Authorisation and Delegation in the Machination Configuration System

Colin Higgs – University of Edinburgh

ABSTRACT

Experience with a crudely delegated user interface to our internally developed configuration management system convinced us that delegated access to configuration systems was worth pursuing properly. This paper outlines our approach to authorising access both to individual aspects of configurations and to collections of configurations. We advocate the use of authorisation of some kind on configuration changes and we believe that the system of authorising primitive manipulations of a configuration representation outlined herein could be accommodated by a number of existing configuration systems. The authorisation system described is still experimental and we regret that real world experience of the system in use with end users is not yet available.

Introduction

There are a number of configuration management systems for computers [12, 13, 15, 6, 7, 14]. Of those, all those known to us assume that use of the configuration management system will be restricted to those people who would have been system administrators on the managed collection of computers in the absence of the configuration management system. One can usually use file-system permissions or other mechanisms (inherited ACLs operating in a similar fashion to file-system permissions in Active Directory [10] or database access controls in SMS) to allow or deny access to the configuration representations of computers, but those mechanisms are difficult to use in a structured manner, do not allow delegated control of sub-parts of machine configuration information and, with the exception of Active Directory's ACLs, are not at all designed with configuration delegation in mind.

There are a number of reasons it is desirable to delegate configuration management as much as possible:

- **Avoiding extra work and bureaucratic delay.** If a system does not delegate some control to end users, those who need something about their system changed (for example a new application added) cannot realise that change without speaking to an operator of the configuration management system. This required interaction between two people creates extra work for the operators (and probably also for the end users). Also, a certain amount of delay between request and action is inevitable, which is undesirable to the end user and usually detrimental to the operation of the business.
  The version of our tool currently in service, which does not include the authorisation features described in this paper, nonetheless allows end users to perform some basic configuration tasks via a graphical interface – namely manipulating packages and printers. From a computer population of 300 and a user population of 200, there are around 400 delegated configuration changes per month. That's 20 change requests per working day that would otherwise need an IT team member to enact. We would expect the rate of change requests to increase slightly if access was delegated to more configuration aspects.
- **End user satisfaction.** End users *like* to feel in control of their computers. A system that gives them as much control as can possibly be given is more likely to be popular.
- **Separation of expertise.** Large organisations – the ones who are most likely to require configuration management tools – have larger teams of administrators. Responsibility may be distributed among these administrators in different ways – some may be responsible for servers, with responsibilities split by server or service, others may be responsible for groups of clients, split by location. Still others may have a responsibility that covers a domain of expertise – for example, networking. It ought to be possible to support the various ways that the organisation might choose to delegate responsibility.

Of course, there are also extra complications when many people can edit configuration information.

- **Conflicts in intent.** When many people from different parts of an organisation can contribute configuration instructions, any given computer might be given *conflicting* instructions from multiple sources. There is no guarantee that those contributing to the configuration of the computer even agree on what it should do. The configuration management system must have a method of resolving such conflicts.
- **Accidental breakage.** End users are not usually, and should not have to be, experts in configuring computers – individually or en mass. This lack of expertise could easily result in

breakage, either of their own computer or further afield unless the system guards against it. An example of this would be trying to make two mutually exclusive configuration changes, or making only one change of a dependent pair.

- **Malicious intent.** If an interface into the configuration management system is provided to everyone, then either the interface or the system itself must guard against input with malicious intent.

Of the systems mentioned above, only Active Directory with its LDAP based hierarchy explicitly allows for delegation and sub-delegation of computer configuration on collections of computers. Active Directory's tree structure suffers from the deficiencies inherent in single inheritance trees outlined later, however. These have a negative impact on the way that configuration information may be organised and composed, and on the way that configuration tasks may be delegated. None of those systems allow delegation of access to *aspects* [2, 3] of the configuration – if one has access to the object representing the computer, one can change anything the configuration management system is capable of controlling.

In this paper, a method for organising and delegating access to configuration information is discussed. These ideas have been implemented in an (as yet) experimental extension to our existing configuration management system, ''Machination'' [11].

First, strategic goals and desired features and properties of the system are discussed. The paper then covers two main authorisation topics: how to represent and manipulate configuration information such that control over aspects can be authorised; and how to organise configuration information when dealing with many configuring entities and configurable objects.

