# Autonomic Computing Approximated by Fixed-Point Promises

Mark Burgess and Alva Couch

Oslo University College and Tufts University

**Abstract.** We use the concept of promises to develop a service oriented abstraction of the primitive operations that make an autonomic computer system. Convergent behaviour does not depend on centralized control. We summarize necessary and sufficient conditions for maintaining a convergently enforced policy without sacrificing autonomy of decision, and we discuss whether the idea of versioning control or "rollback" is compatible with an autonomic framework.

## 1 Introduction

A central problem in system administration is to devise a secure and scalable approach to managing the configuration and runtime behaviour of a computer system. Autonomic computing seeks to achieve this by reducing the role of human intervention in the management strategy: humans are both expensive and unreliable. It is generally accepted that this is a cost-effective and rational strategy, however, progress towards this goal has been more in the realm of ideas than technology. Making computers truly autonomous is not really the goal of autonomic computing, but making them independent is; one would like to retain individual decision-making authority and eliminate manual labour by humans, without sacrificing the freedom of individual ownership.

Let us note at the outset that many authors have used the word "autonomic" to denote a specific kind of automation, involving log analysis and making changes based upon that analysis in a feedback loop. We utilize a broader definition, in which "autonomic control" may also be achieved by event-condition-action (ECA) tuples, or even by operations that actively probe their environment to discover problems and effect corrections. For this reason, some authors would refer to our work as "self-managing" rather than "autonomic". In biological systems (from which the terms "autonomic" and "homeostatic" arose) there are many examples of regulatory systems based upon implicit rather than explicit feedback, or even upon competition between regulatory agents (e.g., in the microbiology of the cell). Thus we utilize the term "autonomic" in a broader context, to mean any kind of regulatory mechanism, not just those based upon log analysis and explicit feedback loops.

### 1.1 On the importance of policy

An important approach to reducing human involvement *in situ*, is to equip computers with automated repair and response mechanisms and to frame certain

pre-cached decisions as a predetermined "policy" that constrains the behaviour of a system within acceptable bounds. This has a number of advantages over traditional centralized management consoles: each device can be equipped with its own policy, enforcement is entirely local, and the basic decisions can be made either by an individual for a single device, or by an authority for the governance of many devices. This kind of flexibility is essential for ubiquitous computing scenarios, where devices are increasingly personal consumer items and users will not accept centralized management.

Thus autonomic behaviour or self-governance, in this form, comprises two main themes:

- Managing *a priori* internal data, with attendant runtime consequences (configuration management).
- Managing *a postiori* environmental influences, which affect runtime behaviour (anomaly discovery and resolution.).

### 1.2 On the importance of autonomy

The traditional view of configuration management has been to think in terms of a control paradigm, with totalitarian decisions made from a centralized location or controller. In this view, hosts are commanded to comply with some template or system blueprint and are possibly monitored for non-compliance from that same location. For this to work, we require an implicit subordination of hosts, i.e. they must follow these commands, which in turn implies a command or authority structure that is external to the system.

Today we are confronted by a free market economic organizational paradigm in our systems, e.g. *ad hoc* networks, the Internet, Service Oriented Architectural computing, and hand-held devices. Here services from a server are requested and sometimes paid for by a client, outsourcing of tasks is common and many other issues that all add up to one thing: with a modern viewpoint, systems are highly distributed, and there are possibly many 'masters' whose decisions must be obeyed. The idea of a single authority who decides for all entities is simply incorrect. Thus, to address the concept of human disinvolvement in a realistic way, we must be somewhat clear about our intentions.

Several models and terms have been used over the years to describe a vision of automated computer management. In most cases, they are based on a framework of external control and decision. Let us mention two terms that are relevant to reducing human involvement.

- Autonomic, meaning self-governing (from Greek).
- Immunity, meaning insusceptible, or self-preserving.

We understand *autonomic* to mean a system that is capable of carrying out its pre-decided tasks without a human nurse-maid (i.e. not a robot revolution based on amok machine intelligence) but ideally also a system that can learn and adapt without external decision and command if it is self-governing (this distinguishes

our definition of "autonomic" from the one referenced in IBM's autonomic computing initiative). An autonomic system might collaborate with other autonomic systems voluntarily[1, 2], but neither would be in charge of the other. We understand *immunity* to mean simply a resilience to deleterious environmental effects. It is not entirely clear whether autonomic is a stronger appellation than immunity (implying some level of autonomous decision-making); in either case, this is surely a matter for policy to decide and hence policy will play a central role in discussing autonomic computing.

### 1.3 The plan

In this paper, we describe the approach to autonomic computing developed for and around the popular software Cfengine[3, 4]. Cfengine is a pragmatic approximation to a number of theoretical ideas about configuration management and machine learning that includes policy-guided decision-making and robust immunity to random change. Its freedom for adaptability has the character of a game-tree, with fixed rules, rather than a genetic mutation – and this is compatible with the idea of limited autonomy. Most importantly, the model takes the strong view that no autonomous component can be commanded by any outside entity, hence the limited autonomy is secured.

We merge Cfengine's so-called immunity model of configuration management together with a theory of autonomous decision-making, using the recently developed notion of *promises*[5] and describe why we believe that this approach is both well-suited to today's economic landscape and inflicts a minimum human cost.

The plan for the paper is as follows: we review the concepts of convergent operations and promises from earlier work, and show how both the autonomous architecture and the detailed operations can be expressed in terms of service-like promises. We provide some concrete examples for implementing basic operational changes within a system. Finally, we discuss the notion of change management in an autonomic setting.

## 2 Policy governed architectures

The notion of policy-based configuration management has been stressed since the early 1990's[6–8, 3] as a way of separating high level decisions from low level details. Policy allows for the pre-decision of certain issues associated with system behaviour. Several approaches to policy based management have been suggested[7, 9, 10]. The most popular notion is that of Event Condition Action (ECA), in which an event monitor receives some kind of alarm signal, which it evaluates according to the conditional rules it has available, and finally associates this with a single action to be carried out. Such an event-driven architecture assumes that all changes to the system will be signalled by leading events. We disagree with this view.

The immunity model[4] (developed originally for the agent system cfengine) shares several features with ECA as well as with the homoeostatic security model proposed in refs. [11, 12]. The name is drawn from the analogy to artificial immune systems as regulators of a policy compliant or healthy state. However, it differs from ECA somewhat in both philosophy and implementation. In particular, the immunity model does not expect all problems to be noticed or alerted as distinguishable events, like a fire-fighting approach. Instead it proposes a continual schedule for maintenance sweeps of the system – if anything is discovered that does not fulfill the parameters of the policy description, it will be repaired. The important difference is that problem discovery and repair are more tightly linked in the immunity model: rather than having separate rules for monitoring and actions, every possible action is equipped with its own autonomic observational capabilities. Rather than waiting for an event, we simply repeat the actions *ad infinitum*; thus the immunity model constantly looks for problems rather than waiting for events to arrive from a third party.

