# Configuration Tools: Working Together

*Paul Anderson and Edmund Smith* – University of Edinburgh

## ABSTRACT

Since the LISA conferences began, the character of a typical ‘‘large installation’’ has changed greatly. Most large sites tended to consist of a comparatively small number of hand-crafted ‘‘servers’’ supporting a larger number of very similar ‘‘clients’’ (which would usually be configured with the aid of some automatic tool). A modern large site involves a more complex mesh of services, often with demanding requirements for completely automatic reconfiguration of entire services to provide fault-tolerance. As these changes have happened however, the tools available to provide configuration management for a site have not evolved to keep pace with these new challenges. This paper looks at some of the reasons why configuration tools have failed to move forward, and presents some suggestions for enabling the state of the art to advance.

### Background and Motivation

*Configuration Tools* have been an important theme at LISA for many years, and most conferences include one or more papers in this area. Despite increasing recognition of the importance of the configuration problem, there remains both a lack of conceptual commonality and a lack of progressive innovation in the area.

No significant standards have so far emerged, and new tools often merely espouse variations on existing approaches, frequently using both completely new specifications and code. For example, ten years separate the initial description of LCFG [5] and the presentation of Newfig [12], yet there is no clear evidence of conceptual progress. There is little or no attempt to create standards in a way which would reduce the barrier to new development or enable a greater shared understanding of the configuration problem.

Within the configuration management community, a great deal of ongoing discussion revolves around this apparent failure to either successfully disseminate the concepts underpinning existing tools, and the experiences gained from those tools, or to realise them in tool implementations suitable for a wider audience.

Furthermore, despite the slow rate of progress, no sign of convergence between tools is apparent. There are essentially no standards in this area, with each tool being not only entirely coded from scratch, but also unable to interoperate with other tools, or share configuration data with them. Although the Configuration Description, Deployment and Lifecycle Management (CDDLM) working group of the Global Grid Forum (GGF) has published a standard interface for configuration tools on grid fabrics, their focus is upon exposing a web-service interface to the underlying configuration system, which falls somewhat wide of the mark in addressing the current problems faced by system administrators.

This absence of standard tools has lead to many large sites developing their own tools. To create a functional configuration management system in its entirety, however, is a significant undertaking, and even those sites which have been able to invest sufficient resources in a system to make it sustainable have been unable to gain a wider community of users. It seems likely that in many cases, the sheer difficulty of developing and maintaining a large monolithic system (often written by people with a shortage of time and no background in software development) has limited both the functionality and the portability of tools.

It is surprising indeed, given the importance of this area, that we find ourselves typically unable to recommend any current system to interested users.

### Levels of Configuration

Perhaps because of the above difficulties, most existing tools deal with configuration specifications at a very primitive level; they involve ‘‘files’’ and ‘‘permissions’’, rather than ‘‘high-level’’ concepts such as ‘‘services’’, and the relationships between nodes – Figure 1 shows some typical statements at increasingly higher levels of abstraction.

Ultimately, the requirements for a service are always expressed in high-level terms and most tools require these to be manually translated first into some lower-level requirements. As systems become more complex, this process is an increasing source of errors. Manual intervention is also unacceptable for autonomic systems which must reconfigure automatically in response to demand and failure. A future generation of configuration tools will need to be able to accept high-level requirement specifications, and to reason in much more sophisticated ways in order to determine the appropriate details. This is an inevitable consequence of the growing complexity of the modern installation, and the fact that the number of administrators a site needs does not scale linearly with the number of machines managed.

There is an analogy with computer programming here: early languages provided very little abstraction from the underlying machine, but as the size and

complexity of systems has grown, so has the separation from the hardware operations. In a similar way, configuration tools must adapt to aid administrators to manage the complexity of their systems.

