operations and those which are considered to be "undo" operations. This is an arbitrary choice. These are denoted

$$T_a = \left\{T_a^+, T_a^-\right\}, \tag{10}$$

and have automatic idempotence, i.e. $T_a^+ T_b^+ = 0$ for all $a, b$. These are sometimes called creation ($T_a^+$) and annihilation ($T_a^-$) operators, because they have opposing effects, and can be viewed as moving the system through an abstract lattice of configurations. With this linear representation of configuration operators acting on sparse matrices that multiply a vector, the combination of operators may be viewed either as a linear addition, or as a multiplication.

Operators which increment or decrement (translate) the value of a variable belong to the general inhomogeneous group of transformations. In order to represent an inhomogeneous group in a pure matrix form, one may embed the configuration vector in extra dimensions, one for each type of object. This may be illustrated by a simple example of two objects of the same type. Let

$$|a, b\rangle = \begin{pmatrix} a \\ b \\ 1 \end{pmatrix} \tag{11}$$

where the 1 is the incremental value which can be applied to either $a$ or $b$. The operators on this configuration may now be written in the generic form of an identity matrix, modified by a combination of primitive transformations:

$$\hat{O} = I + \sum_a \lambda_a T_a^+. \tag{12}$$

$a$ ranges over the independent operations on the independent objects; $\lambda^a$ is a constant indicating the value to be added and $T_a$ is called the generator of the operation. It is a sparse matrix with only a single element. Here, one has

$$T_1^+ = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, T_2^+ = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{13}$$

One may now verify the property that combination of operations is orthogonal and leads to the equivalence of addition of generators $T_a^+$ with the multiplication of operators $O_a$: of the

$$\hat{O}_1 = I + \lambda_1 T_1^+ \tag{14}$$
$$\hat{O}_2 = I + \lambda_2 T_2^+ \tag{15}$$
$$\hat{O}_{1+2} = I + (\lambda_1 T_1^+ + \lambda_2 T_2^+) \tag{16}$$

and

$$\hat{O}_1 \hat{O}_2 |a, b\rangle = \hat{O}_2 \hat{O}_1 |a, b\rangle = \hat{O}_{1+2} |a, b\rangle, \tag{17}$$

i.e.

$$\begin{pmatrix} 1 & 0 & \lambda_1 \\ 0 & 1 & \lambda_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = \begin{pmatrix} a + \lambda_1 \\ b + \lambda_2 \\ 1 \end{pmatrix}. \tag{18}$$

To generate $|a-1, b-1\rangle$ etc., one only changes the sign of $\lambda_a$. These operations are easily implemented in terms of system calls on the hosts. Readers should note, however, that while the editing of a file's contents becomes a somewhat messy operation to represent in formal terms, there is no ultimate technical impediment to doing so, though our discussion here is too primitive to make it convincing.

Consider now, how configurations are built up. Let the state $|\mathbf{0}\rangle$ denote the base-state configuration vector of a host after installation. This is a reference state to which any host may be returned by re-installation. From this state, one may build up an arbitrary new state $|a, b, c, \ldots\rangle$ through the action of sequences of the creation and annihilation operators:

$$\begin{aligned} |a, b, c\rangle &= (I + aT_1^+ + bT_2^+ + cT_3^+)|\mathbf{0}\rangle \\ &= \hat{O}(a, b, c)|\mathbf{0}\rangle. \end{aligned} \tag{19}$$

The set $\{a, b, c\}$ is essentially a matrix representation of the rule set $r \in \mathcal{R}$. It *could* be regarded as the system policy specification, but this would be unsatisfactory without a condition that, once a desirable state had been reached, one renormalized these definitions to allow $|\mathbf{0}\rangle$ to be the new base-state (this is what cfengine does). In other words, we need to code, somehow, the fact that the configuration process stops once the policy base-state is reached.