It is this immunity model that we shall use in the present paper. We believe that this is the correct model for a self-regulating system, since it was shown in ref. [13] that a complete specification of policy determines a system's properties only to within an intrinsic uncertainty. The uncertainty can only be reduced by making each operation a closure[14, 15] by constant polling or maintenance sweeps, as advocated by the immunity model.

### 2.1 From ECA to Immunity

We assert then that a reactive system does not exist without a policy. Policy compliance can be enforced, maintained and regulated by mapping each requirement onto a number of operations that are autonomically self-regulating[3, 13, 16, 4].

We shall draw on some of the results and concepts from earlier work to piece together a description of autonomic computing based on constellations of promises. To make this journey, we begin with some definitions.

**Definition 1** *An* action *is an operation executed by an agent. Actions are directed operations that point the system toward a new state, from a starting state. An action is carried out by a combination of primitive 'transition operators'* $\{T_a\}$*, where the Latin index a runs over the set of independent primitives (e.g. copy, set attribute1, 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. Thus one would not wish to let them run autonomously, without having a human examine their behaviour We would like to replace these with a new set C of constrained, convergent operators that lead to more predictable behaviour, suitable for autonomous operation.

Hidden in this language is the essence of 'event condition action', but with an important distinction: events are entirely self-determined, or acknowledged

at the convenience of the repair operation. Let us consider the structure of the mapping between conditional rules and actions.

An important initial concept is that of a *class* $\chi$ of hosts, i.e., a set of hosts with particular properties. We need this concept in order to "decide" whether action is necessary on a specific host. Let $\mathcal{P}$ represent the set of all identifying host properties (like fingerprints or observed characteristics) with members $p$, and $\mathcal{E}$ represents a domain of entities (with members $e$) to be configured, then the class $\chi$ may be thought of as a function:

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

Thus a class property $\chi(e, p)$ is true if entity $e$ 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 $\mathcal{O}$ of actions to be performed. Suppose there are $m$ such actions; then we may write:

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

where each $\hat{O}$ represents a specific action to perform. Thus the notion of a rule-set implicitly binds pairs of devices and their configuration requirements $(h, \hat{O})$ to certain environmental conditions, expressed by the classes. All that remains is to imbue the operations with sufficiently autonomous behaviour.

### 2.2 Resilience to unpredictability: the importance of sinks

In spite of heady traditions of computer science and the predominance of deterministic transactional models, systems that interact with an environment of humans and clients are not predictable. They must be designed to tolerate undesirable states (illness) that occur with finite probability, and employ autonomous countermeasures[17].

Since the actual state of a system is a stochastic variable, a policy-compliant state is at best a preferred end-goal, not a reliable situation. So how can we maximize the probability that a system will be in its policy-conformant state? Is there a way that we can make the state stable to perturbations, like a ball rolling back to the base of a valley, or unwanted rain-water running down a drain? Two methods come to mind:

- Catch and remove any raindrops that arrive (Event Condition Action queue).
- Equip all streets with drains and gutters (Fixed Point resilient design).

"Water-free streets" is a fixed-point of the latter draining operation, since once the rain or snow is gone, it does not come back by virtue of having drains (and perhaps under-street heating, as in many winter cities).

The lesson of sinks is that they automatically enforce a fixed attractor state: no rain or no snow, or policy conformant streets is a clear an unambiguous goal. The details of how we arrive at the goal are less important than the state itself. Thus, a smart building designer would build the property of immunity to rain and snow into a building, rather than wait for an external government (e.g. King Canute) to order the water away.

Self-governance is thus intimately related to the concept of immunity to undesirable change, as the latter *always* requires a mechanism of active reparative counter-measures. There is no shield that will protect us from undesirable states. To solve a variety of policy issues then (clear streets, lighting at night, garbage removal), what we are looking for is basic atoms of governance that have reparatory actions designed into them:

- Operations that do not interfere or counteract one another's results.
- Operations with a clearly defined, stable policy-compliant end-result (a fixed point).

Note that we say operations here rather than operators. It is possible to have operators that overlap in function (create and destroy for instance), but that should not be used together in the same time and place. Promise theory helps here (see below) in defining such an occurrence as a broken promise. In general, we need operators that counteract one another in order to accomplish changes in intent over time.

### 2.3 Orthogonality and Fixed Point Convergence

These two properties are easily achieved[3, 17, 4]. In geometry and linear algebra the notion of an orthogonal basis that spans a vector space renders many discussions not only possible but lucid (if not Euclid!). Configuration entities do not generally form a vector space, but they are often organized with a Cartesian product structure (like a struct or record, or a database table) and therefore there is a natural decomposition of parameters that can change independently of one another[13, 18]. We would like to adopt such a set of orthogonal change operators, since this allows us to decompose any change into elemental, atomic sub-changes.

**Definition 2 (Orthogonality)** *Two operations are orthogonal if they refer to transitions in orthogonal parameters. Two parameters are orthogonal if a change can be made in one parameter without a change occurring in the other.*

Consider now, how configurations are built up from such atomic operations.

Let the state $|\mathbf{0}\rangle$ denote some base-state configuration of a set of entities (e.g. the state in which all bit-values are set to zero). From this state, one may build up an arbitrary new state $|a, b, c, \ldots\rangle$ through the action of sequences of the transition operators:

$$\begin{aligned} |a, b, c\rangle &= (I + aT_1^+ + bT_2^+ + cT_3^+)|\mathbf{0}\rangle \\ &= \hat{O}(a, b, c)|\mathbf{0}\rangle. \end{aligned} \tag{4}$$

The set $\{a, b, c\}$ is a representation of the rule set $r \in \mathcal{R}$ in the operator basis. One could regarded this as the system policy specification relative to the operator set (just as DNA is a policy code relative to a given set of proteins, imperfect as not all proteins are orthogonal). However, the need to specify an configuration relative to a specific starting state is inconvenient. We would prefer to have a set of operations that simply leads to a satisfactory *final state*, in the manner of an attractor[3, 17]. This can be achieved if we are able to construct a set of operators $C_a$ that possess fixed-points $v^*$ at policy compliant values, i.e.

$$C(v^*) = v^* \tag{5}$$

for some function $C$, $n \geq 1$, and fixed-point value $v^* \subset v$ of the parameter values. Moreover, we would like some number of operations $C^n$ to place us at the fixed point, so that $C$ is a completely absorbing map. Hence, 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[19, 18]). Using this representation, one can now define the meaning of fixed-point convergence.