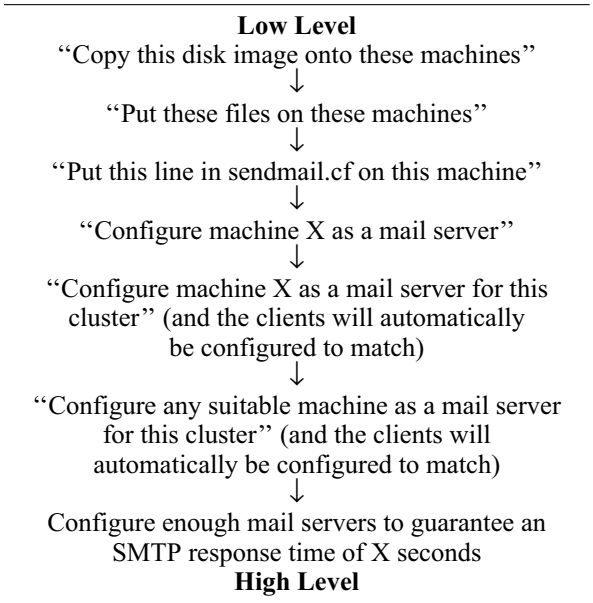


**Figure 1**: Levels of configuration specification.

## The Configuration Tool Zoo

There are a plethora of tools available which purport to address some aspect of the configuration management problem. Many of these are described in more detail in surveys such as [6]. However, we present the following examples as typical, in that they demonstrate a variety of the most common approaches. In particular we note that:

- The tools have different *lexicons* – for example, there is no common agreement on how to specify the "mail relay" for a particular client.
- They use different languages and file formats.
- Their deployment operations provide different guarantees regarding pre- and post-conditions.

In general, these tools are also intended for direct manual use and present no intermediate interfaces between the human-generated specification, and the direct manipulation of the target machine.

### Arusha

Arusha [11] uses an XML-based source language for describing configuration information. It is intended to facilitate sharing of configuration information between sites.

### LCFG

LCFG [5, 7, 2] has a site-wide repository of declarative configuration data which is compiled into XML descriptions for transmission to the individual hosts. Clients execute *components*, each of which takes its specification from a corresponding section of the XML *profile*.

### Quattor

Quattor [3] has a similar architecture to LCFG, differing principally in the exact forms of the components and the capabilities of the central compiler. Both Quattor and LCFG have been used successfully to specify "complete" configurations for large, complex sites.

### BCFG

BCFG [9] is a new tool from Argonne National Laboratories, based upon a highly centralised model of configuration construction. Complete configuration file sets are computed centrally for each client (together with attribute data), before being distributed in an XML envelope.

### Cfengine

Cfengine [8] is probably the most widely-used "configuration tool," but perhaps the most difficult to fit into a common model. It is frequently used to specify only partial configuration information, and its specification language is less well suited to manipulation by independent programs.

### The CDDLM Standard

The CDDLM standard proposal [1] is interesting in that it represents the first formal attempt at a standard in this area that we are aware of. It is a proposed international standard which provides a tool-independent specification for the transmission of configuration information. The intention is to provide complete configuration control, aimed as it is at the management of grids – a highly federated environment requiring autonomic operation.

## A Common Model

Despite the range of views and models set out earlier, there are in fact two distinct capabilities a modern system must provide:

1. Deciding what configuration a particular node should have.
2. Creating the desired configuration on a node.

This is the difference between, for example, deciding that a machine must have a package installed on it, and actually installing that package. The fundamental idea is to separate these two distinct roles into separate systems: systems that communicate through a configuration description. For the sake of brevity, in future we will call a tool able to decide what the configuration of a node should be a *configuration management system*, and a tool capable of creating a configuration on a node a *deployment engine*.

Writing a deployment engine requires an enormous amount of code, and keeping it current requires continuous maintenance. The systems we are asked to manage were not typically designed for neither bulk nor automated maintenance. There are a myriad of file formats, a slew of ways to restart a service, endless configuration files and preconditions. Each of these

will typically find a place somewhere within the deployment engine.

At present, each configuration tool author begins by writing a deployment engine, and many deployment engines exist (albeit inside a configuration tool, and not accessible independently). Once the deployment engine is complete, whatever time and resources remain can be devoted to configuration management. Unfortunately, as we mentioned earlier, the amount of time and resources remaining tends to be small, and configuration management has remained in a relatively primitive state for many years. At present, the only mechanism typically available is the ability to group nodes and configure them as a unit.

An overview of some existing configuration management operations is given in ‘‘Managing Configuration Data’’ together with a discussion of proposed alternatives. We argue that the core of the difficulty in advancing the state of the art is that each current project must include both a deployment engine and a configuration management tool. Whilst deployment engines are well-understood and well-developed, they represent such a large amount of work that must be performed before configuration management can begin that no time is available to develop advanced configuration management systems. This is despite the fact that it is not our inability to deploy a configuration on a node that makes a large modern installation so hard to manage, but that the amount of information involved is too large to be managed in an ad-hoc fashion.

