# Toward Broad-Spectrum Autonomic Management

Edmund Smith
University of Stirling, UK
edmund.smith@stir.ac.uk

Paul Anderson
School of Informatics,
University of Edinburgh, UK
dcspaul@inf.ed.ac.uk

***Abstract*— The fields of autonomics and system configuration share a common goal in decreasing the cost of ownership of large fabrics. In this paper we present a combined vision in which the technical advances of autonomics and the usability advances of system configuration are merged. We present some early system configuration research that forms the first steps toward this vision.**

Computer *fabrics*[1] are becoming ever more complex and ever more central to the operation of organisations. The cost of a single fabric failure can easily dwarf the cost of employing its administrators [1] but, as for software developers before them [2], IT managers are discovering that simply adding more people to a complex problem does not necessarily provide better results.

Autonomics has as its origin the attempt to improve the robustness of an infrastructure by enabling the fabric itself to respond to changes in its environment [3]. But how can we be sure that the the large-scale properties of the fabric will be maintained as it attempts to heal itself? Allowing the fabric to modify itself and evolve its own solutions will make management more difficult, not less, unless the power of administrators to understand and control their fabric is not also increased.

The authors' background is not in autonomics, but in *system configuration* [4], the study of correctly, accurately and scalably building and maintaining computer fabrics. In this paper we present our vision of an autonomic fabric, a fabric which can recover from many errors without human intervention, yet never be beyond the control or understanding of its administrators. This, we believe, will be a fabric which is affordable, both in terms of the effort needed to maintain it and in terms of its support for critical applications.

## I. Introduction

Much current research focuses on increasing the performance of a fabric [5], and on improving the security measures deployed upon it. But down time due to configuration errors has a significant effect upon costs [6], [1], while the benefits of modest performance improvements may be harder to quantify. Similarly, even the best security tools can be compromised by errors in the configuration of the machines they are deployed upon, or by slowly or improperly patched software.

To understand how we can optimise the value a fabric provides, we must begin by understanding its purpose. A fabric exists only to provide its owners with some functionality or deliverables. The precise requirements will vary widely between fabrics and also over time, but they are the only absolute criterion against which we can judge a fabric's performance at any given moment. We will say that a fabric that is delivering upon its objectives is *healthy*, one that is not, is not.

If these objectives include performances bounds, then steps must be taken to ensure that those bounds are met. A security policy will need to be enforced on publicly networked fabrics, although the details of that security policy will vary depending upon the specific application to which the fabric is put. Our argument is not that performance and security are unimportant, it is that they are considerations only as part of the overall objectives that a fabric is trying to fulfill.

To take the example of autonomic application optimisation, it cannot be clear from the outset that in every case the potential gains in terms of response time will, in terms of raw costs, outweigh even a single failure resulting from parameter optimisation, and as more parameter sets are tried, the higher the probability that poorly tested combinations will be explored. Here the trade-off is seen to be between robustness and performance, and must be decided in the light of a fabric's objectives.

If fabrics are to heal themselves autonomically, they must know what it means for them to be healthy, in the same way that for a system administrator to know what the appropriate way to fix a fabric is, he must know what it has to do. This is the same as saying that a fabric must have a notion of what it is that it is trying to deliver upon, because only in the context of its objectives can it reliably identify a state as unhealthy and carry out meaningful recovery strategies. In the course of this paper, we will outline a vision of *goal-directed activity* on the part of an autonomic system, acting continuously to maintain control, and to restore and enhance compliance with an overall specificiation.

As many previous papers have noted [7], [8], empowering a fabric to play a more active rôle in realising its specification does not, of itself, decrease the cost and difficulty of managing that fabric. We must be sure that the specification itself can be maintained and understood, and that it correctly encodes an organisation's requirements. The complexity of many real fabrics is much too high for them to be understood without abstractions [7]; here we will outline a model of multi-resolution specification, in which finer and finer detail is specified in a smaller and smaller context.

But we will need still more than this for large fabrics

[1]We use the term to denote any large, heterogeneous computer installation.