A better representation of policy that codes this idempotence is to define a new set of operators $\{C_a(T_a^+)\}$ that are 'absorbing' (in the sense of a chain or semi-group). Using this representation, one can now define the meaning of convergence.

**Definition 6** *Let $|s\rangle$ be an arbitrary state of the system. An operator $\hat{C}_a$ is said to be convergent if it has the property*

$$\begin{aligned} (\hat{C}_a)^n|\mathbf{s}\rangle &= |\mathbf{0}\rangle \\ \hat{C}_a|\mathbf{0}\rangle &= |\mathbf{0}\rangle, \end{aligned} \tag{20}$$

*for some integer $n > 1$, i.e. the n-th power of the operation is null-potent. For $n = 1$ this is the same as idempotence.*

In other words, a convergent operator has the property that its repeated application will eventually lead to the base state, and no further activity will be registered thereafter. This requires a slight modification of the operators, $\hat{O}$, described above, since the base state must be checked for explicitly.

**Theorem 1** *Modified convergent operators can always be written by introducing a linear dependency on the value of the vector $|v\rangle$ operated on, with representation:*

$$\hat{C}(|v\rangle) = I + \sum_a \theta_a \left( \left| |v\rangle \right| \right) \lambda_a T_a^+. \tag{21}$$

*The precise representation of the operators depends on the coding level of the system (see discussion in ref. [15]) and the choice of a basis at that level. However, once this is chosen, the $T_a$ are simply generators of the translation group.*

The vertical bars around the vector represents taking the absolute value of each vector component (not to be confused with the length of the vector), and $\theta(x)$ is the Heaviside step function, with subscript denoting its action on the $a$th component. It is defined by

$$\theta_a \left( \left| |v\rangle \right| \right) = \begin{cases} 1 & (v_a \neq 0) \\ 0 & (v_a = 0). \end{cases} , \tag{22}$$

where $v_a$ is the $a$th component of the state vector.

**Proof 1** *The proof is rather simple, from the definition of the stepping operators $T_a^+$. We note first that $(T_a^+)^2 = 0$, for all $a$, hence, from either the binomial theorem, or by inspection:*

$$(I + \sum_a \lambda_a T_a^+)^n = I + n \sum_a \lambda_a T_a^+. \tag{23}$$

*Thus, group theoretically, the $T_a^+$ have the form of infinitesimal translation generators satisfying the Chapman-Kolmogorov equation (eqn. 17). The addition of the Heaviside distribution simply turns this into a semi-group with respect to the base vector $|\mathbf{0}\rangle$. $\hat{C}^n$ now generates no more than $n$ steps towards this base vector, and switches of each $T_a^+$ by multiplying by zero as soon as $v_a = 0$.*

This modification preserves the essential properties of commutation and orthogonality, and we *assume* that the raising of the operator by powers implies iteration, rather than the construction of a new composite operator, since it is the repeated action of 'infinitesimals' that gives us the required properties.

A set of operations $\{\hat{C}\}$ with the above properties is a solid grounding for securing convergence in all configurations, but it is not necessarily sufficient to secure convergence of sequences of operations.

**Theorem 2** *The combination of two or more convergent operators is not necessarily convergent.*

**Proof 2** *The problem arises because the notion of convergence is relative to a specific base state. If the base state itself is modified as a result of the operations, then it is possible that the base state will never be reached, because it represents a moving target. To see this, consider two operators which refer to different base states* $|0_1\rangle$ *and* $|0_2\rangle$*. These may be written*

$$\begin{aligned} \hat{C}_1 &= I + \sum_a \theta_a \left( \Big| |v\rangle - |0_1\rangle \Big| \right) \lambda_a T_a^+ \\ \hat{C}_2 &= I + \sum_b \theta_b \left( \Big| |v\rangle - |0_2\rangle \Big| \right) \lambda'_b T_b^+. \end{aligned} \tag{24}$$