By separating out the deployment engine from the configuration management tool, we make it possible to implement one without implementing the other. We believe that this could lead to a rapid advance in the sophistication of management tools. In addition, we believe that many of the existing configuration tools discussed previously could be released as deployment engines relatively easily, as their configuration management functions are both small and peripheral, so one might imagine easy to isolate. This is certainly true for LCFG, for example.

**A Document Interface to Deployment Engines**

In this paper we aim to convince the reader that if deployment engines conformed to a minimal configuration description standard, it would allow the independent development of configuration management tools. This appears the only plausible way to relieve the current stagnation in configuration tool development, and enable the creation of a new generation of configuration tool, capable of managing a modern network.

This section presents one possible interface to deployment engines, which we believe is most usefully realised as a document. This is similar to the familiar tool chain for program development (e.g., autoconf, automake, make, gcc, ld, etc.) More important than the format of the document is what such a document represents:

A *configuration description* specifies a complete, unambiguous and instantaneous configuration for the target machine or system.

- *Complete* means that everything the deployment engine is capable of specifying is given a value.
- *Unambiguous* means that no further logic is required to determine the parameters of the deployment engine. There are no references, no late binding, nor any database queries required.
- *Instantaneous* means that the specification is a snapshot of a node’s state that the engine should be working towards. There is no time dependency stated.

Our proposed document format is XML-based, simply on the grounds that XML is a widely-used and convenient way of representing structured data such as configuration parameters. Most languages provide XML parsers and processing libraries which simplify the development of cross-platform libraries and tools. There is no suggestion here that tools should provide users with an XML interface, only that it is a convenient format for exchanging data between a management tool and a deployment engine.

***Entries***

```
<entry name="foo">
 <value>bar</value>
 <origin>
  <file name="foo.cfg" line="132"/>
 </origin>
</entry>
<entry>
 <value>baz</value>
 <origin>
  <file name="anon.cfg" line="211"/>
 </origin>
</entry>
```

An entry represents a data item. An entry may be either named or unnamed. When named, the entry represents a key-value pair, the most common method of storage for configuration data (e.g., [10, 5, 9, 3]). The entry’s name attribute is the key, and the value is inside the value element (which must be text – there may not be any nested elements.) In the example, the key ‘‘foo’’ was bound to the value ‘‘bar’’. An entry may also be unnamed, for example for use as a member of a list of values. The origin element is optional, specifying a list of files and lines which were used to determine this value. It is purely to enable a deployment engine to give meaningful errors when the files the user edited were passed to a different tool.

***Structs***

```
<struct name="sudoers">
 <entry>
  <value>joe</value>
 </entry>
 <entry name="admin">
  <value>jane</value>
 </entry>
</struct>
```

Like entries, structs have an optional name. They provide a means of structuring data items. Structs may contain both named and unnamed entries and structs. A struct containing only named entries can be thought of as a record, a struct containing only unnamed entries is a list. The elements contained in a struct are ordered. Structs may not contain character data (only structs and entries.)

***Configurations***

```
<configuration target="lcfg-engine"
               version="1.0.3">
 <struct name="applications">
  <entry>
   <value>emacs</value>
  </entry>
 </struct>
</configuration>
```

The configuration element is the root of the XML document. It serves to contain all the other elements. The target attribute indicates the deployment engine for which this configuration is intended, and the version attribute a minimum version number for that engine. This is largely to provide sanity checking (you wouldn't want to accidentally hand a configuration to an old version of the tool and have it trash a machine in confusion.)

**Avoiding the Phantom Lexicon**

This section has presented one possible approach to defining a ''low level'' configuration description document format. We have sidestepped the issue of a standard lexicon (that is, defining a standard set of keys whose values should be similarly interpreted by all compliant deployment engines.) We argue that such a standard lexicon could not be meaningfully defined in the present environment, where there is so little commonality between existing tools.

Our purpose is to enable progress towards greater sophistication in configuration management by providing a standard method of interfacing with low level deployment engines. Whilst a standard lexicon would both allow sites using different deployment engines to share configuration data, and allow them to migrate easily between engines, we do not believe there is enough understanding of what the appropriate lexicon would be to define it at present. The most pressing need in this area is to begin to decompose the problem. When the issues involved are better understood, it will be possible to standardise a lexicon. If we cannot start moving forward, we will never reach that point.

The choice of the ''level'' at which to specify this interface is crucial; if the level is too low (it contains too little structuring information), then it will not be possible for common tools to perform meaningful operations on the configuration. If the level is too high, then the mismatch between the common format and the assumptions of existing tools will be too great, and no meaningful translation will be possible. This proposal is based on a study of the existing tools in the previous section (and others), and we believe that an interface at the level presented is the minimum required to support useful common operations (see the next section), while providing a large degree of architectural freedom for the tools themselves.