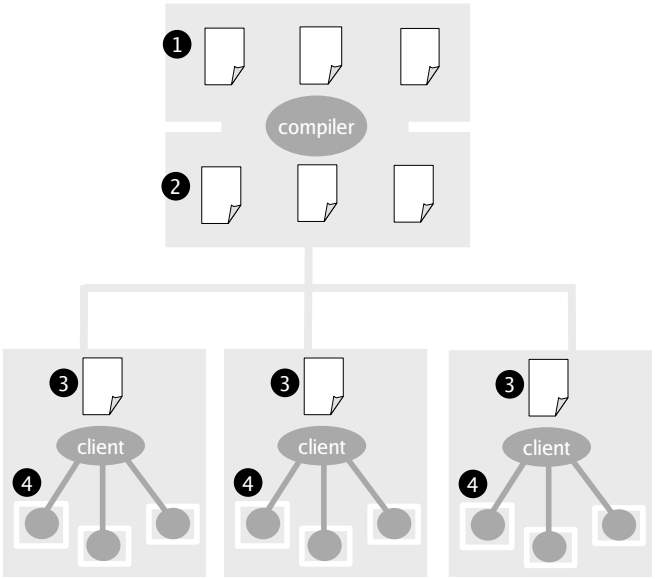


Fig. 1. Configuration with LCFG. (1) Administrators create fabric specifications. (2) These are assembled into machine profiles. (3) Each machine receives its profile. (4) Components act on the profile to configure the machine.

to become truly manageable. Not only must there be many specifications at many resolutions, it must also be possible to divide specifications at a particular resolution along the boundaries of human expertise: there is seldom just one expert administrator involved in the running of a fabric, and many people with different skills typically work together. Dividing a specification in this manner, and merging the different parts together, is called *federation* in system configuration.

The remainder of this paper is structured as follows. The most advanced system configuration systems known to be in use, *two-level systems*, are outlined in the next section as the starting point of our approach. Section III is the core of our vision, based on extending two-level systems to many levels, and incorporating more advanced specifications with autonomic input. Section IV talks about some of the specification technologies which will be needed to make such systems possible, and section V discusses some of the research in this area. Our conclusions are spelled out in section VI.

## II. Two level systems

The current state of the art in the specification and management of large fabrics is the *two-level system* [9], [10], [11] which is split into two components, one for assembling a specification of the target state for each node in the fabric, and one for realising that state separately on each node. These components operate more or less independently, one to reassemble the fabric's specifications whenever administrators change details, the other to reconfigure target machines whenever they diverge from their specification.

In a typical system, each node is associated with a set of property declarations. In the simplest case, these are simply values which can be substituted into the configuration files on the machine, for example the host name, or the address of a DNS server. These declarations are often stored in files on a central server, and several such files may be used to build the specification for each node. This allows for input from several different individuals or teams on the correct specification for a machine, since for fabric of sufficient size, it is usually desirable to split teams along functional lines.

There will be many property declarations for each node, and because several individuals, interests, or aspects, may be responsible for selecting them, some of them may well conflict with one another. The task of selecting which values will be passed on to the clients falls to a *configuration compiler*, which processes all the declarations for each node and selects between them, usually by way of a simple rule in current systems, for example where they occur in the files (order of lexical occurrence).

A configuration compiler provides one other important function: it enables the definition of classes of machine, where machines in the same class share some subset of their configuration properties. A simple scheme of this kind can be developed easily, but even such limited implementations often become the greatest strengths of modern configuration tools. If the sophistication of the compiler is increased, for example by allowing nodes to have membership of multiple classes, even greater rewards can be achieved, at the cost of selection problems when there are conflicting specifications for a single machine from each of its classes. We will return to this problem in more detail later.

The importance of being able to define classes is that it becomes possible to specify the characteristics of a node not in terms of its basic properties like its host name and DNS server, but in terms of the classes which it embodies. If we think of membership of a class as a boolean property in a different domain from our basic properties, then by adding a classing mechanism, we have created a new space of properties, differing from the first set in terms of *resolution*[2]. The generalisation of this approach is at the heart of multi-resolution modelling, and will be discussed further in section III.