*The product of these is*

$$\begin{aligned} \hat{C}_1 \hat{C}_2 = I &+ \sum_a \left[ \theta_a \left( \Big| |v\rangle - |0_1\rangle \Big| \right) \lambda_a + \theta_a \left( \Big| |v\rangle - |0_2\rangle \Big| \right) \lambda'_a \right] T_a^+ \\ &+ \sum_a \sum_b \theta_a \left( \Big| |v\rangle - |0_1\rangle \Big| \right) \theta_b \left( \Big| |v\rangle - |0_2\rangle \Big| \right) \lambda_a \lambda'_b T_a^+ T_b^+. \end{aligned} \tag{25}$$

*Thus, the commutation results in, the difference between the states:*

$$[\hat{C}_1, \hat{C}_2]|\mathbf{s}\rangle = |0_1\rangle - |0_2\rangle \neq 0. \tag{26}$$

We must therefore seek to disallow such contradictions. Cfengine's configuration file defines the base state in terms of convergent operators. These operators contain a knowledge of the base state by their implicit dependence on the current state via the Heaviside step function. This is required for convergence, but it also allows the specifier of policy to code rules (operators $\hat{C}$ and $\hat{C}'$) which have contradictory notions of what $|\mathbf{0}\rangle$ is. This can lead to strings of operations which can "do" and "undo" the state of the system.

**Definition 7** *Two convergent operators* $\hat{C}_1$ *and* $\hat{C}_2$ *are non-contradictory if they satisfy*

$$\begin{aligned} \hat{C}_1|\mathbf{0}\rangle &= |\mathbf{0}\rangle \\ \hat{C}_2|\mathbf{0}\rangle &= |\mathbf{0}\rangle, \end{aligned} \tag{27}$$

*i.e. if they terminate on the same state.*

The aim is thus to ensure that instances of operators, which do not satisfy this property, do not occur in a policy specification. Note that this is the only form of contradiction that can occur, since the orthogonality guarantees that there is no interdependence between the $T_a^+$. The use of orthogonal decomposition is thus an powerful tool for finding inconsistency. It reduces a potentially complex graph theoretical problem into a set of decoupled, trivial graphs.

Cfengine uses the notions described here in the following way:

1. It provides only operators of the form $\hat{C}$, which converge towards some definite state.
2. It uses orthogonal decomposition to detect possible cycles.
3. It does not disallow cycles belonging to different instances of time, since it is possible that a dynamic, time-dependent policy would change with respect to time. That is a decision to be made by humans.

# 6 Commutation and pre-requisite dependency

The ordering of operators, belonging to a fixed configuration policy, is an issue which needs to be resolved. Can the order in which operations are carried out lead to a difference in the final state? Cfengine goes against tradition, in configuration management, by specifying a final state, without regard to the ordering of the steps along the way. This is only possible because of the notion of convergent operators. This was pointed out recently by Couch[21] in a similar study.

The use of orthogonal, convergent operations implies that only one type of prerequisite dependency can occur. For example, let $\hat{C}_{\rm C}$ mean 'create object' and let $\hat{C}_{\rm A}$ mean 'alter object attribute'. The following operations do not commute, because the setting of an attribute on an object requires the object to exist. On an arbitrary state $|\mathbf{s}\rangle$, we have

$$[\hat{C}_{\rm C}, \hat{C}_{\rm A}]|\mathbf{s}\rangle \neq 0. \tag{28}$$

Thus the ordering does indeed matter for the first iteration of the configuration tool. This error will, however, be automatically corrected on the next iteration, owing to the property of convergence. To see that the ordering will be resolved, one simply squares any ordering of the above operations.

**Theorem 3** *The square of a create-modify pair, belonging to the same policy $\pi$, is order independent.*

$$([\hat{C}_{\rm C}, \hat{C}_{\rm A}])^2|\mathbf{s}\rangle = 0. \tag{29}$$

This result is true because the square of these two operators will automatically result in one term with the correct ordering. Orderings of the operators in the incorrect order are ignored due to the convergent semantics.