**Definition 3** *Let $|R\rangle$ be an arbitrary state of a system resource. An operator $\hat{C}_a$ is said to be convergent if it has the property*

$$\begin{aligned} (\hat{C}_a)^n |\mathbf{R}\rangle &= |\mathbf{0}\rangle \\ \hat{C}_a |\mathbf{0}\rangle &= |\mathbf{0}\rangle, \end{aligned} \tag{6}$$

*for some integer $n > 1$, i.e. the $n$-th power of the operation is null-potent on the absorbing state.*

(In general, Cfengine is not able to achieve this property exactly, since it relies on other system functions that do not respect the idea fixed point goals; in such cases idempotence can be ensured, i.e. $C^n(\cdot) \equiv C(\cdot)$.)

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 $|v\rangle$ being operated upon, with representation:*

$$\hat{C}(|v\rangle) = I + \sum_a \theta_a \left( \left| |v\rangle \right| \right) \lambda_a T_a^+. \tag{7}$$

*The precise representation of the operators depends on the coding level of the system (see discussion in ref. [13]) and the choice of a basis at that level. However, once this is chosen, the $T_a$ are simply generators of the translation group.*

These operators contain a knowledge of the base state by their implicit dependence on the current state via the Heaviside step function[4]. This is required

for convergence, but it also allows the specifier of policy to code rules (operators $\hat{C}$ and $\hat{C}'$) that have contradictory notions of what $|\mathbf{0}\rangle$ is. This can lead to strings of operations that "do" and "undo" the state of the system.

**Definition 4** *Two convergent operators $\hat{C}_1$ and $\hat{C}_2$ are non-contradictory if there is a ground state $|\mathbf{0}\rangle$ such that*

$$\begin{aligned} \hat{C}_1|\mathbf{0}\rangle &= |\mathbf{0}\rangle \\ \hat{C}_2|\mathbf{0}\rangle &= |\mathbf{0}\rangle, \end{aligned} \tag{8}$$

*i.e. if they terminate on the same state. Similarly, a set of convergent operations is non-contradictory is it terminates on the same ground state.*

The aim in crafting useful and effective policies is thus to ensure that sets of operators, which do not satisfy this property, are not part of a policy specification. We shall not carry this summary further; readers are referred to refs. [4, 18] for more details of why these operators have fixed points.

Suppose then that we were to outsource the management of a set of autonomous entities. We would like to be certain to apply the policy maintenance rules with a sufficient regularity to combat the likely faults that can occur[13]. Alas, we cannot be so certain, we can only pledge to do our best.

## 3 Promise theory

A theory based entirely upon certainty of action is somewhat presumptuous in an environment over which one does not have guaranteed control. A theory of intentions is more reasonable, since an intended action can at best be translated one-to-one into the appropriate action, or at worst be analysed for inconsistencies or other problems in advance.

Since policy is about pre-deciding or modelling possible scenarios, we develop a predictive theory based on expected action as in Event Condition Action (ECA) models[5, 20, 2]. ECA can easily give the incorrect impression that the only reasons why a device might fail to comply with policy are:

- Inaction.
- Incorrect action.

A ream of other possibilities exists, including, lack of authority, random error, environmental interference, system fault, etc. The word *promise* was chosen rather than *intention* since the latter is both more vague and used in modal logic, with different connotations.

Promise theory is a representation of policy, from the viewpoint that agents make all of their own decisions. Each agent makes promises that inform other agents about its future behaviour. Since each agent makes its own private promises, and no agent can promise something about another (without violating autonomy), it is well suited to discussing autonomy.

There are several ways to motivate the insistence on strict autonomy of components. Let us mention two:

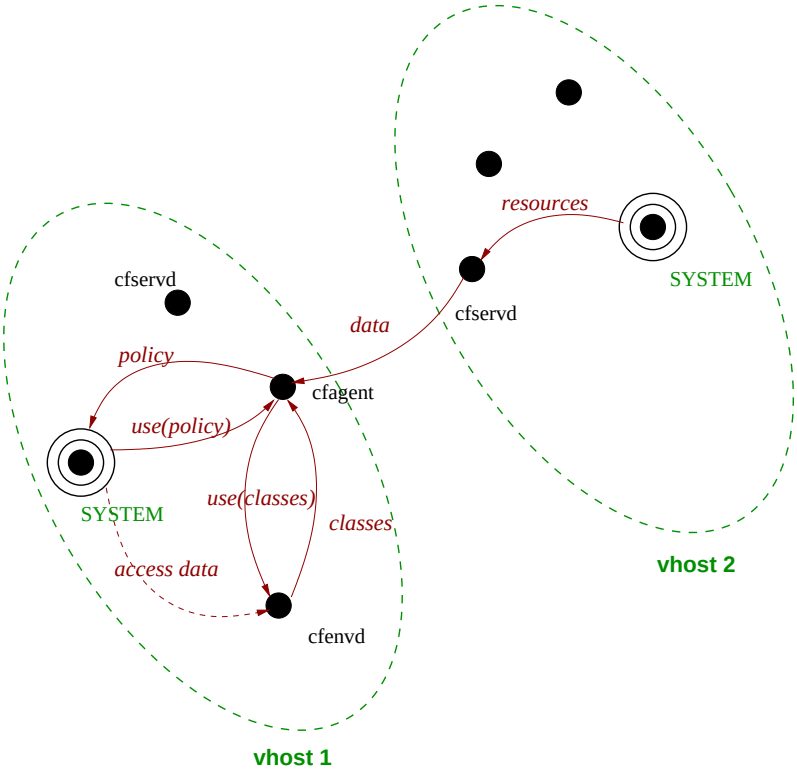


**Fig. 1.** The basic promise relationships within the cfengine architecture. Operators are built into cfagent. Policy is built into operators. Only system resources give up their autonomy by succumbing to operators.

– Reliability folk theorems[21, 22] show that reliability improves when reliability (redundancy) mechanisms are situated at the lowest possible level of a system. When every component in a system is autonomously self-repairing, the system as a whole is maximally reliable.
– The autonomous viewpoint is an extreme viewpoint that makes a minimum of assumptions. Additional assumptions have to be coded explicitly as promises, and hence nothing is concealed in such a model.

The strength of promise theory's autonomous approach lies, amongst other things, in its elemental approach to policy. It makes normally hidden assumptions explicit, allowing one to avoid conflicts of policy quite easily. Several approaches have been used in the past to resolve such conflicts[23–29] but promises allow one to understand these by simple algebraic rules.

In ref. [13], policy is identified as a specification of the average configuration of the system over time. An important aspect of this definition is that it allows for error tolerances, necessitated by randomly occurring (but not necessarily detected) events that corrupt policy in the system management loop. There is a

probabilistic or stochastic element to system behaviour, so policy can only be an average property of a system. We do not require a full definition of host policy here from ref. [13]. It suffices to define the following.

