A TINY OVERVIEW OF CFENGINE: CONVERGENT MAINTENANCE AGENT

Mark Burgess
Oslo University College
P.O. Box 4, St. Olavs Plass, Oslo 0230, Norway
Email: mark@iu.hio.no

Keywords: Autonomous agents, network management, feedback systems.

Abstract: Cfengine is a distributed agent framework for performing policy-based network and system administration that is used on hundreds of thousands of Unix-like and Windows systems. This paper describes cfengine's stochastic approach to policy implementation using distributed agents. It builds on the notion of 'convergent' statements, i.e. those which cause agents to gravitate towards an ideal configuration state, which is implied by policy specification. Cfengine's host classification model is briefly described and the model is compared to related work.

1 INTRODUCTION

Cfengine is an on-going research project, looking at distributed system administration. Since its inception in 1993, the cfengine tool-set has been adopted by a broad range of users from small businesses to huge organizations(Burgess, 2000a). It is currently running on an estimated several hundred thousand nodes around the world. Since its inception, cfengine has developed considerably, although the basic framework and key principles have remained the same.

Cfengine falls into a class of approaches to system administration which is called policy-based configuration management (Sloman and Moffet, 1993). To some extent, policy-based configuration can be seen as a reaction to the inadequacies of control and monitoring software, now typified by many Simple Network Management Protocol (SNMP) implementations. Although developed independently, cfengine has developed many features in common with multi-agent systems (Ferber, 1999). Administrative schemes, employing autonomous agents, are the only administrative solutions which scale to large numbers of hosts (Burgess, 1998), because this does not rely on the funnelling of configuration instructions through a serial queue, governed by a human. Today, the approach is used by many large organizations to manage anything from a handful to thousands of systems.

2 KEY IDEAS IN CFENGINE

Cfengine's task is to configure the files and processes running on networked computers, e.g. Unix or Windows workstations.

- *Policy* (P) is a description of intended host configuration. It comprises a partially ordered list of operations or tasks for an agent to check.
- *Operators* ($\hat{O}$) or primitive skills/*actions* are the commands that carry out maintenance checks and repairs. They are the basic sentences of a cfengine program. They describe *what* is to be constrained.
- *Classes* are a way of slicing up and mapping out the complex environment into discrete ('digital') regions that can then be referred to by a symbol or name. They are formally constraints on the degrees of freedom available in the system parameter space. They are an integral part of specifying rules. They describe *where* something is to be constrained.
- A cfengine *state* is a fuzzy region within the total system parameter space. It is defined formally with symbols *classes* that define the environment in which a policy rule lives and by the specificity of the policy rules themselves with respect to the internal characteristics of the operators (e.g. file permissions, process characteristics). States have the form: `(address,constraint) = (class,values)`

Cfengine differs from similar approaches to configuration management in that it embraces a stochastic model of system evolution. Rather than assuming 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 at any time, as part of the environment to be discovered.

The cfengine project and derivative work (Burgess, 2003) accepts the idea of randomness in the interaction with environment. User interaction (Burgess et al., 2001) forms a mixture of signals which tends to disorder the system configuration, over time. Cfengine holds to a set of principles, referred to as the *immunity model* (Burgess, 2004b), for seeking correctness of configuration. These embody the following features:

- Centralized policy-based specification, using an operating system independent language.
- Distributed agent-based action; each host agent is responsible for its own maintenance.
- Convergent semantics encourage every transaction to bring the system closer to an 'ideal' average-state, like a ball rolling into a potential well.
- Once the system has converged, action by the agent desists.

In an analogous way to the healing of a body from sickness, cfengine's configuration approach is to always move the system closer to a 'healthy' state (Burgess, 1998), or oppose unhealthy change: hence the name 'immunity model'. This idea shares several features with to the security model proposed in refs. (P.D'haeseleer et al., ; Somayaji et al.,). Convergence is described further below.

A 'healthy state' is defined by reference to a local policy. When a system complies with policy, it is healthy; when it deviates, it is sick. Cfengine makes this process of 'maintenance' into an error-correction channel for messages belonging to a fuzzy alphabet (Burgess, 2002a), where error-correction is meant in the sense of Shannon (Shannon and Weaver, 1949).