One well-established example of a two-level system is LCFG [12], [9], whose rough architecture is presented in figure 1. A set of node files and "header" files are created by administrators. Header files represent data common to classes of machine, and are bound to node files by C pre-processor include directives. These are processed by the LCFG compiler to yield a set of *profiles*: unambiguous property sets describing the intended configuration of each machine.

The reason for explicitly introducing LCFG is the interesting properties of its *deployment engine*, the component responsible for configuring a machine from its specification. Once each machine has received its profile, it is processed by a group of independent components, each of which is responsible for an orthogonal subsystem (orthogonal in that no two components try to configure the same OS entity). Each component is able to determine whether the current

[2]At higher resolution means in finer detail, lower resolution, broader detail.

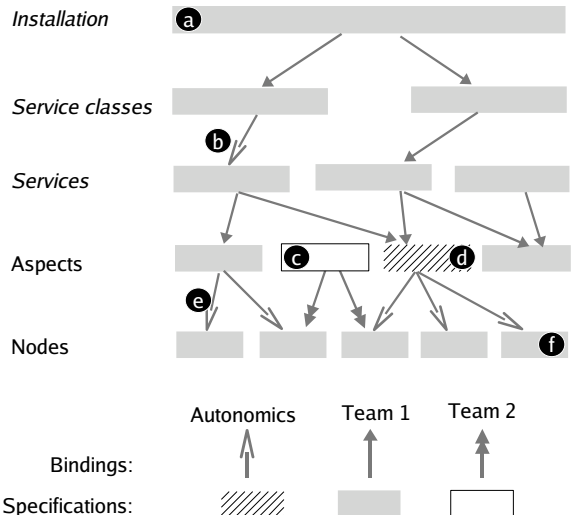


Fig. 2. A sketch of a multi-resolution system. Note that autonomic modules can provide either specification (as in (d), which might be an application parameter tuner), or bindings (as in (b) and (e), which might be a fault tolerance module). (c) might be a separately managed aspect, like hardware with its own specialists. (a) contains a low-resolution specification of the whole fabric's goals, whereas by (f) almost all direct specification is derived from more general network properties.

specification is consistent with its deployed entities, and take appropriate action if it is not. This might be as simple as rewriting a service's configuration files and then restarting it, or as complicated as altering kernel parameters and requesting a reboot.

Components do not need to interact directly because they all act from a common, unambiguous specification, and thus have only to interpret the semantics of that specification in the same way to interoperate seamlessly. This is a powerful and understandable way to manage these large-scale interactions: specify a target state, a *goal*, and work toward it, rather than trying to manage the fabric in terms of behaviours which might achieve that goal.

## III. Many level, multi-resolution, systems

Two-level systems make configuring a node from its specification easy. Current systems are limited instead by the ways in which such specifications can be created and managed. A good example is their inability to specify relationships between nodes or aspects: for a given service, we can specify most of the configuration of client nodes and server nodes easily enough, but correlating shared or dependent properties between them must be done manually (for example, specifying consistent port numbers and host names).

Because there is no explicit representation of the properties of the fabric as a whole, that is properties are ascribed only to the nodes of which it consists, an appropriate low-resolution specification must be both deduced and manually maintained by administrators. This makes it difficult to separate logically distinct elements of the specification, preventing any automated oversight, and hindering attempts to integrate the work of system administrators (and, as we shall see later, of autonomic modules).

Our vision is of a solution in terms of *multi-resolution modelling*, in which fabric configurations are specified and maintained at many different, explicit levels of abstraction. But what constitutes a well-defined level of abstraction, and how should more be introduced? Our view is that there should be two parts to defining a new, lower-resolution view of the system: the specification of a higher resolution description for each new lower-resolution property, and then the binding of lower resolution properties to higher resolution entities.