## Managing Configuration Data

It might seem that the interface set out earlier is too limited to support sophisticated management techniques. In this section, we will attempt to show that a wide range of transformations can be applied to structured configuration data, without needing to have the management tool ''understand'' that data. There is an analogy here with the use of simple UNIX command line tools, applied in a filter chain, to perform more complex tasks: tools such as sort, grep, m4, and even awk can be combined to perform powerful transformations on text files, even though the programs themselves have very little knowledge of the text file structure apart for a simple division into fields and records.

**Classing**

The most widespread operation available in configuration tools to manage configuration data is classing: groups of machines are assigned to particular classes, with configuration data being associated with each class. Alternatively, this can be seen as establishing predicates about machines, and associating with each predicate a set of configuration information.

A simple example might involve a ''webserver'' class, associated with configuration data that specifies webserver specific packages and configuration options. In order to deploy a new webserver, it is only necessary to place it in the ''webserver'' class, and all the necessary configuration files and packages will be deployed.

Basic classing gives the administrator an enormous increase in the manageability of their network. By separating out roles into classes, it becomes possible for an administrator to create complex combination configurations very quickly. Complex updates can also be rapidly deployed, as changes can be made to class definitions rather than to individual machines: if a bug is found in xinet on Fedora Core 3, it suffices to update the class definition for ''Fedora Core 3 machine,'' which all FC3 machines inherit. There is no possibility of missing, or overlooking a machine, nor is it necessary to modify each machine by hand.

***Advanced Classing Operations***

Whilst almost all configuration tools provide their own mechanism for managing simple classes of objects, this feature still remains to be fully explored. The simple scenarios described above are both powerful and useful, but as the use of automation matures, administrators find there are unanswered questions in this approach. Consider a ''highly-secured'' class, and a ''web-server'' class, each managed by different teams. These classes have different objectives, and

overlapping domains. It is entirely possible specifications from one class will conflict with another. To our knowledge, no current configuration tool presents a satisfactory answer to the question of how conflicts can be resolved.

Current suggestions for progress in this area include:

- **Prioritisation** A machine might belong to different classes with different priorities, enabling the data specified by one class to be outweighed by another. Or classes might set values with different priorities, so allowing them to be safely overridden elsewhere. This would avoid the order-dependent (or even oscillatory) decision process displayed by current tools.
- **Constraints** Rather than specifying definite values for properties, it might be possible to specify ranges of values. This would enable a tool to mediate between different groups automatically. Consider the web-server team specifying that either port 80 or 8080 be used, and the security team that only ports over 1000 be used. These requirements are not conflicting, but had the webserver team simply said port 80, there would be no way for the tool to mediate.

**Aggregation**

It is common for some aspect of the configuration of a system to be derived from the configurations of other machines. This might be a client being configured with respect to its servers, or a server from its clients. Specifying this information manually is error prone and time consuming. By introducing automation, we both save ourselves time and increase our confidence in the correctness of our network's configuration. Typical examples include:

- A firewall host may want to open holes for services marked as public. Specifying this in the configuration of the hosts, rather than in the configuration of the firewall, means that when the host's service is decommissioned, so is the firewall hole (automatically).
- A DHCP server might want to allocate fixed addresses to individual nodes. Again, the fixed address for a host is most naturally stored with the host itself. This means the server must aggregate those values into its configuration. This can be done systematically by a management tool.

The only tool we are aware of that provides explicit aggregation operations is LCFG, although Quattor and others have the ability to derive parts of a host's configuration from a database. Future research in this area is likely to focus upon the ways in which aggregation can be provided without a central configuration server – it has been suggested that one way to achieve this would be through the use of peer-to-peer technologies.

Again we note that, given just a structured tree of data, it is possible to implement a generic aggregation function, for example by converting values across a group of files into a list of values in the target file.

**Sequencing and Planning**

Configurations cannot be changed atomically, as there are frequently multiple dependents upon a single value. For example, it may be necessary to both bring up a new server and redirect all clients to point at it, before the old server can be taken down. Thus far we have avoided talking about how configuration changes can be sequenced. To the best of our knowledge, none of the available tools tackle this question.

One possible approach, as yet untested, is to allow the user to specify a set of invariants that must remain true during the deployment of any configuration. These can be used to break down a deployment into multiple steps, each of which maintains the invariants. One example of an invariant might be "A client must not be configured to point at a non-existent server."