In ref. (Burgess, 2003) it was shown that a complete specification of policy determines an approximate configuration of a software system only approximately over persistent times. There are fundamental limits to the tolerances a system can satisfy with respect to policy compliance in a stochastic environment.

3 COMPONENTS

The main components of cfengine are (see table 1 and fig. 1):

Component	Cfengine 1.x	Cfengine 2.x
Agent	cfengine	cfagent
Server	cfd	cfservd
Scheduler	cron,cfwrap	cfexecd
Poller	cfrun	cfrun
Key Gen	cfkey	cfkey
Long term state	–	cfenvd
State grapher	–	cfenvgraph

Table 1: Components in cfengine

- A central repository of policy files, which is accessible to every host in a domain.
- A declarative policy interpreter (cfengine is not an imperative language but has many features akin to Prolog (Couch and Gilfix, 1999)).
- An active agent which executes intermittently on each host in a domain.
- A secure server which can assist with peer-level file sharing, and remote invocation, if desired.
- A passive information-gathering agent which runs on each host, assisting in the classification of host state over persistent times.

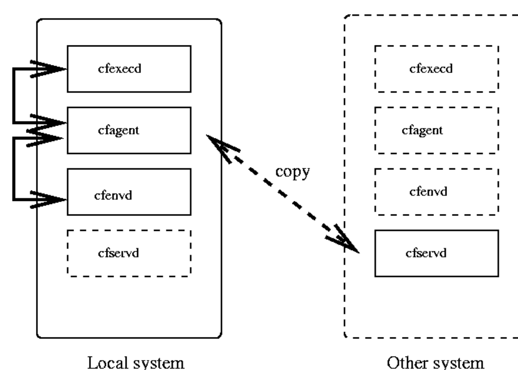

Figure 1: Cfengine components.

4 CLASSES AND ENVIRONMENT

Setting configuration policy for distributed software and hardware is a broad challenge, which must be addressed both at the detailed level, and at the more abstract enterprise level. Cfengine is deployed throughout an environment and *classifies* its view of the world into overlapping sets. Those tasks which overlap with a particular agent's world view are performed by the agent.

A class based decision structure is possible because each host knows its own name, the type of operating system it is running and can determine whether it belongs to certain groups or not. Each host which runs a cfengine agent therefore builds up a list of its own attributes (called the classes to which the host belongs). Some examples include:

1. The identity of a machine, including hostname, address, network.
2. The operating system and architecture of the host.
3. An abstract user-defined group to which the host belongs.
4. The result of any proposition about the system.
5. A time or date.
6. A randomly chosen strategy element.
7. The logical combination of any of the above, with AND (.), OR (|), NOT (!) and parentheses.

The environment is large and complex and we cannot describe it in precise terms, so cfengine classifies it into coarse abstract properties that are suitable for management purposes. The classifiers form a patchwork covering of the environment (see fig 2). Given

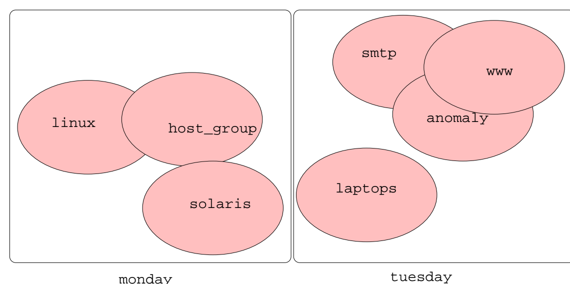

Figure 2: Overlapping classes form a covering of the environment parameter space.

that the agent, running on a host, can determine the class attributes for that environment, it can now pick out what guidelines it needs from a globally specified policy, since each policy task is also labelled with the classes to which it applies. This policy predicates the agent's application of skills according to broad criteria, encompassing distributed collaborations.