In the case of the class mechanism of existing systems, the first part is readily apparent, being the class definition in which property values shared by all nodes in that class are listed. The binding of classes to nodes is usually done "upside down" at present however; each node specifies a list of classes from which it inherits, rather than our (functionally equivalent) definition of each class being bound to a list of nodes which embody it.

But we can take this approach much further. Consider a potential, lower resolution description of the properties of a fabric in terms of services which are deployed upon it. Suppose, for the sake of argument, we consider a service to consist of a set of clients and a set of servers; the clients needing to be configured to use the correct servers, and each server potentially needing a list of its clients. Let us say that a specification at this resolution is just a server and a list of clients for each service.

We will need to have separately defined the bindings for each service in terms of a set higher resolution properties which must be correlated between the server and the client. Clients might need their server host name property for that service to match the server's host name, and servers might need a map containing some information for all their clients (for example, a DHCP server will need a list of all its clients' MAC addresses). Both clients and servers will need to be configured to make the service available, which we can model as them inheriting certain classes[3].

By creating this lower resolution specification, we have made the knowledge of the desired fabric services generally available, facilitating reasoning by both administrators and tools. We have also created a natural level for a powerful fault tolerance module, which can now simply manipulate the choice of nodes for server and client rôles based, for example, on monitoring information. This fault tolerance module has a well defined interface for interacting with administrators, and is thus manageable by them. In essence, it is similar to a junior member of staff who sleeps at his desk, changing the service assignments when monitoring detects a problem.

The key point here is not the exact form that service specifications take, but the idea that specifications at multiple resolutions, coupled with bindings between lower and higher

[3]Were we to allow classes to be parameterised, we could make our correlated data parameters of those classes, and thus deal only with properties one resolution higher (classes). This detail isn't essential for the scheme to work however.

resolution entities can be used to model a fabric in a powerful and intuitive way. Above services, we might define clusters, virtual organisations, or sub-fabrics which are largely self contained with well-defined interfaces and provisioning agreements. In some environments, more tiers would be needed, but the approach remains the same: the expression of specification in terms of lower and lower resolution views.

In this outline then, autonomic modules act like administrators with limited scope and authority. We have already seen one example, changing low resolution properties to provide fault recovery. Autonomic modules could equally operate at the highest resolution, tweaking the parameters of an application to optimise its performance, as suggested in [3]. By bounding the authority and scope of the module, we can be confident that it cannot modify parameters beyond a safe ranges, and that it can be overridden where necessary by human controllers.

For autonomic modules operating at lower resolution, the benefits are also clear. By working as part of a team with human and other autonomic modules, they can be made much smaller and simpler, and should need less context than they might otherwise. With an explicit, visible, overridable and queryable interface, administrators are more likely to trust an autonomic module to operate, and guaranteed hard bounds on what it is possible for the module to do protect the organisation in the event that it fails.

We believe this vision offers enormous power for autonomic behaviour. To create a complete automated system administrator from scratch will remain unrealistic for the forseeable future, due to the sheer complexity of the systems, their rate of evolution, and the amount of expertise that is required. By working with administrators to specify a configuration, and in turn leveraging their expertise, autonomic systems could much more rapidly become a useful part of the systems workplace.

## IV. Working with specifications

In the previous section we presented the multiple resolution model of fabric specification. The crucial bar, at present, to the realisation of such a model is the lack of sufficiently sophisticated tools for working with such specifications. Such tools, which will be needed to build a coherent network specification from many sources at many resolutions, are in their infancy at the time of writing [14]. This may be attributed both to the slow uptake of two-level systems and to a lack of vendor engagement [15]. Here we present an overview of some theoretical problems in this area; the next section summarises prototypes and current progress.

### A. Aspect orientation

In section II we noted that we might wish a node to embody more than one class. When working with specifications, this is usually called *aspect orientation*, to prevent it from being confounded with multiple inheritance in object-oriented programming (specifications do not compose like objects). In the sketch of a multi-resolution system we presented (see figure 2) aspects were the first level of abstraction, equivalent to node rôles.