Goals

With the above justification in mind, the following high level goals were set and used as guidelines to extend our existing configuration management system.

1. **Maximise delegation.** Use delegation to help improve acceptance of the configuration management system and to streamline the configuration change process. Any delegation so introduced should also have authorisation and access controls sufficient to satisfy management that the system will not be misused, where the meaning of ''misused'' is defined by management.
2. **Structure with flexibility.** The system should be compatible with with the way people and things are organised now, or most naturally – not the other way around. This requires a flexible structure with which to organise both configuration and authorisation information. Delegation should be possible to individual end-users, to experts based on their expertise, over computers based on their organisational affiliation or their purpose or their physical locations, along with a number of other criteria.

At a more detailed level, the following requirements relevant to authorisation were set for the system:

1. **Ability to authorise access to configuration aspects individually.** This is required to achieve the strategic goal of Maximising Delegation, allowing delegated access to people who could not be granted access to the whole configuration. This feature should allow one to guard against malicious intent targeted at an the configuration representation of an individual configurable object.
2. **Ability to authorise access to collections of some kind.** To effectively manage large numbers of configurable objects without large amounts of effort, the configuration systems mentioned above all have facilities to collect objects in some way before applying configuration instructions. The authorisation layer should be capable of applying to the same kinds of collections.
3. **Inheritance and/or aggregation.** It was decided early on that the configuration system should have an inheritance or aggregation structure for configuration instructions. The structure chosen is described later. Although this feature is more targeted towards the organisation of configuration instructions, it has strong implications for the way authorisation should work.
4. **Merging and conflict resolution mechanism.** The organisational structure mentioned above requires a mechanism for merging configuration information from multiple potentially conflicting sources. This mechanism has consequences for authorisation and delegation, as described later.
5. **User interface.** There is no use in giving people permission to do something if they lack the means to do it. Delegating configuration to end users requires a user interface that is suitable for end users to use.
6. **Dependency mechanism.** This is required to alleviate the accidental breakage problem described above. Machination has such a dependency mechanism, but it is not discussed further since it is only peripherally related to authorisation.

Authorising Access to Configuration Aspects

The ability to authorise access to configuration aspects individually requirement, set out above, requires the system to authorise access to individual configuration aspects. These aspects are elements of configuration information which are logically connected in some way – packages, web server configuration, network settings or the like. To facilitate this, a representation for

configuration information was sought with the following properties:

- The rules for constructing representations must allow representations that are capable of representing all required configurations!
- One should be able to collect related configuration elements together so that access to related elements can be authorised and delegated sensibly.
- One should be able to utilise a finite, and preferably small, number of primitive instructions to manipulate the representation. Authorisation rules will apply to these primitives, so each primitive should have a limited scope so as to avoid the potential for circumnavigating said authorisation.

In the rest of this section, the particular XML representation used in Machination and the primitives used to manipulate it are described.

The XML Representation

In our case the representation chosen was based on XML with the following restrictions:

1. Mixed content elements are not allowed. Every element should either contain child elements or text – not both. It is permissible for elements to contain nothing (normally carrying their information in their attributes).
2. Every element whose tag *could* appear multiple times within a given parent *must* be given an id attribute, which is *unique* amongst that element's siblings with the same tag.
3. The id attribute must only be used for the purpose outlined above.

Rules 1 and 2 together guarantee that every element in the representation can be *uniquely* referenced by an xpath of the form:

```
/tag/arrayTag[@id='id1']/tag/...
```

This is important for the configuration manipulating instructions described in the next section. In fact, since the id attribute is the only one required to address elements, in many cases xpaths are abbreviated to the form:

```
/tag/arrayTag[id1]/tag/...
```

Rule 3 is stated for completeness, though it is considered unlikely that anyone would wish to use the id attribute for anything else.

Assuming a mechanism exists for translating XML to real configuration (and it does exist) the problem of configuration management has been reduced to producing and distributing valid XML for that mechanism. Quite a number of configuration management systems take a similar approach – for example LCFG distributes configuration information via an XML representation [4], while bcfg2 configurations are specified in XML [5]. The rules described above *restrict* our XML representation to be simpler than would be allowed in plain XML, and allow it to be manipulated using a set of primitive operations (as described in the next section) which require the unique xpaths described above to address elements in the representation. It is those primitive operations to which authorisation rules are applied.