A command or action is only executed if a given host is in the same class as the policy action in the configuration program. There is no need for other formal decision structures, it is enough to label each statement with classes. For example:

```
  linux::   linux-actions
solaris::   solaris-actions
```

More complex combinations can perform an arbitrary covering of a distributed system (Comer and Peterson, 1989), e.g.

```
  AllServers.Hr22.!exception_host::
actions
```

where `AllServers` is an abstract group, and `exception_host` is a host which is to be excluded from the rest. Classes thus form any number of overlapping sets, which cover the coordinate space of the distributed system (h, c, t), for different hosts h, with software components c, over time t. Classes sometimes become active in response to situations which conflict with policy. Class predication allows policy to encompass many-to-one maps and one-to-many maps. Many hosts can belong to the same class, and therefore policy actions can be common to many hosts. Similarly, each host can be characterized by many different classes or attributes which label its intended state, recognizing the multiple functions of each node in the virtual community, and the distributed nature of software systems.

Additional classes are automatically evaluated based on the state of the host, in relation to earlier times. This is accomplished by the additional *cfenvd* daemon, which learns and continually updates a database of system averages, which characterize "normal" behaviour.

5 STATES

The inherent unknowability of the host environment means that cfengine does not operate with any single notion of state; it has effectively several template definitions. Administrators do not use the same mental model to describe network arrival processes as they do the permissions of files, even though the essential nature of maintenance is the same.

A state is defined by policy. The specification of a policy rule is like the specification of a coordinate system (a scale of measurement) that is used to examine the compiance of the system. The full policy is a patchwork of such rules, some of which overlap. A cfengine state does not appear as a digital string, but rather as a set 'language' classes(Lewis and Papadimitriou, 1997), often represented in the form of a number of regular expressions, that place bounds on

- Characterizations of the configuration of operating system objects (cfagent digital comparisons of fuzzy sets).
- Numerical counts of environmental observations (cfenvd counts or values with real-valued averages).
- The frequency of execution of closed actions (cfagent locking).

Figure 3 illustrates schematically how a state can be a fuzzy region consisting of values that are 'as good as one another' This is called internal stability in the language of graph transitions (Burgess, 2004a). If the value of an object strays outside of the set of internally

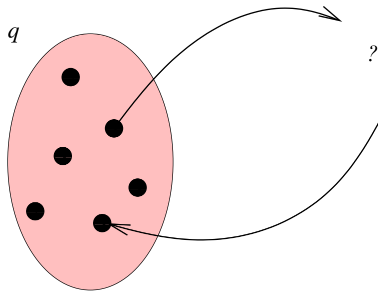

Figure 3: A cfengine state has both value and address. The address is a potentially fuzzy region, i.e. it is a constrained region rather than a necessarily unique point.

stable values, it is deemed to be a different state. A cfengine operator is then required to restore the value of the object to any one of the values in the stable set. For example, given the cfengine rule:

```
files:

 /etc/passwd mode=a+r,o-w owner=root
```

we see that this is not a unique specification of file permissions, but a set of acceptable 'equally good' values. The mode specification says that the acceptable state must be readable by all and must not be writable by any other than the owner of the file or its group owner. There is thus an implicit wildcard here.

6 MAINTENANCE MODEL

The maintenance model, underpinning cfengine, was outlined in ref. (Burgess, 2000b) and is fully described in ref. (Burgess, 2003). Although a precise description of the cfengine viewpoint is involved, the idea is rather simple. Each computer system that is to be managed by cfengine is treated as an autonomous system embedded in an environment (see fig. 4). It is important here to divide the system into two parts: host and environment because these are fundamentally different representations of processes that occur in the human-computer system.

The environment of a computer is a stochastic system. The behaviour of a host, on the other hand, is governed by a relatively simple computer program, with easily digitizable content. The level of complexity or information (Cover and Thomas, 1991) in the computer is much less than that of the environment. This distinction between the impulses that a computer receives from its environment and those that are programmed into it means that we must form a hybrid model for describing the changes that occur in a computer system.