**Definition 5 (Policy I)** *The policy of an individual computing device (agent or component) is a representative specification of the desired average configuration of a host by appropriate constraints[13].*

Promise theory takes a service viewpoint of policy and uses a graphical language to compose system properties and analyse them. The promise theory methodology is to begin with detailed, *typed* promises, from which the algebra of cooperation can be understood, and then gradually to eliminate the types through transformations that elucidate large scale structure and reliability.

**Definition 6 (Policy II)** *The policy of an individual computing device (agent or component) is a collection of promises, i.e. specifications of its future state and behaviour.*

The form of a promise is[5, 20, 2]:

$$n_1 \xrightarrow{\pi} n_2 \tag{9}$$

meaning $n_1$ is the node making a promise to node $n_2$ and $\pi$ is the typed promise body, which describes the interpretation and limitation of the promise. A promise body is a typed constraint on behaviour, asserted from one agent to another. Note that a promise has the form of a service: it provides information to the recipient and the promised behaviour can be viewed as a service too. Hence there is a relationship to Service Oriented Architectures[30].

Promises fall into approximately four notable categories:

| Promise type | Notation | Interpretation |
|---|---|---|
| Basic | $n_1 \xrightarrow{\pi} n_2$ | Provide service/flow |
| Cooperative | $n_1 \xrightarrow{C(\pi)} n_2$ | Imitate/Follow |
| Use | $n_1 \xrightarrow{U(\pi)} n_2$ | Use/Accept from |
| Conditional | $n_1 \xrightarrow{\pi_1/\pi_2} n_2$ | Promise 'queue': $\pi_1$ if $\pi_2$ |

Conditional promises will be important. A conditional promise $a \xrightarrow{\pi/c} b$ *is not a promise* unless the condition $c$ can be determined to be true by the recipient of the promise. This might require additional promises to complete.

Basic service promises form any number of types, e.g. 'I will provide web service in less than 5 milliseconds' or 'I will answer you three questions only'. Of course, a promise at this level is far too vague to be useful unless it can be decomposed into a number of elemental specifics that will lead to the proposed result. This is where the operational decomposition from the immunity model can show us how to create a spanning set of basic promises.

Some other example promise templates include:

- $X \overset{q \leq q_0}{\rightarrow} Y$: agent $X$ promises to never exceed the limit $q \leq q_0$.
- $X \overset{q = q_0}{\rightarrow} X$: agent $X$ promises to satisfy $q = q_0$.
- $X \overset{\ell \subset L}{\Rightarrow} Y$: $X$ promises to keep the value $\ell$ in some sub-language of the language $L$.
- $X \overset{S}{\rightarrow} Y$: agent $X$ offers service $S$ to $Y$.
- $X \overset{R}{\rightarrow} Y$: agent $X$ promises to relay $R$ to $Y$.
- $X \overset{\neg R}{\rightarrow} Y$: agent $X$ promises to never relay $R$ to $Y$.
- $X \overset{S,t}{\rightarrow} Y$: agent $X$ promises to respond with service $S$ to $Y$ within $t$ seconds.

It is worth noting that promises are broadly analogous to Service Level Agreements offered in the establishment of contracts. Indeed, it would be both practical and informative to express service level agreements in terms of low level, concrete promises.

Cooperative promises are used in order to build concensus amongst individual agents with respect to a basic promise type. If $\pi$ is a promise type, then $C(\pi)$ is a corresponding promise type between agents to collaborate or imitate one another's behaviour with respect to $\pi$. A promise is said to be *broken* if an agent makes two different promises of the same type at the same time (this is different from a promise that has expired or been changed). Readers are directed towards refs. [5, 20, 2] for details.

## 4 A promise model of operations

The benefits of promise theory are two-fold. Autonomy of the nodes in a promise graph forces us to make all relationships and policy atoms explicit, and the service-nature of the promises makes all changes appear on an equal footing, no matter whether the resources are local or remote, whether the communication is over an internal bus or an external network. This makes certain descriptions that we are used to taking for granted seem cumbersome in promise theory, but it also means that we avoid problems like hidden inconsistencies. Finally, it allows us to take any component of a system and make it into an independently autonomic device.

The familiar understanding of an operator in mathematics is an entity that initiates change in a state (as in eqn. (6)). There is normally no discussion about whether or not an operator is authorized to make such a change – however, here we need to take such authorization into account in promise theory, since an agent is under no obligation to accept a change offered to it. It must first grant its authorization, by agreeing to use the change.

### 4.1 Promise of action

An operator is easily conceptualized in terms of promises, 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.

$$\hat{O}_{q_p} \equiv R \overset{\text{use } q_p}{\rightarrow} A \otimes A \overset{\text{please set } q_p}{\rightarrow} R$$
$$\equiv \begin{pmatrix} \overset{q_p}{\leftarrow} \\ R \overset{\text{use } q_p}{\rightarrow} A \\ \end{pmatrix}. \tag{10}$$

How can the resource promise something? If the resource is a file, how can it promise to adopt a state proposed to it by a configuration agent? This is a formality that we can easily address. We can, for instance, model the software configuration agent in two parts: 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. Alternatively, one could go through an authorization broker (another independent agent).

Consider first the method of delegating authority in fig. 2. In this figure, we illustrate how to understand how a resource submits to the authority of an external agent on a voluntary basis. Such submissive behaviour is built into some systems from the start.

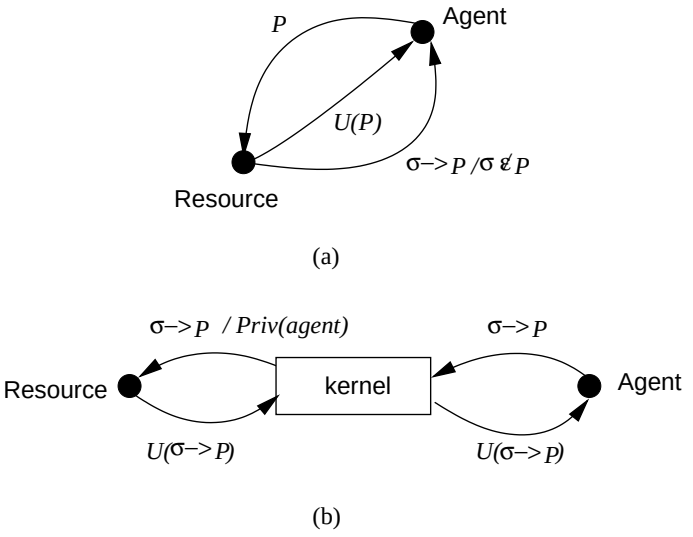


**Fig. 2.** Autonomous enforcement of policy (a) by direct interaction between policy agent and resource, and (b) through a middle-man or kernel that brokers privilege.

In the first case (a), an agent promises to inform the resource about the policy state $P$. The resource agrees to accept the agent's advice and *use* $P$. It further promises the agent that it will change its state $\sigma$ to the policy state if the current state is not in the range of acceptable values in $P$.