Configuration Manipulating Primitives

XML representations conforming to the rules described above may be manipulated using the following primitives:

- add_elt <element_path>
- del_elt <element_path>
- set_att <element_path> <attribute name> <value>
- del_att <element_path> <attribute_name>
- set_text <element_path> <text> NB – also erases any child elements.
- order <tag_path> <id> <first|last> or
  order <tag_path> <id1> [before|after] <id2>

All element_paths are *abbreviated xpaths* as described earlier. All primitives are *declarative* in the sense defined by LCFG [1], which means that add_elt and the like are slightly mis-named. A name more in keeping with it's function might be 'ensure_elt_exists', but this was judged a little unwieldy. It is also worthy of note that the primitives described above allow reasonably general XML to be constructed. This results in the above approach potentially having quite wide applicability.

In Machination these primitives are called *configuration instructions* and are collected together in a hierarchical structure as described later.

Authorisation of Configuration Instructions

The instruction set described above can be used to make demands about some configuration representation. They are very general – one can construct fairly arbitrary XML with them – and that they are reasonably small in number. It is to these primitives, or configuration instructions, that authorisation rules are applied.

Authorisation instructions are held in a database, and a typical instruction might be represented as

```
is_allow:    1
entities:    joe
operation:   add_elt
xml_path:    /profile/worker[packageman-1]
pattern:     <pattern>
             <constraint on="tag" type="string">package</constraint>
             <constraint on="id" type="set" set_id="unrestricted packages"/>
             </pattern>
```

Listing 1: Authorising configuration instructions.

something like that shown in Listing 1, which says that user joe[1] should be allowed to add sub-elements to /profile/ worker[packageman-1] as long as the sub-element's tag is ''package'' and its id matches the name of one of the elements of the set ''unrestricted packages.''

The schema for any eventual representation should be designed such that related pieces of configuration are bundled together in the representation. It is these related configuration pieces that form configuration *aspects*. An authorisation instruction relates to, and thus controls access to, an aspect if its xml_path attribute contains the XML path of the aspect.

The optional pattern clause allows the instruction to be more selective, subject to the content of the potential change. Tags and ids may be constrained for add_elt and del_elt, attribute names and values for set_att and so on. Current constraints allowed include string-wise equality, regular expressions, one of a provided list of strings, equal to the name (or other specified) attribute of one of the members of a set and their appropriate negatives.

Authorisation instructions control the ability of configuration instructions (primitives acting on XML) to access individual aspects of a configuration. The other promised topic for authorisation was authorising access to the structure which organises configuration information for lots of objects, which is introduced next.

[1]One can specify individual entities or sets of entities here, or a description using operators like or (require any of the pair), and (require both), and any N of some set.

Authorising Access to Aspects of Collections of Configurables

When configuring lots of objects, it is necessary to collect them together in some way. Strategies for this include database sets with no inheritance relationship (e.g., collections in Microsoft SMS), tree inheritance (e.g., OU tree in Microsoft Active Directory) and inverted tree inheritance (e.g., #include in LCFG, groups in Bcfg). SMS's unstructured sets lack an inheritance mechanism. The difference between the other two systems is similar to the difference between the *is-a* and *has-a* relationships in object oriented programming, though the distinction between the two is less clear in configuration management since 'inheritance' here is usually synonymous with accruing properties and values, and has no meaning in terms of method inheritance and method over-rides.

We now consider the implications of authorising actions on representations for these two types of inheritance.

is-a Inheritance

This kind of structure forms a tree of categories with the whole organisation at the root. The categories can be thought of as containers (as they are in Active Directory), containing configurable objects and other containers. Configuration instructions are attached to