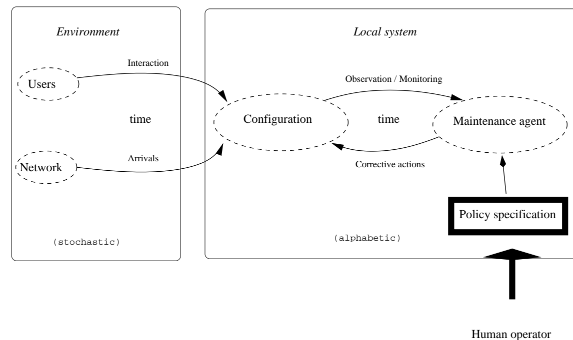

Figure 4: A schematic view of the management model. We define host and environment. The environment leads to randomness in the configuration of a host as it 'propagates' into the future. The maintenance agent responds in a finite time to repair the changes that are contrary to policy.

In reference (Burgess, 2003), a computer policy is described as an average specification of what we would like a human-computer system to do, from the viewpoint of the computer; it allows for short-term errors. Implicit in this definition is the projection of a complex human-computer system onto a simplistic digital computer system. We cannot therefore treat the computer system as a rule-based deterministic automaton, because we cannot predict the input from users and network. We have two options: we can isolate the computer from all random input (by making it a closed batch device), or we are forced to deal with the problem of stochastic change within the digital system. Since system administration is about human-computer interaction, we must take the latter view.

7 POLICY

The view of policy taken in ref. (Burgess, 2003) is that of a series of instructions, coded into the computer itself, that summarizes the *expected* behaviour. The precise behaviour is not enforcable, so there is no sense in trying to specify it at each computational timestep.

Cfengine does not attempt to provide a complete description of system policy. It deals with a specific problem: how to configure the alphabetic properties of the system of a single host, given that we cannot fully predict what changes are taking place. This, in turn, can affect the human aspects of policy through access control and other behavioural constraints.

Policy is a really description about what we consider to be normal. A description of normality is a decision about how we define errors. If we cannot equate normality with policy then we have not even a partially predictable system to manage and the con-

cept of ‘management’ would be meaningless.

This is where the split between system and environment has a fundamental conceptual bearing on our description of it. There are two kinds of normality that pertain to:

- Properties that we feel confident in deciding for ourselves (permissions of files, processes etc). These are decided and enforced. Deviations from these ‘digital’ specifications can be repaired or warned about directly by Shannon-like error correction.
- Properties that are controlled by the environment and must be learned (number of users logged in, the level of web requests). These have fluctuating values but might develop stable averages over time. These cannot normally be ‘corrected’ but they can be regulated over time (again this agrees with the maintenance theorem’s view of average specification over time).

Cfengine deals with these two different realms differently: the former by direct language specification and the latter by machine learning and by classifying (digitizing) the arrival process.

Learning an environmentally controlled state requires extra processing. First, the environment daemon cfenvd collects data and learns the normal state of the system using machine learning methods, then the state of the system is measured relative to the learned average state.

8 CONVERGENT REGULATION

The Shannon communication model of the noisy channel has been used to provide a simple picture of the maintenance process (Burgess, 2002a). Maintenance is the implementation of corrective actions, i.e. the analgoue of error correction in the Shannon picture. Maintenance appears more complex than Shannon error correction, however. What makes the analogy valid is that Shannon’s conclusions are independent of a theory of observation and measurement. For alphabetic strings, the task of observation and correction is trivial.

To view policy as digital, one uses the computer science idea of a language (Lewis and Papadimitriou, 1997). One creates a one-to-one mapping between the basic operations of cfengine and a discrete symbol alphabet. e.g.

```
A -> ``file mode=0644''
B -> ``file mode=0645''
C -> ``process email running''
```

Since policy is finite, in practice, this is denumerable. The agent interprets and translates the policy symbols into actions through operations, also in one to one correspondence.

1. Cfagent observes : X
2. Policy says :$X \rightarrow A$
3. Agent says :$A \rightarrow \hat{O}_{\rm file}(\texttt{passwd}, 0644, \texttt{root})$

Although the space of all possible policies is potentially very very large (though never truly infininte due to finite memory etc), only a small fraction of the possibilities is ever realized on a real system and this problem is not a limitation.