Although there can be significant sophistication required to maintain an arbitrary invariant, simple invariants of the kind given above can be shown to generate safe 3-step transitions between configurations. Of course, some integration with monitoring, or user-feedback, would be needed in a real tool to step through the intermediate configurations. It does little good to create several intermediate steps then deploy them all at once!

The important point here is that the specification of invariants can be done in a general way (all that is required is to be able to establish requirements on pairs of values – to the tool it is unimportant what those values represent.) This is an area in which even very simple tools can generate large advances in the state of the art.

**Delegation and Authentication**

A large installation is not managed by a single person. Often there will be several teams working together, each tasked with maintaining different aspects of the network's configuration. We discussed the possibility of using a configuration management system to perform some automatic mediation between teams. Here we are concerned with the security issues involved in delegating configuration aspects to people outside of the central teams.

Configuration management systems at present have a boolean notion of authorisation: a user is either authorised to perform any action whatsoever on any machine, or they are not authorised to perform any. Although we might feel that there are many people involved in the management of a system who should have some limited control of some aspects of a machine, current tools do not enable us to act upon that idea.

The simplest example is allowing users to control their own machines to some extent. It seems that many sites allow users a choice either of complete

freedom and responsibility, or of accepting a completely prescribed system. Machines may have gigabytes of unnecessary software installed on them because users are unable to choose which software is most applicable to them.

The solution to these problems is to introduce a notion of limited delegation, or authorisation, into our tools. For example a constraint-based system could set hard boundaries on what a user can do to their machine, while still providing much more flexibility than is currently available. Nor would it be necessary for users to understand the tools if unprivileged user-space helpers were available to guide them through the options.

Again, we believe that authorisation and limited delegation are features that could be explored without a known underlying lexicon to work to. Restrictions and security information can be identified with areas of the configuration data tree, and are no less securely or usefully enforced for not being understood by the tool that does so.

## An Example

The following example demonstrates how some of the above features might be used to implement the comparatively high-level configuration requirement:

> Configure two DHCP servers on every Ethernet segment. The DHCP servers should provide fixed IP addresses for all the other machines on the segment.

Conventionally, this policy might typically be implemented using a partially automated approach such as the following:

- Two appropriate machines on each ethernet segment might be identified manually.
- The lists of machines on each segment, together with their corresponding IP and MAC addresses may be extracted from some database using an ad-hoc script and massaged into the right form for the configuration files.
- The configuration files would be distributed to the appropriate machines, and the DHCP daemons started on them, possibly with the aid of some tool such as cfengine or LCFG.

Of course, there are many others ways of doing this, but this is certainly typical, and this particular example provides a simple illustration of several important configuration manipulations.

Note that most existing ‘‘configuration tools’’ would be able to handle the last operation automatically, but probably not the first. This illustrates the different between *configuration deployment* and *configuration management*. The manual operations involved in the above configuration management process make it unsuitable for an autonomic environment; if a DHCP servers fails, a new one must be selected manually. The disconnect between the independently created IP database, and the actual deployed machines is also a potential source of errors (if we decommission a machine, can we guarantee that someone will remember to remove the MAC address from the database?).

A configuration management tool must accept the high-level statement of the requirement, and translate it automatically into a form suitable for the deployment engine. For example, this might involve the following process:

- A monitoring system would provide a list of active machines on each ethernet segment.
- A constraint process would be used to select two (active) candidate machines on each segment.
- A classing mechanism would allocate a DHCP-server class to each of these machines. This class would define the appropriate parameters to configure and start the DHCP service.
- An aggregation mechanism would collect the IP/MAC mappings from the individual node configurations and make them available as part of the server configuration.

The result of this process would be a complete, unambiguous configuration specification for each machine which could then be passed to the deployment engine. This solves the two problems of the manual approach noted above; replacements will automatically be configured for failed servers, and machines which are decommissioned will automatically be removed from the DHCP configuration files.

The result of the configuration management process is clearly expressible using the simple document interface outlined earlier. Clients would be configured to run DHCP client software (with no special parameters), and servers would be configured to run DHCP servers (with a specified list of IP/MAC address pairs). The deployment engine would need to translate these simple requirements from the XML description into the appropriate configuration files and daemon operations – a conceptually simple process, but one which may involve a considerable amount of code to handle the details.