**Proof 3** *Suppose that the correct ordering (create then set attribute) leads to the desired state* $|\mathbf{0}\rangle$*:*

$$\hat{C}_{\rm A}\hat{C}_{\rm C}|\mathbf{s}\rangle = |\mathbf{0}\rangle; \tag{30}$$

*performing the incorrect ordering twice yields the following sequence:*

$$\hat{C}_{\rm C}\underbrace{\hat{C}_{\rm A}\hat{C}_{\rm C}}\hat{C}_{\rm A}|\mathbf{s}\rangle = |\mathbf{0}\rangle. \tag{31}$$

*The action of* $\hat{C}_{\rm A}$ *has no effect, since the object does not exist. The under-brace is the correct sequence, leading to the correct state, and the final* $\hat{C}_{\rm C}$ *acting on the final state has no effect, because the system has already converged. Hence we may write:*

$$\begin{aligned} C_{\rm A}|\mathbf{s}\rangle &= |\mathbf{s}\rangle \\ \hat{C}_{\rm C}|\mathbf{0}\rangle &= |\mathbf{0}\rangle \end{aligned} \tag{32}$$

*and thence:*

$$\begin{aligned} ([\hat{C}_{\rm C},\hat{C}_{\rm A}])^2|\mathbf{s}\rangle &= \hat{C}_{\rm C}\hat{C}_{\rm A}\hat{C}_{\rm C}\hat{C}_{\rm A} - \hat{C}_{\rm C}\hat{C}_{\rm A}\hat{C}_{\rm A}\hat{C}_{\rm C} - \hat{C}_{\rm A}\hat{C}_{\rm C}\hat{C}_{\rm C}\hat{C}_{\rm A} + \hat{C}_{\rm A}\hat{C}_{\rm C}\hat{C}_{\rm A}\hat{C}_{\rm C} \\ &= \hat{C}_{\rm A}\hat{C}_{\rm C} - \hat{C}_{\rm A}\hat{C}_{\rm C} - \hat{C}_{\rm A}\hat{C}_{\rm C} + \hat{C}_{\rm A}\hat{C}_{\rm C} \\ &= 0, \end{aligned} \tag{33}$$

*completing the proof.*

The same property is true of sequences of any length, as shown by Couch[21]; in that case, convergence of $n$ operations is assured by a number of iterations less than or equal to $n$. We may refer to this property as Couch and Daniels' Maelstrom theorem [21], after the authors who generalized cfengine's method to more complex dependencies:

**Theorem 4** *A sequence of n self-ordering operations is convergent in n iterations, i.e. is of order* $n^2$ *in the primitive processes.*

The proof may be found by extending the example above, by induction. In cfengine sequences of larger than order 2 do not occur at the primitive level, owing to the convergence of the primitives. Only the exist-modify orderings are important. At a higher level, however, the completion of one operation can trigger subsequent operations. The property is carried through to all sequences, provided there are no contradictions, because the construction of operators is such that any operator can be reduced to order 2 exist-modify orderings.

It follows as a corollary to theorem 3, that the square of any two convergent operators commutes with the square of any others, since the linear independence of objects requires any two such pairs to refer either to independent objects or be a repetition of an already converged operation.

# 7 Paths and sequences

So far the operational algebra has been defined for constant policy. In a cfengine program, artificial dependencies can be introduced into policies, by dynamically modifying a host's properties as a result of the completion of a primitive operation $P$ and hence classes $\chi$, such that the successful completion of an operation leads to a dependent follow-up action. These is sometimes referred to as 'feedback' classes.

Let us refer to such dependencies as being *constructed* dependencies. Constructed dependencies lead to sequences whose lengths are predicated on the actual path taken to the final state – they explicitly defy the idempotence and commutation property that characterize the primitive operators $\hat{C}_a$ because they contain hidden variables: namely, the current policy of the system after each previous operation. By allowing the predicates for policy to change at each step, they prevent the commutation of the $\hat{C}_a(\pi)$. There is thus a potential *path-dependence* that must be addressed.