There is a larger point here: this is indeed what we mean by management. If we cannot reduce policy to a finite number of assertions and tasks then we have effectively lost control of the system and the idea of management becomes meaningless. We can define manageabilty as the ratio (Burgess, 2004a)

$$\mathcal{M} = \frac{\text{Information in environment}}{\text{Information in policy}}$$

It can therefore be assumed that the number of realistic policies and desirable operations is finite and that these can be defined and enumerated into a set of alphabetic operations.

These presumed and observed states of the system feed into the definition of the policy (Burgess, 2004b; Couch and Sun, 1994). However, if one defines the operations into classes $\hat{O}_1, \hat{O}_2, ...$ etc, then these form a strict alphabet of ‘black box’ objects. The fuzziness in the operations can be eliminated by introducing a new symbol for each denumeration of the possible parameters. Here is an example operation, simplified for the purpose of illustration.

```
files:

 /etc/passwd mode=0644 owner=root
```

This is actually a special instance of

```
files:

<filename> mode=<permissions>
           owner=<username>
```

which tells the agent to assert these properties of the named file object. Since there is a finite number of files and permissions and users, there is no impediment to listing every possible permutation of these and assigning a different alphabetic symbol to them. In operator language, the above action might be written:

$$\hat{O}_{\rm file}(\text{name}, \text{mode}, \text{owner}) \quad (1)$$

Let us suppose that example above evaluates to the alphabetic symbol ‘A’. When the agent observes these properties of the named object it comes up with a symbol value based upon what it has measured. Suppose now that a user of the system (who is formally

part of the environment) accidentally changes the permissions of the password file from mode=0644 to mode=0600. Moreover, we can suppose that this new value evaluates to the alphabetic character 'X'.

The transmission medium in this process is time itself. We regard the system (as is normal in the physical sciences) as being propagated from its current location to exactly the same place, over time. In other words, the time development of the system is just the transmission of the system into the future over no distance.

The primitive nature of the basic cfengine operations makes this set of operations fall into the following categories:

- Orthogonal (non-overlapping) operations that are strictly independent alphabetic atoms.
- User defined operations, outside of cfengine's framework, that cannot be guaranteed to be independent and might therefore 'overlap' with other alphabetic symbols.
- Operations entirely outside of cfengine's jurisdiction.

Cfengine introduced the notion of 'convergence' into system administration. This was orginially only implicit in the early work, but was named explicitly in the Computer Immunology essay in (Burgess, 1998) and was immediately taken up by Couch et al (Couch and Gilfix, 1999) and formed the basis of the configuration management workshops. This concept was quickly understood to be important.

A key part of avoiding uncontrolled behaviour are cfengine's transaction locks (Burgess and Skipitaris, 1997). These were designed to ensure three things:

- Consistency of the outcome of atomic operations, i.e. avoid contention due to concurrent execution of multiple agents.
- To limit the frequency with which operations could be repeated.
- To ensure that operations would not be able to hang indefinitely.

Behind these, is the assumption that new cfengine agents will be spawned frequently to check for maintenance operations.

One would like to secure the property that changes made to a configuration move towards a definite state, terminate after a small number of iterations, and that the route taken back towards the ideal state is unique and unidirectional. If this were not the case, then contradictions and non-terminating cycles would result. We require there to be absorbing states, or for operations to behave like semi-groups (Couch and Sun, 2003; Burgess, 2004a).

Cfengine uses the idea of *convergence* to an ideal state. This means that, no matter how many times cfengine is run, its state will only get closer to the ideal configuration. This is a stronger condition than *idempotence* as in Couch's interpretation (Couch and Sun, 2003; Couch and Sun, 1994). Since idempotence requires only $\hat{O}^2 = \hat{O}$, while convergence is relative to a specific policy state q_0 (Burgess, 2004a):

$$\begin{aligned} \hat{O}q &= q_0 \\ \hat{O}q_0 &= q_0. \end{aligned} \quad (2)$$

The point of convergence over multiple runs is that multiple orthogonal, convergent operations will always lead to the correct configuration, no matter which part of the configuration is incorrect, or in what order things occur. Complex operations might not complete within a single scheduled iteration, if external factors intervene in an untimely manner; but they will always converge eventually. This is proven in ref. (Burgess, 2004b).