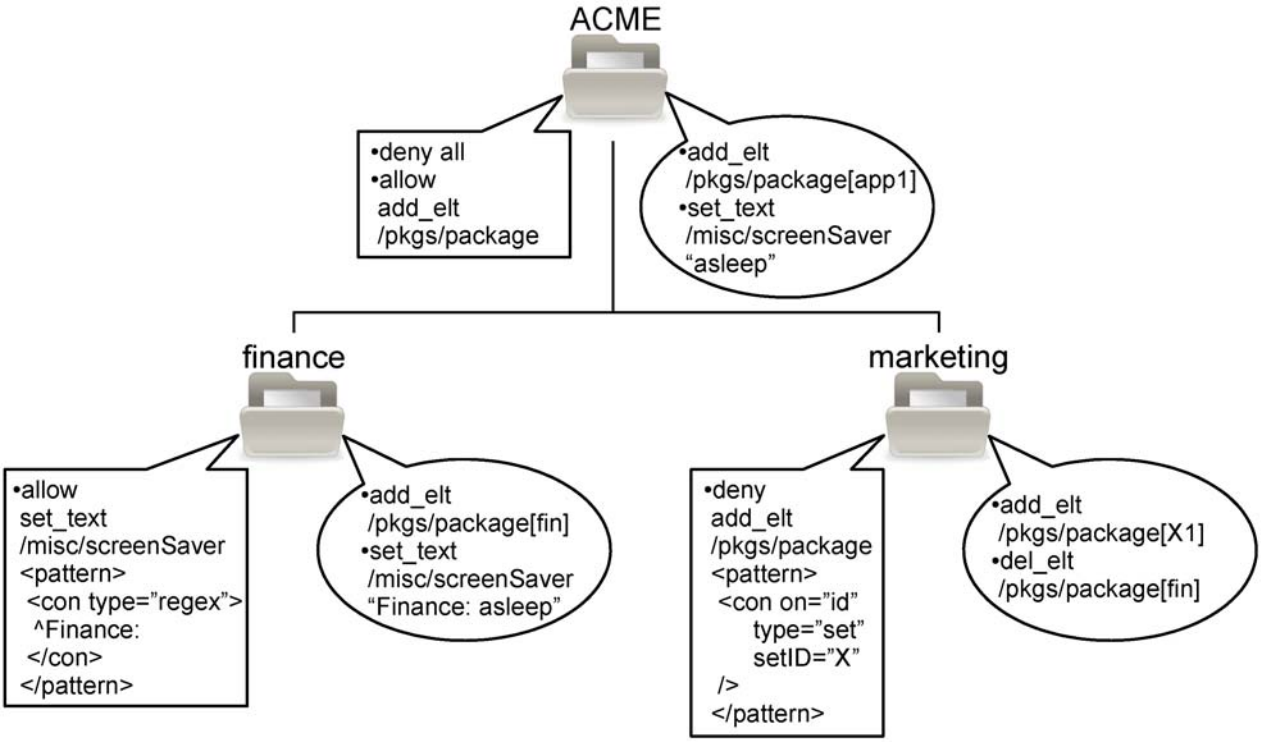

Figure 1: Is-a inheriting tree. Rectangular call-outs show authorisation instructions, round ones show configuration instructions. Containers inherit authorisation instructions from all containers above them in the tree. Some paths and instructions are abbreviated or omitted for reasons of space.

containers and apply to all configurable objects in that container or any of its sub containers.

Machination uses this style of inheritance (including multiple inheritance, as described below). Aspect authorisation instructions in Machination are attached to containers in the same way as configuration instructions, as shown in Figure 1. The authorisation instructions determine which configuration instructions are allowed to apply when the instructions are compiled into a representation, and are inherited from the root down the tree to the container being evaluated. A computer in the ''finance'' container in the tree shown would accrue instructions as shown in Table 1. Notice that the instruction to add package ''fin'' attached to the finance container is allowed because of an authorisation instruction attached to the ACME container.

has-a Inheritance

With this kind of structure, each configurable object has an associated tree rooted at that object. Each object can include or subscribe to collections of configuration instructions, and each such collection can include other collections.

For this style of inheritance aspect authorisation instructions could be included with the collections of configuration instructions as shown in Figure 2. Such authorisation instructions would determine which configuration instructions are allowed in the collection, which could be evaluated either at attachment time or at representation compile time.

The full list and order of authorisation instructions (and thus the results of the authorisation step) is order dependent, and cannot be determined from Figure 2 without further ordering information. This is explored in more detail later.