In (b), the whole process goes though a privilege broker that has a list of agents whom it agrees to trust. This is the basis of a traditional switched-mode operation kernel. The same model works for a sudo-like program.

In either case, the end resource must submit to the advice it receives about changes of state $\sigma$ with a use promise $U(\sigma \rightarrow P)$. i.e. the resource gives up its autonomy. In a traditional operating system this promise of access is hardwired into the system. In a distributed (e.g. web-based) system, permissions must generally be granted through some kind of access list.

### 4.2 Convergent semantics

There is a set of possible configurations for an object of a given type. We can denote the type as $\tau$, e.g. file attributes, web process configuration. The set of all possible behaviours of this type is denoted $\tau$. In making a policy, we promise to restrict the behaviour of our system to a subset of that possible realm $\tau$, which we denote $P$ (for policy constraint). Finally, if the system falls outside of the allowed constraint, we must identify a unique state $q \in P$ 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 $P$.

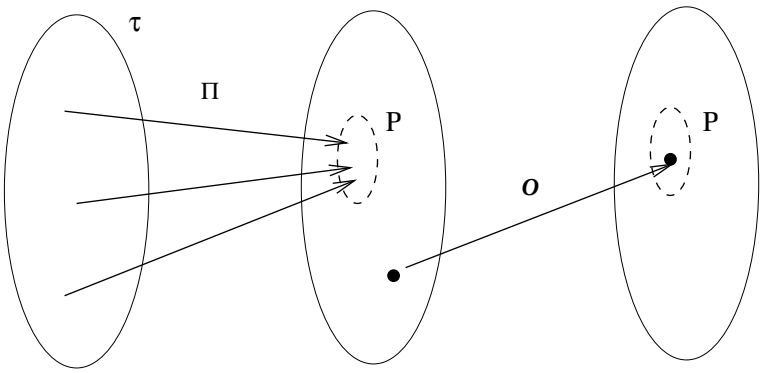


**Fig. 3.** The mappings in policy from the space of possibilities to the policy-allowed subset, to a representative element of the policy allowed set: $\tau \rightarrow P \rightarrow q \in P$

Consider the diagram in fig. 3. A promise is a description of what sub-domain of values $P$ a state of type $\tau$ may occupy, according to policy. The set $\tau$ represents a domain of possible values for the state. The sub-domain $P \subset \tau$ could be represented as a regular expression, or a list, etc.

We now want to associate configuration promises with configuration operations, and service promises with services (which in turn are configured by operations). For each type of promise we assume the existence of 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.

An operation $\hat{O}$ is applied to any state $q \in (\tau - P)$, i.e. any state outside of policy. An operation is an idempotent surjection from $\tau - P$ to $P$.

### 4.3 Enforcement

We might want to imagine that policy enforcement means that a configuration management system will promises to make everything on the system "right," but this is an improper way to define a promise for autonomous nodes, because autonomy implies that each resource controls its own attributes. A configuration agent can not promise something about the state of attributes that do not belong to it. Thus we start from a scenario that is more akin to the behavior of the Web than that of a Unix system administrator.

To reconcile this viewpoint we must turn things around and imagine a configuration agent to be an observer of everyone else's promises. Policy is not something that is centrally defined, it is the sum of many individual promises.

**Definition 7 (Configuration Agent as Promise Observer)** *A configuration agent functions as a observer of promises. This monitoring allows resources to (conceptually) make promises about their configuration to a monitor.*

But, we would also like to have the configuration agent 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 behaviour of the prototype configuration agent, 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 $P$.

We shall pick a particular $\tau$ and stay with it so that we may simplify the notation by eliminating $\tau$. The resource promises the agent that it will obey $P$ 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. 4).

$$\text{resource} \overset{q \in P}{\rightarrow} \text{agent} \tag{11}$$

and the agent promises to set the value of $q$ to a specific value in $P$. These promises are consistent but they do not capture the usual convergent semantics of the promises made by configuration agents (e.g. cfengine [31]). 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. 5), we add a conditional promise that gives the promises convergent semantics. Let $A$ be the agent and let $R$ be the resource:

$$R \overset{q \in P}{\rightarrow} A$$

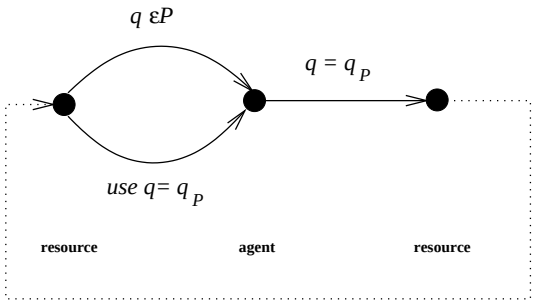


**Fig. 4.** The simplest agreement between a configuration agent and a resource.

$$\begin{aligned} R &\overset{U(q_p)}{\rightarrow} A \\ A &\overset{q_p}{\rightarrow} R \end{aligned} \tag{12}$$

In other words, the resource promises that it will comply with policy (a constraint that can allow several values). The agent promises to provide a policy compliant value $q_p$, and the resource promises to use it.

There is nothing wrong with this set of promises: the promises are self-consistent and converge to the value $q_p$. However, the general promise to keep $q \in P$ is ignored here. It is assumed that this representative value is part of the policy, i.e. $q_p \in P$, and yet the resource accepts the value $q_p$ without qualification, so one might as well have had:

$$q_p \in P \rightarrow (q := q_p). \tag{13}$$

In this expression, $q := q_p$ means that $q$ is assigned the value $q_p$ as a result of the promise.

Let us try again using conditional promises to enhance this convergence to 'fix only if necessary'. We denote the complement of $P$ by $\neg P$. Now suppose the agent promises:

$$\begin{aligned} R &\overset{q \in P}{\rightarrow} A \\ R &\overset{q=q_p / q \in \neg P}{\rightarrow} A \\ R &\overset{U(q_p)}{\rightarrow} A \\ A &\overset{q_p}{\rightarrow} R \end{aligned} \tag{14}$$

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 $q_p = B$.

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

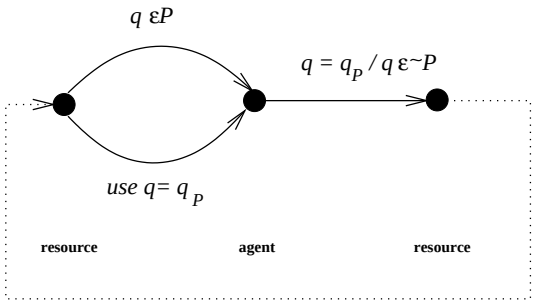


**Fig. 5.** An agreement between a configuration agent and a resource with a conditional promise.

a more general fuzzy range $q \in P$. No inconsistency or indeterminacy of action arises. This means that promises can account for acceptable uncertainty in a system. This is an important feature, as uncertainty is inevitable. Note also that this fuzziness allows one to model external forces, e.g., abusers and attackers, as if they were operators upon the system.