It is important to note that most of the above configuration management operations themselves are quite generic; there are many other applications of the operations such as ‘‘classing’’ and ‘‘aggregation’’ – they are not specific to the DHCP example, and can be implemented as generic operations on the simple XML structures proposed earlier.

## Conclusions

Advances in the sophistication of modern configuration tools have been hampered by the inability to decompose the problem into subtasks with well defined interfaces. If everyone who wanted to create a programming language chose their character set differently,

wrote their own editor, then created the entire toolchain, we would not see the sophistication and flexibility that we do today.

This paper proposes a simple level of interoperation which should allow independent development of higher-level configuration management tools, without the need to reinvent the wheel at each step. Such tools would benefit from the full range of available deployment engines, and separate out the maintenance burden onto more than one team.

We are sufficiently realistic to recognise that many tools, developed for internal use, will not be modified simply to meet the requirements of some external standard unless there is a significant practical benefit. We do hope, however, that the principles outlined herein will at least influence the development of new configuration tools, and promote a higher degree of interoperability in the future.

We would like to invite comment and discussion of the issues raised here on the lssconf mailing list[1].

## Acknowledgements

Many people have been involved in the development of the ideas presented in this paper, and contributed their time to explain tools, and principles. In particular, the authors are grateful to Narayan Desai (Argonne National Laboratories), Luke Kanies (Reductive), Kent Skaar (BladeLogic), John Sechrest (Alpha Omega Computer Systems), Andrew Hume (AT&T Research), and all the participants of the LISA and Edinburgh [4] Configuration workshops.

This work has been partly funded by a grant from the Joint Information System Committee (JISC).[2]

## Author Biographies

Paul Anderson is a Principal Computing Officer with the School of Informatics at Edinburgh University, where he divides his time between research projects in System Configuration and practical management of the School's computing infrastructure. He can be reached at dcspaul@inf.ed.ac.uk .

Edmund Smith worked as a researcher in system configuration on the OGSAConfig project at the University of Edinburgh, which finished in 2004. He is currently studying for a Ph.D. in Psychology at the University of Stirling. He can be reached at esmith4@inf.ed.ac.uk .

## References

[1] The GGF CDDLM working group, https://forge.gridforum.org/projects/cddlm-wg .

[2] LCFG, http://www.lcfg.org .

[3] Quattor, http://www.quattor.org .

[4] *JISC-sponsored workshop on representations of configuration data*, http://homepages.inf.ed.ac.uk/group/lssconf , April, 2005.

[5] Anderson, Paul, ''Towards a high-level machine configuration system,'' *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pp. 19-26, USENIX, Berkeley, CA, http://www.lcfg.org/doc/LISA8_Paper.pdf, 1994.

[6] Anderson, Paul, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, and Peter Toft, *Technologies for large-scale configuration management,* Technical report, The GridWeaver Project, http://www.gridweaver.org/WP1/report1.pdf, December, 2002.

[7] Anderson, Paul and Alastair Scobie, ''Large scale Linux configuration with LCFG,'' *Proceedings of the Atlanta Linux Showcase*, pp. 363-372, USENIX, Berkeley, CA, http://www.lcfg.org/doc/ALS2000.pdf, 2000.

[8] Burgess, Mark, ''Cfengine: a site configuration engine,'' *USENIX Computing Systems*, Vol. 8, Num. 3, http://www.iu.hio.no/mark/papers/paper1.pdf, 1995.

[9] Desai, Narayan, Andrew Lusk, Rick Bradshaw, and Remy Evard, *BCFG: A configuration management tool for heterogeneous environments*, Technical report, Argonne National Laboratory, 2003.

[10] Goldsack, Patrick, *Smartfrog: Configuration, ignition and management of distributed applications,* Technical report, HP Research Labs.

[11] Holgate, Matt and Will Partain, ''The Arusha project: A framework for collaborative systems administration,'' *Proceedings of the 15th Large Installations Systems Administration (LISA) Conference*, USENIX, Berkeley, CA, http://www.usenix.org/events/lisa2001/tech/full_papers/holgate/holgate.pdf, 2001.

[12] LeFebvre, William and David Snyder, ''Autoconfiguration by file construction: Configuration management with Newfig,'' *Proceedings of the 18th Large Installations Systems Administration (LISA) Conference*, pp. 93-104, USENIX, Berkeley, CA, http://www.usenix.org/publications/library/proceedings/lisa04/tech/lefebvre.html , 2005.

[1] http://homepages.inf.ed.ac.uk/group/lssconf/

[2] http://www.jisc.ac.uk/ .