Multiple Inheritance

Either inheritance mechanism described above can support multiple inheritance. In is-a trees this is achieved by allowing objects to appear in more than one container and in has-a inverted trees by allowing

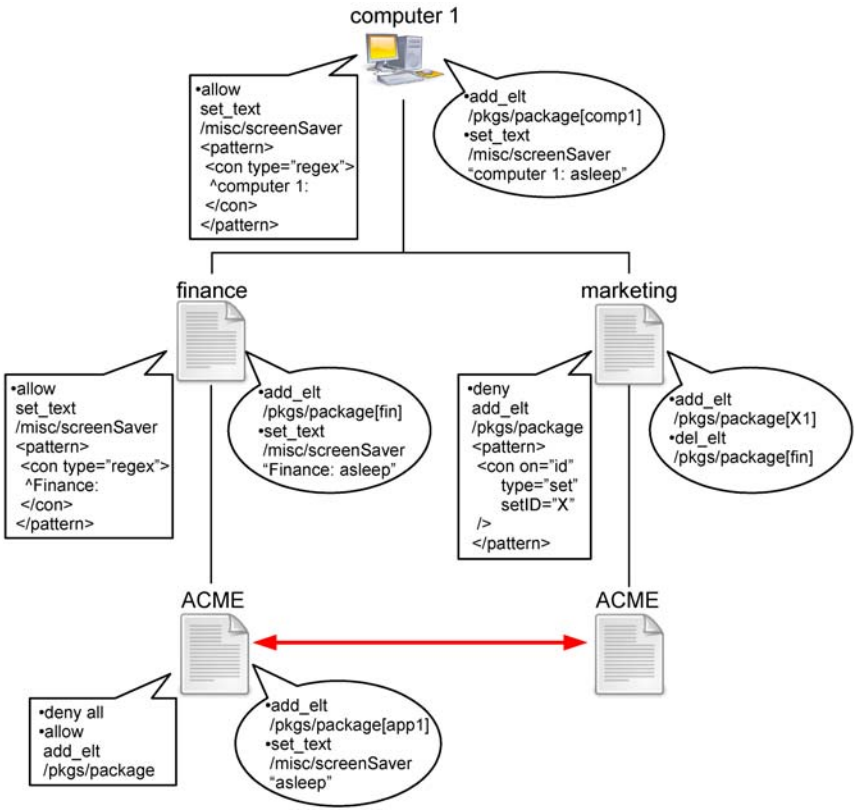

Figure 2: Has-a inheriting inverted tree. The two ACME icons represent the same file. Inheritance of authorisation instructions is order dependent, as explained later in the ''Conflicts and Merging'' section.

Source	Instruction	Allowed?	Reason
ACME	add package ''pkg1''	allowed	ACME allow packages
	set screensaver text ''asleep''	denied	ACME deny all
finance	add package ''fin''	allowed	ACME allow packages
	set screensaver text ''Finance: asleep''	allowed	finance allow screensaver

Table 1: Authorisation instructions for a ''finance'' computer.

more than one include or subscription. It was decided that Machination *should* support multiple inheritance to avoid the following two problems:

1. **Different tasks require different division criteria.** For example, consider the is-a tree in Figure 3. This would work well for configuration tasks that follow departmental boundaries, like "install finance_app1 on all finance computers," but would be rather clumsy for tasks like "make A4 paper the default on all computers in the UK," for which one would rather the computers were organised like the tree shown in Figure 4. A similar argument follows for delegation boundaries.
2. **Some objects fit more than one category.** In general, there will be objects that require configuration information for multiple reasons, and that therefore ought to be in multiple categories.
   As an example, consider the tree organised by department in Figure 3. Computers in the finance container have the finance suite of applications installed, while computers in the marketing container have the marketing suite. Now consider a computer that is used by both finance and marketing people, or by a person who has both finance and marketing roles. Such a computer needs both application suites (the union of the two sets of applications).

Figure 3: A simple tree showing ACME Corp. organised by department.