Cfengine addresses convergence in two ways: by making each successful operation convergent in a single step, and by checking for contrary 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 repeated later; this is sufficient to ensure that simple configurations converge. We now consider how this works.

If two operations are *orthogonal*, it means that they can be applied independently of order, without affecting the final state of the system. Using a linear representation of vectors and matrix valued operators, this is equivalent to requiring their commutativity. The construction of a consistent policy compliant configuration has been subject to intense debate since it was understood that cfengine actions are not generally order dependent (Burgess, 2004b; Traugott, 2002; Couch and Sun, 1994).

The identification of convergence with order-free configuration led S. Traugott to formulate an alternative philosophy which he referred to as 'congruence' (Traugott, 2002). The concept was the opposite of the cfengine view and suggests that extreme ordering is the answer to reliability. Instead of always being able to find a path to the correct configuration from a given configuration, Traugott proposes simply destroying a faulty machine and building it up from scratch, like the differentiation of a biological stem cell. The argument about the necessity of ordering has been successfully refuted in (Burgess, 2004b; Couch and Sun, 1994), with certain qualifications. The two approaches can both be made to work, but only the convergent approach can be used for realtime maintenance.

A little-discussed but relevant part of the ordering problem is the matter of cfengine's adaptive transaction locking (Burgess and Skipitaris, 1997). The transaction locks allow cfengine processes to 'flow

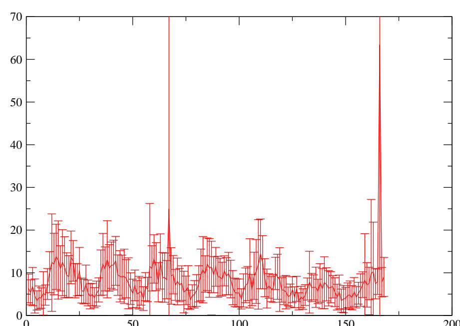

Figure 5: Weekly average (with standard deviation error bars) of incoming web request events as learned by cfenvd. Apart from some outliers we see a periodic pattern with peaks for Monday, Tuesday, etc, i.e. for each day of the week.

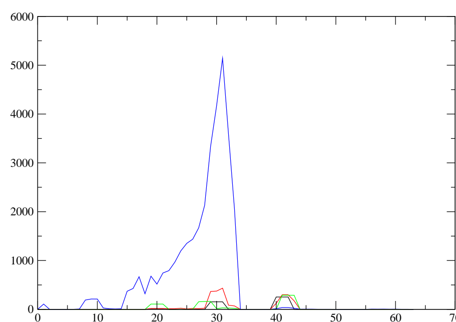

Figure 6: Distribution of outgoing network traffic about mean. This shows that there is a sharp expected value but that there is sometimes less tha this amount. The shape of this distribution is used to define a relativistic policy for anomalous behaviour.

through' one another and avoid going into infinite regression and also prevents agents from repeating themselves too often, or getting stuck on a problem. If an agent gets stuck, another one will destroy it and take over.

Game theory has been suggested as a way of optimizing decisions about system policy (Burgess, 2003; Burgess, 2004a). Cfengine can be thought of as a gaming agent that plays in a game against system 'gremlins', always trying to do their worst. Sometimes these gremlins are motivated users, at other times they might be the forces of chance.

9 ANOMALIES

In cfengine, an extra daemon (cfenvd) is used to collect statistical data about the recent history of each host (approximately the past two months), and classify it in a way that can be utilized by the cfengine agent. The agent learns. Data are gradually aged so that older values become less important (Burgess, 2002b) (see fig. 5).

The current data include the number of users, numbers of processes, etc.; cfenvd is also able to learn a number of network traffic anomalies. The daemon automatically adapts to the changing conditions, but has a built-in inertia which prevents anomalous signals from being given too much credence. Persistent changes will gradually change the 'normal state' of the host over an interval of a few weeks. Unlike some systems, cfengine's training period never ends.

Cfenvd sets classes in cfagent which describe the current state of the host in relation to its recent history (see fig. 6). The classes describe whether a parameter is above or below its average value, and how far from the average the current value is, in units of the standard-deviation (see above). This information could be utilized to arrange for particularly resource-intensive maintenance to be delayed until the expected activity was low. For instance: disk backups.