## 5 The relationship with game theory

Let us comment briefly on the relationship between the promise approach and economics. In many ways, the promise graph approach makes each promise relationship into a Principle-Agent scenario, i.e. a game in which one uses incentives to achieve voluntary cooperation rather than control by force.

The alternative promises an agent can make correspond to alternative rules like the strategies in a strategic game (this is proven in [2]). The aim is to always be able to collapse a game tree into a stable strategic form, else there will be a combinatoric explosion of a game, i.e. the game will never terminate or have an equilibrium. For stable functioning, we would like a system to be sitting in an economic equilibrium configuration. For more details, readers are referred to ref. [2].

## 6 Example operators

To implement our theory in a practical setting, we seek an orthogonal set of basic operations that can be promised. We shall choose a small set of configuration primitives for concreteness. Even these basic items are non-trivial once we require them to be fixed-points operators – meaning that correct configuration implies no action. Operators from Cfengine serve as a starting point for our search, but some do not have appropriate fixed-point properties and must be modified appropriately.

### 6.1 Copy from node

Basic data copying from a source is a basic management function, e.g. package installation, repository lookup.

A copy operation is a straightforward service interaction in which a policy-node informs a destination node of its recommended policy source server-node. The receiver node agrees to that policy by promising to use the information from the policy-node. The source server promises (necessarily) the contents of the data. Note that the receiver does not have to promise the server that it will use the data, since it owes the server no compliance (but it has already promised this to the policy monitor node). The server also promises to agree to the policy

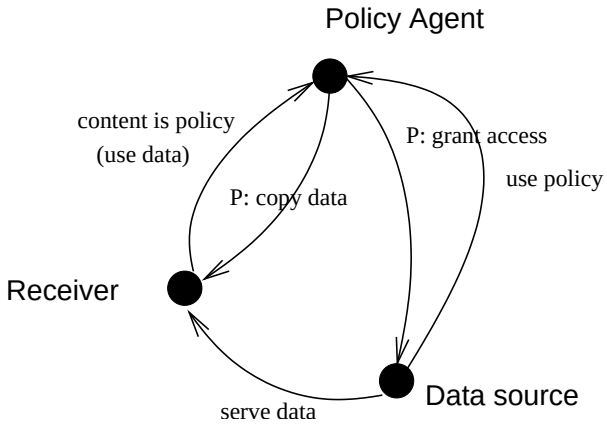


**Fig. 6.** A copy action, represented as promises. Promises labelled $P$ : are policy directives from the agent.

whose contents are promised by the policy-node. Note the role of the third party policy agent in delivering policy and observing its compliance.

Let the nodes be denoted $\langle S, R, A\rangle$, for source, receiver and agent. Let $P_S$ be $S$'s policy and $P_R$ be $R$'s policy.

$$\begin{array}{lll} A & \stackrel{P_S}{\rightarrow} & S \\ A & \stackrel{P_R}{\rightarrow} & R \\ S & \stackrel{U(P_S)}{\rightarrow} & A \\ R & \stackrel{U(P_R)}{\rightarrow} & A \\ P & \stackrel{U(\mathrm{data})}{\rightarrow} & A \\ S & \stackrel{\mathrm{data}}{\rightarrow} & R \end{array} \qquad (15)$$

### 6.2 Edit node attributes

Editing the attributes of an object – e.g. a file, a record, or object attribute – is also a basic management function. Regrettably the general problem is far too complex to cover here in detail, since a node attribute could be expressed in any formal language[32]: e.g. it could be a simple scalar attribute, like a user-name. It could be a line-based file, or a string in a regular language; it could conform to a context-free grammar, which requires complex parsing in order to alter values within a file or database. We assume here that such as method is given; we write only a promise to edit and, in turn, use the promise.

### 6.3 Copy then edit (chained)

Chaining together the primitives for copying and then editing is a common idiom in configuration management: copy a template and then customize. The basic algorithm is as follows:

1. Copy from source $S$ to $X$ if md5 checksum differs, define data1.
2. Copy from $X$ to $Y$ if data are newer (by date stamp) iff (1)
3. Edit data at $Y$ convergently to customize.
4. Copy from $Y$ to destination $D$ by any method.

In terms of promises, we have the following. Let $S, X, Y, D$ be the nodes in the chain, from source to destination.

$$\begin{array}{lcl}
S & \stackrel{\text{hash}}{\longrightarrow} & X \\
S & \stackrel{\text{data1}}{\longrightarrow} & X \\
X & \stackrel{\text{U(hash)}}{\longrightarrow} & S \\
X & \stackrel{\text{U(data1)/!match(hash)}}{\longrightarrow} & S \\
X & \stackrel{\text{data2}}{\longrightarrow} & Y \\
Y & \stackrel{\text{U(data2)/newer(data2)}}{\longrightarrow} & X \\
Y & \stackrel{q\text{=edit}}{\longrightarrow} & D \\
Y & \stackrel{\text{data3}}{\longrightarrow} & D \\
D & \stackrel{U\text{(data3)}/q}{\longrightarrow} & Y
\end{array} \tag{16}$$

### 6.4 Create node

Creation of an object is an interesting case for both operator theory and promises. How can the act of creation be an autonomous one? To make sense of such a promise within the boundaries of the model, one must essentially think of the act of creation and destruction as the setting of an attribute, i.e. we imagine that all objects exist in a ghost state to begin with, and we assign them 'reality' as an attribute. This somewhat artificial notion is the price one pays for a simple model. We believe the price is worth it.

Clearly it is the parent of a node that makes the promise to create a child. This leads us into many issues about whether or not agents should inherit attributes from their parents. One can either:

– Duplicate the parent (fork) with inheritance.
– Create a new blank slate (new born baby - which takes its own decisions with no imprint).

In the first case, one might worry that the autonomy of the child is now in question. However, this need not be the case. As soon as a child is born, it makes its own decisions. The details of its internal state and services do not have to be modelled since they are not modelled in any other agent either. We refer only to what kinds of promises the agents will make.

### 6.5 Destroy node

Once an agent has autonomy, it is an equally tricky question whether or not it can be removed by force. Agents might try to attack other agents. (An attempt by one agent to change the state of another might be regarded as a definition of an attack.)

A node must grant access or permission to destroy itself. Normally an authorized third party like a kernel does this, but then the node has effectively granted the kernel authority. Let $R$ be the resource and $A$ be the policy agent.

$$R \overset{\mathrm{U(destruct)}}{\rightarrow} A \tag{17}$$

The agent agrees to self-destruct on command.