Because of the significant part aspects play in mediating between administrators and between concerns, the way in which it is most useful to compose them is still unknown. If multiple aspects disagree on the value of some property, are there meaningful ways to decide between them, for example can authority be vested in individuals (authors), declarations, aspects or particular choices of value to make this mediation automatic? An effective solution to this problem will steer between providing brittle, inflexible boundaries with too much negotiation required between concerns, and loose boundaries in an unpredictable hierarchy, in which it is hard to know which values will dominate.

### B. Authorisation

It will not be sufficient in the systems of the future to assume that every possible source of specification is entitled to modify any configuration properties it chooses, at any level of configuration. Determining which entities a source can legitimately interact with, and in what ways it can do so, is the problem of authorisation. Authorisation can occur concurrently at many different layers and in many different places in the system, both to force clean boundaries between administrative teams and to provide protection against software errors.

To illustrate the first case, consider that some sites will find it desirable to ensure that the security team is not overruled without consultation, and that any deviations from their specifications are cleared with them personally first. More generally, making sure that teams interact properly at certain boundaries of authority, rather than simply changing each others specifications, is likely to be generally desirable, and can be enforced by an authorisation system.

To see the second case, take the example of our autonomic application tuner. We can prevent it from damaging the configuration of any other application, and from rendering the application non-functional by an ill-formed or invalid choice of parameters. This may allow us both to code the tuner more simply and generally, and also to trust it to operate (as we pointed out in section I, without such guarantees some sites will be wary of deploying such a tuner, given the huge costs of downtime). More generally, we can enforce boundaries on the operation of autonomic software, reducing the risk that errors will compromise service provision.

### C. Loose specification and constraints

Once the system has been split into multiple intersecting aspects, it is often more desirable to use an aspect to express constraints on what a particular value should be, rather than choosing a value. Often there are a set of reasonable values for each property, yet because authority for the value is delegated between multiple aspects, the writer of each aspect feels obliged to provide a specific value. This can give rise to many unexpected, artificial conflicts between aspects.

The obvious solution is to allow values to be specified more loosly, using constraints. The sheer scale of the fabric immediately makes this difficult (for example, a mature LCFG fabric

may specify 5,000 properties per node), and has prompted a search for non-optimal solvers which provide some looseness, but much more rapidly.

For the two essential configuration property types, maps and scalars, a variety of useful constraints have been proposed. For maps these include, partial ordering of elements, and both key and value inclusion and exclusion (specifying that the map must contain such a key, or must not, or must contain a key with such a value, or must not). For scalars, these have included equality, set membership (the scalar's value must be one of a set), regular expression matching, and referencing (the scalar's value must be equal to another scalar's value) [16].

### D. Validation

Related to the concept of constraints on values, is the idea of validating those values. This can be used to catch typographical and other simple errors. An easy example is validating that where an IP address or MAC address has been entered, the address is valid and reasonable (any degree of validation from checking it consists of the right format, to checking it falls within the fabric's range of addresses, is reasonable and useful).

Validation can also play a more significant rôle in checking that large scale properties are maintained, and checking for errors that might have arisen. When a specification is built from many parts by a complicated process, such an independent verification can be important, and difficult to impossible to do by hand. This idea of correctness of the specification is one of the few metrics of interest. As simply generating the fabric's specification from correctness properties is impractical, both in terms of complexity and in terms of the amount of other data that must be somehow specified, we suggest instead that it is significantly cheaper and more practical to test a configuration for correctness, and reject it if it is not.

For an idea of what such correctness consists of, consider a network in which there are many servers and consumers of the services they provide. With many sources of configuration, and many people involved, it can be hard to know that all clients are expecting services from servers which are actually trying provide them. Yet this is something that can be asserted logically and verified at compile time, before ever reaching the network. Another example is, when using virtualisation, to check that redundancy is not compromised by mapping multiple redundant instances to the same physical machine. This too can be asserted logically and verified.

### E. Workflow

A final problem we consider here is the transition between specifications, something which occurs regularly and must be carefully managed. Although we have so far considered a specification as a static statement of the target of the system, which is to be achieved as rapidly as possible, in practice, not all routes between specifications are equal nor acceptable. Downtime is often intolerable, and migration must be as transparent as possible.