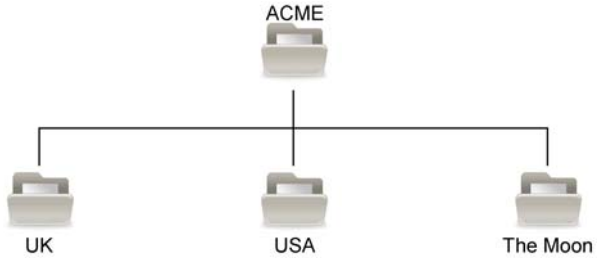

Figure 4: A simple tree showing ACME Corp. organised by location.

Conflicts and Merging

Supporting multiple inheritance immediately leads to the possibility of conflicts. Considering again the trees in Figures 1 and 2, if a computer exists in both the finance and marketing containers, or includes both finance and marketing files, it will have instructions both to add and remove the package "fin". Clearly these conflict. These configuration instructions must be merged in such a way that such conflicts are resolved.

The usual way this is resolved is to *order* the instructions. For is-a trees this means choosing a fixed evaluation order for all sibling containers. For example, the evaluation order of the finance and marketing containers in Figure 1 would need to be specified. For has-a inverted trees this means the inclusion order must be specified. In Figure 2, this would mean specifying the order in which the finance and marketing files were included, as well as whether the computer 1 instructions come before or after those inclusions and whether the ACME file is included before or after the instructions in each of the marketing and finance files.

Controlling the Hierarchy

So far the discussion has proceeded as if the hierarchy (is-a or has-a) is fixed, and the results are being computed based on placement of configurable objects and instructions within that hierarchy. The shape of the tree structure (includes or containers), the position of configuration and authorisation instructions, the position of configurable objects and the ordering of all of these are important both to the final configurations and to the nature of delegated portions. Changes to all of these need to be authorised.

It is beyond the scope of this paper to discuss authorisation of actions on the chosen hierarchy itself in any depth; however such operations fall more within the scope of the usual access control mechanisms to filesystem or directory objects. Machination's approach is to treat the hierarchy as a configurable object with a representation as defined earlier and to authorise modifications of that representation as described. Thus everything is unified under one authorisation system.

Choosing One

Both inheritance mechanisms described above have their advantages and disadvantages. The choice in Machination came from a trade-off between configuration flexibility and clarity for delegated contributors.

The has-a model is more flexible. The fact that instructions can be re-ordered on a configurable object by configurable object basis means that conflicting inclusions and instructions can be re-ordered to suit. However, such re-ordering of inclusions changes what delegated contributors are allowed to change, due to re-ordering of the associated authorisation information. This makes it less clear to contributors what they are and are not allowed to do. This and the possibility of interleaving includes, configuration instructions and authorisation instructions also make it more difficult to present such information to contributors in a graphical user interface.

The multiple inheriting is-a structure was chosen as the basis for the Machination hierarchy, in large part due to the relative ease of presenting compartmentalised views to delegated contributors.

Machination Specifics

The following features of the Machination hierarchy are more specific design choices in Machination and less relevant to the discussion on how to apply authorisation to general configuration systems. They are nonetheless important design choices with respect to the way that delegated portions of the Machination hierarchy interact.

Merge Policies

As mentioned earlier, choosing is-a style inheritance leaves Machination with a less flexible structure. We gain some flexibility back by introducing the concept of *merge policies* which may be attached to containers, and are applied when configuration instructions from multiple sibling containers are merged.

These specify the precedence of instructions between sibling containers depending on the contents of the instruction. Currently, the only information one can use is the element path to be altered, though this may be expanded to include details such as the attribute or text value being set. As an example, one could specify that the tree under the by network merge point should have precedence over the firewall area of a Machination Windows profile by attaching a merge policy of the form:

```
for xml_path /profile/worker[firewall-1] \
          local wins
```

where, similarly to the authorisation instruction shown earlier, the policy is stored as discrete data values in the hierarchy, rather than as a written command.

If there is a clash in merge policies (for example if two sibling containers' policies both claim precedence over one path in the profile) the siblings order in the parent is used to resolve precedence, just as if the policy statement did not exist.

Combining merge policies with authorisation instructions can be useful when delegating expertise specific configuration tasks. For example, suppose that an organisation has a networking expert. This expert should make sure that firewall rules are in place on each client appropriate for the client's network and status. We can set up a merge point called by network and delegate full control of the substructure to the expert (who will have a good idea of how to structure it for best results). The merge policy should be set to give precedence to that container for firewall rules, but to cede precedence for everything else and the authorisation instructions should allow configuration of only the firewall section of the profile. The expert can now organise computers into containers as required, perhaps resulting in a tree like that shown in Figure 5.