## 7 The Rollback Debate

One of the ideas that frightens system administrators about autonomic computing is the idea that, if a mistake is made (in policy, or implementation), there is no clear way of "rolling back the change" to undo the damage.

System administrators often like to maintain the idea of a version control on their system configurations, as they generally believe that they are in control of every aspect of their configurations. The NETCONF protocol operates with this idea[33], as do many other commercial schemes for change management.

There is a basic conflict between the idea of policy and version control. Policy based configuration management is about control of final state, and the scope of the changes involved in reaching it. This approach to the state is not versioned. Either a system is correct or it is incorrect.

For any operator, there is the possibility of having an inverse, or undo operation. However, in configuration management, in a partially predictable environment (without closure), we need to say what we mean by an inverse. In fact this is not as straightforward a concept as is commonly believed. In particular, if we take seriously the idea that it is *behaviour* that must be governed rather

than state, then one must be much more sophisticated than the simple notion of "undo" in a scheme of change management.

Below we make a semi-formal argument for why roll-back is a fictitious notion (i.e. promises more than it delivers) and then introduce the notion of *restoration* to replace it. The advantage of the latter is that restoration can lead to predictable behaviour as well as predictable state.

### 7.1 Transitions and inverses

Let $R$ be a resource with state $S = \langle op, cf \rangle$, which includes both operational state and configuration state. We want to configure the $cf$ part of this state, and then manage its change over time. Operational state $op$ changes, on the other hand, with every transaction that is made with the stochastic environment in which users interact with the computer via programs. Configuration state is normally only changed by privileged access, but operational state is changed by the very execution of the program.

A program behaviour $B(S)$ is a function of both kinds of state. We do not need to define behaviour in detail, but it is clear that the state is one of behaviours dependencies. Thus the actual future behaviour of the system is being altered both due to changes in configuration (programmed changes) and by machine learning (operational state changes). If we want to constrain system behaviour to make it predictable, then we must not only consider the $cf$ states, we must also extend policy to include behaviour. This means one must also place constraints on the operational state[34, 15].

Consider a change operation $O(cf \mapsto cf')$ that obeys the rule:

$$\frac{\langle op, cf \rangle}{\langle op, cf' \rangle} \; O(cf \mapsto cf'), \tag{18}$$

i.e. the application of the operation to a configuration state $cf$ alters the state to $cf'$. We shall assume that this operation has an inverse, so that it is possible, in the traditional sense, to undo the operation:

$$\frac{\langle op, cf' \rangle}{\langle op, cf \rangle} \; O^{-1}(cf' \mapsto cf), \tag{19}$$

Since this change is a relative change (i.e. a "diff" or "patch"), it is only defined if the initial state is partially known. i.e. the following transitions are undefined:

$$\frac{\langle op, cf_r \rangle}{\langle op, ? \rangle} \quad O(cf \mapsto cf'), \tag{20}$$

$$\frac{\langle op, cf_r \rangle}{\langle op, ? \rangle} \quad O^{-1}(cf' \mapsto cf). \tag{21}$$

where ? means undefined. If such a transition is attempted, it might lead to a further unknown state, or it might simply balk and lead to no change, depending on the implementation. Thus, our ability to undo changes makes the important

assumption that we can predict the initial state. As Couch has pointed out, this requires the notion of a closure[34, 15].

Operational state develops by itself, with nothing more sinister than the passage of time, but it does so in an unpredictable manner that depends on a particular history of stochastically arriving transactions. Let $T$ be the time translation operator.

$$\frac{\langle op, cf \rangle}{\langle op', cf \rangle} \; T : t \mapsto t + \Delta t, \tag{22}$$

In order to compute an inverse to such changes, one needs to record the state $op$ at each moment in time to trace each increment. This has a memory cost that grows linearly with the life of the system.

Now consider interactions with the environment (the world beyond the closure of the change management system): these lead to changes in both operational state (by program execution) and configuration state (perhaps by mistake and conflicting actions that break the closure of the system) in general.

$$\frac{\langle op, cf' \rangle}{\langle op_r, cf_r \rangle} \; E(S \mapsto \text{random}), \tag{23}$$

The state is predictable, hence if such an operation occurs in between a do and undo operation, the final state will be, at best, unpredictable.

The common solution to this problem is to ignore it. Even if no changes occur to $cf$ through environmental interaction, the sequence

$$O(cf \mapsto cf') \; \left\{ \begin{array}{c} T(t \mapsto t + \Delta t) \\ E(S \mapsto \text{random}) \end{array} \right\} \; O^{-1}(cf' \mapsto cf) \tag{24}$$

does not lead to the operational state $op$, and hence does not lead to the same future behaviour in the general case. In short, the concept of rollback is intrinsically unreliable: it does not exist.

If we express these operations as promises from the resource to some observatory agent, the nature of the problem becomes clear. Each relative change operator is represented as a conditional promise.

$$\begin{aligned} O(cf \mapsto cf') &\simeq R \overset{cf'/cf}{\rightarrow} A \\ O^{-1}(cf' \mapsto cf) &\simeq R \overset{cf/cf'}{\rightarrow} A \end{aligned} \tag{25}$$

But a conditional promise that depends on a condition that is no longer true is not a promise at all. The right hand sides are equivalent to $R \overset{\emptyset}{\rightarrow} A$.

The problem with these promises is that, by seeking to make them relative to a given state, we invalidate the promises themselves. Thus we cannot promise what the final state of such a transition might be. The solution is straightforward, we must promise without condition on specific knowledge, and we must constrain sufficiently both kinds of state in order to be able to promise behaviour.

### 7.2 Commutation and non-relative promises

Why are relative changes broken by environmental changes? There are two simple explanations. The first is that the operational changes do not commute with the environmental changes, i.e.

$$OT \neq TO \tag{26}$$

so that

$$\begin{aligned} O\,T\,O^{-1} &\not\to T. \\ O\,E\,O^{-1} &\not\to E. \end{aligned} \tag{27}$$

In the immunity model[4], it is shown that such commutation can be achieved for all cases except existence/non-existence.

The second point is that the promises made by relative changes are conditional and therefore incomplete.

We would like to replace the would-be inverses with convergent promises to absolute fixed points, just like the forward operations. The apparent disadvantage of having convergent fixed points in the forward direction can be seen from the figure 3: there is no bijection between states under the operation of convergent operators. Such operators have no inverse because they are not one-to-one. It turns out that there is an approximate solution that makes a sensible compromise on the difficulties – one simply remembers to point of origin.

### 7.3 A fragile approximation

We can approximate the notion of an operator inverse, for a convergent operator (and hence a limited notion of rollback) by adding memory of state. If an operator can remember its history and one interprets the promise to return each resource object to this previous state using the same convergent and commutative operators that were used for the forward changes, then it is possible to support a limited notion of restoration. A complication, however, is that all operators must have memory synchronized in some manner, with the same version, if restorations are to lead to the correct state. We must also deal with the inherently irreversible effects of creation/deletion.