One way of understanding cfengine's behaviour in relation to environment and policy is as a Markov fluctuation model of change (Grimmett and Stirzaker, 2001), seeking an equilibrium configuration. In the original model, only first order Markov processes were available; everything about the system had to be deduced from the environment upon each new invocation. The only exception was the use of adaptive locks, which were used to regulate repetition. That approach has succeeded in solving many problems, but it lacked the ability to track long-term changes to the system, such as seasonal variations and changing patterns of usage.

The immunity analogy is also useful here. Immunological memory is like a stack of previously combatted problems, which works like an ordered list of changing priorities. This enables repeatedly actual problems to be dealt with more quickly than otherwise. It represents a change in biological policy toward threat. The same problem arises in configuration management, in the face of unpredictable faults or attacks.

The challenge of future anomaly detection is the find a stochastic anomaly language for a reactive agent policy. For statistical characteristics one has the shape of the distribution about the mean, the mean itself and the scales represented by the moments of the distribution. Recently the entropy of the distribution of originating IP addresses has been used to predicate anomalies (Burgess, 2002b):

10 Summary

This summary of cfengine outlines how the general theory of system maintenance is applied and implemented in the software. Further information about the detailed software implementation may be found in the extensive manuals at (Burgess, 1993). The key points to understanding cfengine management are

- The stochastic nature of the environment and hence we must expect randomness in hosts.
- The discrete nature of host configuration which must be constrained.
- Classification of environmental changes into a patchwork of sets.
- Policy constraints on hosts.
- Operator responses to policy deviations (anomalies and countermeasures).
- Convergence and idempotence of configuration countermeasures.

Cfengine is an on-going project and, at present, it is being re-written to more directly reflect the theoretical model that has emerged in tandem with its development, and extended to allow more general software control of robotic systems.

REFERENCES

Burgess, M. (1993). Cfengine www site. *http://www.iu.hio.no/cfengine*.

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

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

Burgess, M. (2000b). Theoretical system administration. *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV) (USENIX Association: Berkeley, CA)*, page 1.

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

Burgess, M. (2002b). 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.

Burgess, M. (2003). On the theory of system administration. *Science of Computer Programming*, 49:1.

Burgess, M. (2004a). *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester.

Burgess, M. (2004b). Cfengine's immunity model of evolving configuration management. *Science of Computer Programming*, 51:197.

Burgess, M., Haugerud, H., Reitan, T., and Straumsnes, S. (2001). Measuring host normality. *ACM Transactions on Computing Systems*, 20:125–160.

Burgess, M. and Skipitaris, D. (1997). Adaptive locks for frequently scheduled tasks with unpredictable runtimes. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 113.

Comer, D. and Peterson, L. (1989). Understanding naming in distributed systems. *Distributed Computing*, 3:51.

Couch, A. and Gilfix, M. (1999). 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.

Couch, A. and Sun, Y. (1994). On observed reproducibility in network configuration management. *Science of Computer Programming*, (to appear).

Couch, A. and Sun, Y. (2003). On the algebraic structure of convergence. *Submitted to DSOM 2003*.

Cover, T. and Thomas, J. (1991). *Elements of Information Theory*. (J.Wiley & Sons., New York).

Ferber, J. (1999). *Multi-agent Systems: Introduction to Distributed Artificial Intelligence*. Addison Wesley.

Grimmett, G. and Stirzaker, D. (2001). *Probability and random processes (3rd edition)*. Oxford scientific publications, Oxford.

Lewis, H. and Papadimitriou, C. (1997). *Elements of the Theory of Computation, Second edition*. Prentice Hall, New York.

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

Shannon, C. and Weaver, W. (1949). *The mathematical theory of communication*. University of Illinois Press, Urbana.

Sloman, M. and Moffet, J. (1993). Policy hierarchies for distributed systems management. *Journal of Network and System Management*, 11(9):1404.

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

Traugott, S. (2002). Why order matters: Turing equivalence in automated systems administration. *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI) (USENIX Association: Berkeley, CA)*, page 99.