The workflow problem is to find a path between an existing specification and a new specification which maintains their critical shared properties. At a network level, this could entail, when a service is specified in both specifications, but the servers which provide it change, finding a path between the two such that service provision is constant, despite some servers ceasing provision, and some beginning provision. Such solutions can be found by hand, but the process is both time consuming and error prone.

## V. Relevant implementations

This section provides a brief and incomplete survey of research in this area which has lead to implementations, both prototype and well-established.

### A. Current production tools

*1) LCFG:* LCFG is probably both the most advanced two-level system, and the oldest still in use. It provides a form of aspect orientation, using C pre-processor include directives to pull aspect headers into a node's specification. Declarations are resolved between using file ordering (the last line in the node's specification has the highest priority).

Individual property values can be validated using regular expressions to ensure well-formedness, also keys in maps can be specified in terms of their partial ordering, and LCFG will perform a topological sort to provide the given ordering. This is used, for example, to manage the order in which `init` scripts run.

LCFG also provides an example of a primitive service-resolution specification in the form of *spanning maps* which enable a list of clients and data about them to be automatically collated for each server, via a publish/subscribe mechanism. The interested reader is referred to [17].

*2) SmartFrog:* SmartFrog[4] is an advanced fault tolerant system, focussed primarily on deployment and error recovery. Experimental work was done to combine it with LCFG to provide fault tolerant services [18]. Applications and services are bound together loosely at run-time, and in the event of failure, a new instance can be deployed from an aspect-like specification.

*3) Alloy:* Alloy [19] is an advanced example of a configuration system, which takes specifications in predicate logic, and uses an advanced solver to efficiently create network transport configurations. This has many conceptual advantages, however doubts must remain about its extension to all properties of all machines in an infrastructure; even if this can be done efficiently, predicate logic may be too difficult to allow day to day modifications to be made in it.

### B. Recent prototypes

*1) mcfg:* mcfg[5] is an experimental *aspect compiler* which takes a specification of a network in terms of any number of aspects and nodes, each of which may inherit from any number of aspects in any combination. The properties of all aspects

[4]Standardised as CDDLM in the Global Grid Forum.

[5]All prototypes are available from the authors on request.

from which a node inherits are resolved together using rules of precedence based on the priority of a given constraint and its locality in the aspect graph of that node.

Its other contribution is to offers two simple constraint operators for maps: inclusion proposes that the map should contain a scalar whose value is the given value, exclusion proposes that the map should not contain a scalar whose value is the given value. These are resolved using broken resolution (resolving one property at a time) which is fast but does not allow all legitimate cases involving references to be resolved.

*2) cfgas:* cfgas is an experimental authoriser, which acts to restrict the resources whose value can be "tainted" by any particular user. The prototype implementation uses the file system for authentication. It takes a set of specifications in a standard format [14], and requires that each value be marked with provenance information regarding the files that were involved in deciding its value. Each file is examined to determine which local users could have modified it, and this is combined with a set of policies specifying which resources may have been edited by which users in which generated specifications to decide whether the specification is authorised.

*3) cfgw:* cfgw can take a group of specifications and a set of predicates in first order logic detailing correctness properties, and assert either their truth or falsity. Variables may be introduced by quantifiers (existential or universal) and have a domain of either the set of machines specified or a set of resources in a map. Any predicate that cannot be shown to be true can trigger either output of a predicate specific warning or a full truth graph, justifying the decision to reject it.

## VI. Conclusions

This paper presents a framework for the integration of the best elements of modern autonomics research and system configuration practice. The objectives of these two areas are the same, to affordably and effectively control fabrics, yet there has thus far been little overlap between them.

Our vision is primarily of clear specification of the target state of the network at all levels, and of goal-directed action which can always be understood as having this specification as its objective. When behaviour rather than an objective is specified, fewer options for delegation, automation and validation available. Equally importantly, clarity is compromised, and in this environment, the cost of mistakes can be astronomical. Ensuring that administrators understand their networks, especially when autonomic systems are operating on them, will remain the key challenge for designers in the future [7], [8].