To counter the problem of undefined states, we make use of internal memory in each resource agent or node. This memory could be limited to a fixed number of recollections in order to avoid a memory explosion. We must observe two rules:

1. Policy must consist of promises about both configuration and operational state.
2. All promising agents/nodes must be versioned collectively, not just the objects that change explicitly.

The protocol for restoring will be as follows:

1. A full set of promises must be removed or withdrawn first, leaving the system unconstrained.

2. A previous version of state must be reinstated from memory.
3. The previous version of all promises must be reinstated, ensuring convergence.

Let $M$ be a memory-equipped agent, and $cf_n$ be the $n$th version of a promise. Then the serial sequence becomes:

1. $(M \stackrel{cf_n}{\rightarrow} A)(M \stackrel{\text{keep history}}{\rightarrow} A)$, for all $M$.
2. The above promises are withdrawn.
3. $R \stackrel{\text{restore } cf_{n-1}}{\rightarrow} A$.
4. $R \stackrel{cf_{n-1}}{\rightarrow} A$ for all $M$.

We feel obliged to point out that any consequences of operating with the new policy (operational state changes) that are not constrained by these promises will not be correctly undone. Worse, we have a distributed versioning problem in the case of rolling back multiple operators, requiring that the versioning operators share some notion of global time (an expensive proposition). There is also the danger that the agent memory will be corrupted. Where shall we keep it? Will it be secure? Will it conflict with other policies (e.g. the policy to delete old data)? These quandaries are unavoidable and suggest how one should use the concept of policy, rather than suggesting operators that should or should not be used.

### 7.4 Creation/deletion

It is clearly not possible to reverse a deletion that has been executed. There is therefore an inherent non-commutativity for deletion[4]. This problem can be avoided in some cases however by avoiding these operations. One can always work from an initial state in which all objects exist in some form (e.g. as empty, inaccessible files) so that the act of creation simply involves making the resource available. Destroying a file can, conversely, be performed either by renaming or making a file inaccessible (using an extra attribute bit). This is plausible for files.

For processes the matter is somewhat harder. It is however, conceivable of making an operating system that can "swap out" and "free up" the resources associated with a process for cold storage. During restoration, this could then be warmed up and reconnected.

Operating systems that support these operations do not exist today, but there would be no serious impediment to creating them. We shall not carry this point further here.

## 8 Conclusions

We have presented an interpretation of the immunity model for autonomic system regulation that uses promise theory to implement self-governing behaviour,

without loss of autonomy. From this model one can build any level of management system one desires by voluntarily subordinating machines and resources in a distributed system.

The advantage of the promise language is that we can easily see the requirements of such a system without being misled by concepts such as "machine" or "network", which are unnecessary for describing the semantics. Moreover, we can describe constraints that we do not necessarily know how to implement yet, such as the full details of operational state.

We discuss the reliability of the concept of "rollback" as used colloquially by system administrators and show that the traditional understanding of rollback is not a behavioural constraint, but a local state constraint that is easily botched. We define a behavioural restoration in terms of promises, which can be implemented without too much difficulty in a variety of cases. However this notion is fragile to loss of memory.

This work is supported in part by the EC IST-EMANICS Network of Excellence (#26854)

## References

1. M. Burgess and K. Begnum. Voluntary cooperation in a pervasive computing environment. *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX) (USENIX Association: Berkeley, CA)*, page 143, 2005.
2. M. Burgess and S. Fagernes. Voluntary economic cooperation in policy based management. *IEEE Transactions on Software Engineering*, page (submitted).
3. M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.
4. M. Burgess. Configurable immunity for evolving human-computer systems. *Science of Computer Programming*, 51:197, 2004.
5. Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.
6. 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.
7. M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, **2**:333, 1994.
8. 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.
9. M.S. Sloman and J. Moffet. Policy hierarchies for distributed systems management. *Journal of Network and System Management*, 11(9):1404, 1993.
10. N. Damiannnou, A.K. Bandara, M. Sloman, and E.C. Lupu. *Handbook of Network and System Administration*, chapter A Survey of Policy Specification Approaches. Elsevier, 2007 (to appear).
11. P.D'haeseleer, S. 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.

12. A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *New Security Paradigms Workshop, ACM*, September:75–82, 1997.
13. M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003.
14. 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.
15. 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.
16. 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.
17. M. Burgess. Computer immunology. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.
18. A. Couch and Y. Sun. On the algebraic structure of convergence. *LNCS, Proc. 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Heidelberg, Germany*, pages 28–40, 2003.
19. G.R. Grimmett and D.R. Stirzaker. *Probability and random processes (3rd edition)*. Oxford scientific publications, Oxford, 2001.
20. M. Burgess and S. Fagernes. Pervasive computing management: A model of network policy with local autonomy. *IEEE Transactions on Software Engineering*, page (submitted).
21. M. Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.
22. A. Høyland and M. Rausand. *System Reliability Theory: Models and Statistical Methods*. J. Wiley & Sons, New York, 1994.
23. E. Lupu and M. Sloman. Conflict analysis for management policies. In *Proceedings of the Vth International Symposium on Integrated Network Management IM'97*, pages 1–14. Chapman & Hall, May 1997.
24. R. Ortalo. A flexible method for information system security policy specifications. *Lecture Notes on Computer Science*, 1485:67–85, 1998.
25. J. Glasgow, G. MacEwan, and P. Panagaden. A logic for reasoning about security. *ACM Transactions on Computer Systems*, 10:226–264, 1992.
26. H. Prakken and M. Sergot. Dyadic deontic logic and contrary-to-duty obligations. In *Defeasible Deontic logic: Essays in Nonmonotonic Normative Reasoning*, volume 263 of *Synthese library*. Kluwer Academic Publisher, 1997.
27. A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proceedings of the 5th IEEE Workshop on Policies for Distributed Systems and Networks*, 2004.
28. A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks*, 2003.
29. A.L. Lafuente and U. Montanari. Quantitative mu-calculus and ctl defined over constraint semirings. *Electronic Notes on Theoretical Computing Systems QAPL*, pages 1–30, 2005.
30. Olaf Zimmermann, Pal Krogdahl, and Clive Gee. Elements of service-oriented analysis and design. In *IBM DeveloperWorks article*. 2004.

31. M. Burgess. Configurable immunity model of evolving configuration management. *Science of Computer Programming*, 51:197, 2004.
32. H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation, Second edition*. Prentice Hall, New York, 1997.
33. IETF. The netconf charter. http://www.ietf.org/html.charters/netconf-charter.html.
34. A. Couch and Y. Sun. On observed reproducibility in network configuration management. *Science of Computer Programming*, 53:215–253, 2004.