Merge Points

Multiple inheritance can be a dangerous thing. As a configurable object inherits configuration information from more paths it is more likely that conflicts will result and it becomes more difficult to follow and visualise a configuration back to its sources. Some temperance is required. As an aide to this (or perhaps an enforcement of it), Machination only allows configurable objects to be placed in two containers if they are separated by a *merge point*. For example, in Figure 5, a computer could be placed in both the low risk and subnet A containers, but not in both the low risk and high risk containers.

Merge points essentially break the multiply inheriting tree up into multiple singly inheriting trees, which are much simpler. Merge points should be inserted wherever one or both of the limitations of singly inheriting trees appears (i.e., different division

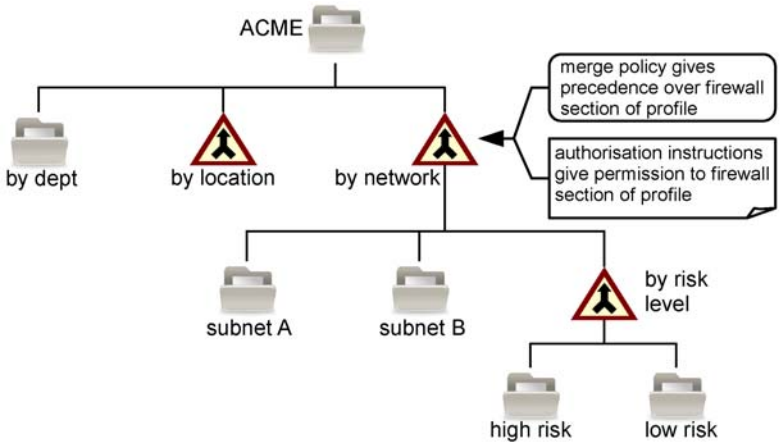

Figure 5: A networking expert has created a sub-tree in an area delegated for that purpose. The merge policy attached to "by network" gives this portion of the tree precedence over its siblings for firewall rules, which would otherwise normally take precedence due to the way they are ordered. Three containers ("by location", "by network", and "by risk level") have been nominated as merge points, which determine how configurable objects may be placed in the tree.

criteria or multiple categories are required). The idea is to use as many merge points as one needs, but as few as one can get away with.

Conclusions and Future Work

We have the beginnings of a system we hope will allow us to reduce our configuration workload at the same time as making our users and our managers happy. Our hope is based on experience with a useful but deficient system and underpinned by some new authorisation features. We are confident we will be able to delegate access to configuration aspects to desktop and laptop systems' end users, as well as more sophisticated forms of delegation with respect to collections of computers.

We believe that the authorisation work presented here, particularly the work on authorising access to individual configuration aspects, is transportable to other configuration systems and would be pleased to see others consider it. To this end, the Machination project is being open sourced, and code, or a link to it, should be available at the Machination project page [11] by the time this paper is published.

There is much work left to do, both on Machination and investigating more general consequences of authorising configurations. Some we have identified in the following.