This might seem to introduce a problem for the notion of consistent convergence, but this is not the case due to the orthogonality and constraints of the primitive operations. The effect of this is only to slow the rate of convergence from a single idempotent operation to a longer sequence.

To prove that a given sequence will converge regardless of the additional ordering requirement, we make the argument in two steps. To begin with, we note that the issue of creation and modification operations leads to a potential cycle length of two.

**Lemma 1** *Any sequence of orthogonal, convergent operations, $C^*(N|\pi)$, at constant policy $\pi$ and of length $N$, that is free of simple contradictions, can be implemented consistently (will converge) within $t$ iterations of the entire sequence, where $0 < t \leq 2$, i.e. between $n$ and $2n$ primitive operations.*

**Proof 4** *Any configuration string $S(2)$, free of constructed dependencies, is ordered within two iterations, by virtue of eqn. (29). The presence of constructed dependencies only extends the number of convergent actions performed, possibly with respect to different objects. Repeated operations can be ignored. Thus,*

$$\begin{aligned} S(2N)|\mathbf{s}\rangle &= (\hat{C}_1\hat{C}_2\ldots\hat{C}_{N-1}\hat{C}_N)^2|\mathbf{s}\rangle \\ &= ((\hat{C}_1\hat{C}_2)\ldots[C_i,\hat{C}_j]\ldots)\ldots C_N|\mathbf{s}\rangle, \end{aligned} \tag{34}$$

*since we may freely insert the commutator of any two operators (zero action) at any place, from eqn. (29). Thus we may re-organize the string into pairs of operations whose squares commute and into the remainder of operations that commute by themselves, by repeated application of eqn. (29) the squared string. Suppose that operators 1 to $j$ have create-alter issues, and $j+1$ to $N$ do not. We have:*

$$\begin{aligned} S(2N)|\mathbf{s}\rangle &= ((\hat{C}_{A_1}\hat{C}_{C_1})^2(\hat{C}_{A_2}\hat{C}_{C_2})^2\ldots(C_{j+1}^2C_{j+2}^2\ldots C_N^2))|\mathbf{s}\rangle \\ &= (S(N))^2|\mathbf{s}\rangle \\ &= |\mathbf{0}\rangle. \end{aligned} \tag{35}$$

*It is assumed here that there are no simple contradictions, so none of the operators cancel one another; thus, by construction, consistency is assured, since the square of any convergent operator commutes with the square of any other.*

This property is sufficient to bound the convergence time of any contradiction free configuration string with constructed dependencies.

Define the notation $\hat{C}(\pi_n, \pi_{n-1}, \ldots, \pi_2, \pi_1, a_n, \ldots, a_1)$ for the operator that implements a constructed dependency, whose ordering is defined by the ordered intermediate states $(\pi_1, \pi_2, \ldots, \pi_n)$ of length $n$:

$$\hat{C}(\pi_n, \pi_{n-1}, \ldots, \pi_2, \pi_1, a_n, \ldots, a_1)|s\rangle = \hat{C}_{a_n}\hat{C}_{a_{n-1}}\ldots\hat{C}_{a_2}\hat{C}_{a_1}|\mathbf{s}\rangle, \tag{36}$$

where

$$\begin{aligned} \hat{C}_{a_1}|\mathbf{s}\rangle &= |\mathbf{s}_1\rangle \\ \hat{C}_{a_1}|\mathbf{s}_1\rangle &= |\mathbf{s}_2\rangle \\ \hat{C}_{a_{n-1}}|\mathbf{s}_1\rangle &= |\mathbf{s}_n\rangle, \end{aligned} \tag{37}$$

etc. The constructed dependency can therefore be expressed in terms of ordinary idempotent-convergent operators, which either commute or whose squares are known to commute. We may therefore bound the convergence time straightforwardly:

**Theorem 5** *A constructed dependency of length $N > n \geq 1$, in which policy changes $n$ times as a chained dependency, will converge in $t$ iterations of the entire sequence, where $0 < t < n$ (for $n > 1$) and $0 < t < 2$ (for $n = 1$), i.e. it is of order no more than $n^2$ in the primitive operations.*

**Proof 5** *If a configuration sequence has $N$ operators and $n < N$ of them result in a stepwise change of policy, then a maximum of $N - n$ of the operators has the potential for create-alter dependencies. Since these occur at constant policy, they can be resolved in one or two iterations of the string, regardless of $N$. Thus, in the worst case where all operators are evaluated in the reverse order, $n^2$ operations will suffice (since $n^2 \geq 2$ for any non-trivial sequence, thus the constant policy orderings will be included in the policy altering transitions). In the best case, in which all the policies transitions are ordered, one or two iterations at most will suffice, from the previous theorem.*

It is noteworthy that this result is independent of $N$, or the size of one's policy. Thus, it is not the complexity of policy that bounds the likelihood of correct configuration, but the number of artificial dependencies that one adds. One therefore concludes that coding specific orderings and pathways into a policy (as is advocated in ref. [22]) is to be avoided, especially when repeated iteration of policy control is not exercised.

Two important properties of any configuration system are reproducibility of result, and predictable behaviour on failure. Schemes for ensuring such consistency have been discussed, for instance, in refs. [23, 24]. Cfengine addresses

these issues in several ways. Once again, the property of convergent behaviour is central to reproducibility. Predictability on failure and the principle of minimal disruption are ensured algorithmically.

The long term evolution of the system is thus irrelevant to the ability to order the system, because the operators are defined to be idempotent and convergent. Convergent constraints are a compact way of encoding highly compressed information that maintains order, without the need to retain or analyze the entire history of the system's evolution.

# 8 Discussion

The cfengine model has been developed, used and evaluated extensively for ten years to approximate the foregoing model as closely as is practically possible. Complete adherence to the notions of primitive operations, idempotence and convergence cannot always be guaranteed because other systems (shell scripts and system commands), that are used for convenience, do not obey such constraints. This means that the spectre of hidden constructed dependencies can still haunt actual usage.

The unique aspect of cfengine is its notion of convergence, coupled with a description of end-state, rather than the specific configuration path. Although there have been many autonomous agent-based systems for system management, no others have a provably convergence at the configuration level, to the author's knowledge. Controller regulation schemes such as those in refs. [6, 8] work at the statistical level, but cfengine combines both statistical and symbolic regulation.

This closes the chapter on cfengine's foundations with a proof of the convergent symbolic properties that make cfengine unique. There is not room here to make extensive comparisons with other approaches, nor are there many similar approaches except at the superficial level, but it is interesting to note a similarity to the evolving philosopher model described by Kramer and Magee[23]. This work details methods for maintaining consistency of operation in the face of dynamical change. These authors refer to modifications and extensions of the system which were not envisaged at design time. While the authors do not explicitly mention responding to stochastic changes from unpredictable sources, their model encapsulates such changes also to a certain degree. The use of declarative languages for the purpose of configuration description has been considered by many authors[1, 2, 25, 4].

Convergence to a dynamic equilibrium is central to cfengine behaviour. Health is defined as a matter of policy combined with machine-learned normality. In recent work, cfengine has been combined with statistical methods for detection and tolerization of system resource measurements[26], as well as an explicit link to the Linux kernel symbolic anomaly detector pH[27] developed at the University of New Mexico[28]. The theoretical model presented in refs. [15] points to the need for strategic feedback in policy determination, in order to build stable policies around the Nash equiibria of the system. Measurements of the interaction between system and environment can provide such

feedback. Significant study still remains to bring the combination of long-term and short-term information to fruition. These issues will be revisited in future work.