When specifications lie at the centre of the operation of the network, it becomes possible to integrate the work of both autonomic tools and system administrators together in a coherent way. Production implementations in this area lag well behind research, but we presented a brief survey of the capabilities available in the field, and some prototypes created by us to investigate these issues.

Autonomic systems have yet to find their rôle in the day to day workplace of the system administrator. Although many aspects of our story are still incomplete, we believe our core vision of a multi-resolution autonomic fabric, in which autonomic modules and system administrators interact through explicit specifications, is both a powerful and a realisable one, and offers a way forward for the integration of autonomics with existing staff and systems.

## References


[1] D. A. Patterson, "A simple way to estimate the cost of downtime," in *Proceedings of the Sixteenth Systems Administration Conference (LISA '02)*. Usenix, 2002, pp. 185–188.

[2] J. Frederick P. Brooks, *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1978.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer Magazine*, January 2003.

[4] P. Anderson, *System Configuration*, ser. Short Topics in System Administration, R. Farrow, Ed. SAGE, 2006, vol. 14.

[5] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, "Managing web-server performance with auto-tune agents," *IBM Systems Journal*, vol. 42, no. 1, 2003.

[6] D. Oppenheimer, "The importance of understanding distributed system configuration," in *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, April 2003.

[7] R. Barrett, E. Kandogan, and J. Bailey, "Usable autonomic systems: The administrator's perspective," in *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, 2004, pp. 18–26.

[8] D. Russell, P. P. Maglio, R. Dordick, and C. Neti, "Dealing with ghosts: managing the user experience of autonomic computing," *IBM Systems Journal*, vol. 42, no. 1, 2003.

[9] P. Anderson and A. Scobie, "LCFG: the next generation," in *UKUUG winter conference*, 2002.

[10] R. G. Leiva, M. B. López, G. C. Meliá, B. C. Marco, L. Cons, P. Poznański, A. Washbrook, E. Ferro, and A. Holt, "Quattor: tools and techniques for the configuration, installation and management of large-scale grid computing fabrics," *Journal of Grid Computing*, vol. 2, no. 4, December 2004.

[11] N. Desai, A. Lusk, R. Bradshaw, and R. Evard, "BCFG: A configuration management tool for heterogeneous environments," in *Proceedings of the 5th IEEE Conference on Cluster Computing*, 2003, pp. 500–503.

[12] P. Anderson, "Towards a high-level machine configuration system," in *Proceedings of the Eighth Systems Administration Conference (LISA '94)*. Usenix, 1994, pp. 19–26.

[13] M. Burgess, "Computer immunology," in *Proceedings of the Twelfth Systems Administration Conference (LISA '98)*, 1998, p. 283.

[14] P. Anderson and E. Smith, "Configuration tools: working together," in *Proceedings of the Nineteenth Systems Administration Conference (LISA '05)*. Usenix, 2005, pp. 31–37.

[15] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field studies of computer system administrators: analysis of system management tools and practices," in *Proceedings of the 2004 ACM conference on computer supported cooperative work*, 2004, pp. 388–395.

[16] A. Holt and J. Hawkins, "Making collaborative system administration easier: constraints and declarative aspect precedence," in *Proceedings of SAICSIT 2004, Stellenbosch, South Africa, 4–6 October 2004*, 2004, pp. 249–253.

[17] P. Anderson, *The complete guide to LCFG*. [Online]. Available: http://www.lcfg.org/doc/guide.pdf

[18] P. Anderson, P. Goldsack, and J. Peterson, "SmartFrog meets LCFG: autonomous reconfiguration with central policy control," in *Proceedings of the Seventeenth Systems Administration Conference (LISA '03)*. Usenix, 2003, pp. 213–222.

[19] S. Narain, "Network configuration management via model finding," in *Proceedings of the Nineteenth Systems Administration Conference (LISA '05)*. Usenix, 2005, pp. 155–168.