- **Applicability to other configuration management systems** This paper describes a representation designed for configuration information and some primitives to build such representations which may be authorised. It goes some way toward describing how those configuration and authorisation instructions can be collected in the structures commonly used in other configuration systems before focusing on the structure used in Machination.
  From this follow two broad areas for possible investigation: could the given representation or another, better one be used to provide authorised access to configuration aspects across configuration management systems; and could the work on applying the authorisation instructions within other types of hierarchy be taken to the point where it is usable?
- **User interface** As stated in the section on goals, it is a requirement for us that the system be manipulated by a user interface which is suitable for use by a broad spectrum of users: possibly unskilled end users, computing professionals with expertise in some domain other than configuration management, and configuration experts with a good overview of the whole configuration system. Such an interface is currently under heavy development and not many details are available at this time, but the following are seen as requirements:
  - Unsophisticated use involving configuration of only one computer should be possible and easy. In most cases, this should involve picking things that the computer should "have" (like packages) from a list of available options. This mimics the current interface used to configure Machination computers.
  - Authorisation rules should be capable of allowing our common case of "you can configure the computer you are sitting at."
  - No one should be editing XML – not for configuration instructions, authorisation instructions or anything else. The XML is there for the computers to communicate with each other and a friendlier interface should be provided.
  - Sophisticated users should be able to view how given instructions will affect computers throughout the tree, how given computers inherit their configuration, where any conflicts are and a number of other types of information which span the whole or part of the tree. It is likely this information will be conveyed using toggled overlays.
  - Authorised contributors should easily be able to determine what they have control over.
- **Applicability to higher order configuration management** A matter of ongoing research in configuration management is how to raise the level of instruction from statements like "add package A" or (as a set of instructions) "install and configure a print service" to statements like "ensure there are two DHCP servers on every subnet" [9]. In a recent paper [8], Alva Couch also suggested that the semantic level of configuration management should be adjusted, such that configuration management systems reason more about the *meaning* of configuration instructions, rather than the eventually delivered content. This paper deals with authorisation at a more primitive level than either of these, and further investigation would be required to determine whether the authorisation schemes outlined in this paper could be relevant.
- **Building other representations** The representation rules outlined allow fairly general XML to be generated. The main restriction being the lack of mixed content elements. from this follow two areas of investigation: what things might one want to represent that cannot be represented without mixed content elements; and what other kinds of XML representation might a hierarchy like this usefully construct. On the later front, it has already been mentioned that Machination represents (and authorises actions on) its own state in this way. In fact the Machination hierarchy gives all configuration and authorisation instructions a service identifier,

which identifies which XML representation the instruction targets, and which can be used to keep separate several target representations.

Author Biography

Colin Higgs graduated from the University of Edinburgh with an Honours degree in Mathematical Physics. After a few years working as a physicist, Colin became the sole system administrator for the department of Chemical Engineering back at the University of Edniburgh. Several mergers and re-organisations later, he is now part of a larger team for the School of Engineering and Electronics, where he at least tries to work toward the ''Bahamas'' model of computing support.

Bibliography

[1] Anderson, Paul, *A Declarative Approach to the Specification of Large-Scale System Configurations*, 2001, http://www.dcs.ed.ac.uk/~paul/publications/conflang.pdf .

[2] Anderson, Paul, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, Jim Paterson, and Peter Toft, *Gridweaver Project Report D3.1: Large- Scale System Configuration with lcfg and smartfrog*, 2003, http://www.gridweaver.org/WP3/report3_1.pdf .

[3] Anderson, Paul, and Alva Couch, *LISA 2004 invited talk: What is This Thing Called System Configuration?*, 2004, http://www.usenix.org/publications/library/proceedings/lisa04/tech/talks/ couch.pdf .

[4] Anderson, Paul and Alastair Scobie, ''LCFG: The Next Generation,'' *UKUUG Winter Conference*, UKUUG, 2002, http://www.lcfg.org/doc/ukuug2002.pdf .

[5] *bcfg2 Documentation: Writing Specifications*, http://trac.mcs.anl.gov/projects/bcfg2/wiki/WritingSpecification .

[6] *bcfg2 Home Page*, http://trac.mcs.anl.gov/projects/bcfg2 .

[7] *cfengine Home Page*, http://www.cfengine.org/ .

[8] Couch, Alva, ''From x=1 to (setf x 1): What Does Configuration Management Mean?'' *USENIX ;login:*, Vol. 33, Num. 1, pp. 12-18, 2008.

[9] Delaet, Thomas and Wouter Joosen, ''Podim: A Language for High-Level Configuration Management,'' *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, pp. 261-273, 2007.

[10] Iseminger, David, *Active Directory Services for Microsoft Windows 2000 Technical Reference*, pp. 28, 45, Microsoft Press, 2000.

[11] *Machination Home Page*, http://www.see.ed.ac.uk/machination .

[12] *Active Directory Home Page*, http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/default.mspx .

[13] *Microsoft Systems Management Server Home Page*, http://www.microsoft.com/smserver/default.mspx .

[14] *Reductive Labs Puppet Project Page*, http://reductivelabs.com/projects/puppet/ .

[15] *LCFG Home Page*, http://www.lcfg.org .