Note added: an older version of this work appeared on the cfengine web site in February of 2001. I am grateful to Alva Couch for constructive criticism.

# References

[1] B. Hagemark and K. Zadeck. Site: a language and system for configuring many computers as one computer site. *Proceedings of the Workshop on Large Installation Systems Administration III (USENIX Association: Berkeley, CA, 1989)*, page 1, 1989.

[2] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, **2**:333, 1994.

[3] D. Marriott and M. Sloman. Implementation of a management agent for interpreting obligation policy. *Implementation of a management agent for interpreting obligation policy*, IFIP/IEEE 7th international workshop on distributed systems operations and management (DSOM), 1996.

[4] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Oxford, 1999.

[6] P. Hoogenboom and J. Lepreau. Computer system performance problem detection using time series models. *Proceedings of the USENIX Technical Conference, (USENIX Association: Berkeley, CA)*, page 15, 1993.

[7] J.L. Hellerstein, F. Zhang, and P. Shahabuddin. An approach to predictive detection for service management. *Proceedings of IFIP/IEEE IM VI*, page 309, 1999.

[8] Y. Diao, J.L. Hellerstein, and S. Parekh. Optimizing quality of service using fuzzy control. *IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002)*, page 42, 2002.

[9] M. Burgess. System administration as communication over a noisy channel. *Proceedings of the 3nd international system administration and networking conference (SANE2002)*, page 36, 2002.

[10] M. Burgess. Evaluation of cfengine's immunity model of system maintenance. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, 2000.

[11] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *New Security Paradigms Workshop, ACM*, September 1997:75–82.

[12] M. Burgess. Computer immunology. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.

[13] A. Couch and M. Gilfix. It's elementary, dear watson: Applying logic programming to convergent system management processes. *Proceedings of the Thirteenth Systems Administration Conference (LISA XIII) (USENIX Association: Berkeley, CA)*, page 123, 1999.

[14] P.D'haeseleer, Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis, and implications. *In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy (1996).*

[15] M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003.

[16] J.V. Neumann and O. Morgenstern. *Theory of games and economic behaviour.* Princeton University Press, Princeton, 1944.

[17] J.F. Nash. *Essays on Game Theory.* Edward Elgar, Cheltenham, 1996.

[18] M. Burgess. *Theory of Network and System Administration.* J. Wiley & Sons, Chichester, 2004.

[19] M. Burgess and F.E. Sandnes. Predictable configuration management in a randomized scheduling framework. *IFIP/IEEE 12th International Workshop on Distributed Systems: Operations and Management (DSOM 2001)*, page 293, 2001.

[20] R.B. Myerson. *Game theory: Analysis of Conflict.* (Harvard University Press, Cambridge, MA), 1991.

[21] A. Couch and N. Daniels. The maelstrom: Network service debugging via "ineffective procedures". *Proceedings of the Fifteenth Systems Administration Conference (LISA XV) (USENIX Association: Berkeley, CA)*, page 63, 2001.

[22] 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.

[23] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, **16**:1293, 1990.

[24] K.M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. *Proc. of 3rd International Conference on Configurable Distributed Systems (CDS '96), Annapolis, Maryland, USA*, IEEE Computer Society Press:62, 1996.

[25] P. Anderson. Towards a high level machine configuration system. Proceedings of the Eighth Systems Administration Conference (LISA VIII) (USENIX Association: Berkeley, CA):19, 1994.

[26] M. Burgess. Two dimensional time-series for anomaly detection and regulation in adaptive systems. *IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002)*, page 169, 2002.

[27] A. Somayaji and S. Forrest. Automated reponse using system-call delays. *Proceedings of the 9th USENIX Security Symposium*, page 185, 2000.

[28] K.M. Begnum and M. Burgess. A scaled, immunological approach to anomaly countermeasures. *Proceedings of the VII IFIP/IEEE IM conference on network management*, 2003.