
A Declarative Approach to the Specification of Large-Scale System Configurations



by Paul Anderson <paul@dcs.ed.ac.uk>

DICE Computing Environment Project
Division of Informatics
University of Edinburgh

1 Abstract

Many tools are available for assisting in the configuration of large numbers of inter-related computer systems, but there has been little work on languages which provide a convenient high-level way to specify and manage the complexity of these configurations. Building on experience with existing configuration systems and other languages, this paper explores the possibility of developing a largely declarative specification language for describing the configuration of large numbers of related machines. Some implementation issues are also discussed.

2 Introduction

When managing large collections of computer systems, it is essential to have some automatic tool for installing and configuring operating systems and other common application software. This usually involves a common set of software packages, and a machine-specific set of configuration parameters (the *profile*). A configuration tool should be able to install a new machine automatically by using the profile to determine which software packages should be loaded, and how they should be configured¹. The following operations can then be performed easily and quickly, without requiring specialist staff:

- ❑ New machines can be installed automatically according to some predefined profile.
- ❑ Failed machines can be recreated identically on new hardware, even if the configuration is complex and unique.

¹Copies of both the software packages, and the profiles must be stored safely away from the machine they are being used to configure.

A Word About Terminology

Existing configuration systems and the associated literature use a wide range of different terminology for similar concepts; even when referring to the same tool, different terms have been used for the same concept at different times. A number of the papers reviewed here also attach a very specific meaning to common words such as “feature” or “aspect”. Where terms are used with a specific meaning, they are printed in *italics* and listed in the glossary.

- ❑ Machines can be rebuilt instantly if there is any doubt about the validity of configuration.
- ❑ Existing machines can be *cloned* to produce similar machines, simply by copying the profile.
- ❑ Only one system backup is required for all machines.

A *dynamic* configuration tool (as opposed to a *static* tool) is one which can regularly monitor the machine configuration and adjust it to match the profile. This has the following advantages:

- ❑ If the profile changes, then the configuration will be updated automatically. Profiles change continuously at most sites, since an individual machine configuration depends on the configuration of related machines and network parameters.
- ❑ If the machine configuration becomes damaged (either accidentally, or maliciously), then it will be automatically repaired (this process has been compared to immunology [Bur98a, Bur98b]).

Providing that a dynamic configuration tool is running, then the machine configuration is guaranteed to match the specification. With a static tool, the machine configuration *rots* and it is impossible to have any confidence in important properties such as the security of the system.

For a large installation, it is clearly impractical to maintain individual profiles for each machine. Most configuration tools provide some way of using a higher-level description (*HLD*) to specify the configuration of whole collections of machines. Typically, the HLD will contain conditional code to generate variations of the profile for different classes of machine. The focus of this paper is to investigate the requirements and difficulties of such an HLD and, in particular, the possibly of a more declarative approach to the specification.

It is useful to view a configuration tool as a compiler which translates the HLD into a set of individual machine configurations. The input to this compiler is high-level specification which describes the configuration of various aspects and classes of machine; this is discussed in section 4. The output from the compiler is a set of individual machine profiles which are discussed in section 3. Section 5 compares some existing configuration tools, classifying them according to the concepts developed in previous sections. Section 6 develops the concept of a declarative language, and section 7 looks at developments in other languages which may be relevant. Finally, section 8 considers some implementation issues.

3 Machine Configurations

Most operating systems expect the configuration of a machine to be defined by numerous parameters which are scattered across many different configuration files and databases on the machine's local filesystem. When discussing configuration languages, it is useful to have a more explicit representation of this profile; even if the explicit representation is not supported by a particular implementation, it provides a convenient, formal definition of the elements that the language is intended to describe.

3.1 Profile Representation

In practice, the complete configuration information can be represented quite easily as a list of key-value pairs (*resources*) as demonstrated by systems such as LCFG [5.2]. However, it is useful to have a small amount of structure, including support for explicit lists and sub-elements; XML appears well-suited to this application and has been used in practice by systems such as Arusha[5.4] and GConf[Pen]). Figure 1 shows how a typical profile fragment might be represented.

This represents a machine with two disks, the first disk having two partitions with the specified sizes and mount points, and the second having a single partition. Experience with LCFG has shown that a typical site

Figure 1: A Profile in XML

```
<disks>
  <disk>
    <partitions>
      <partition size="1024"
                mount="/" />
      <partition size="free"
                mount="/usr" />
    </partitions>
  </disk>
  <disk>
    <partitions>
      <partition size="free"
                mount="/home" />
    </partitions>
  </disk>
</disks>
```

may require about 2000 different resources [Sco01], although only about 10% of those will need to be set to non-default values for any individual machine.

The “level” of the information in the profile should probably be quite low; i.e. the resources should correspond quite closely to the real configuration parameters required by the operating system. It may be useful to represent some parameters at a slightly more abstract level, so that the same resources would be applicable to slightly different operating systems, for example. However, this should normally be handled by the compiler, and defining profile parameters at too high a level loses the ability to specify machine-specific details in the configuration.

For similar reasons, the structure of the profile should probably be oriented towards the operating system configuration files, rather than any high-level configuration structure. This makes it easier to implement, since sections of the profile correspond directly to the software modules responsible for configuration of the various different subsystems. As we will discuss later (section 6), it is also rare that there is any one single high-level hierarchy which is appropriate.

3.2 Profile Implementation

Many configuration tools simply manipulate the operating system configuration files directly according to the HLD, and never construct an explicit profile (for example, Cfengine [5.1]). If an explicit profile is generated, then software modules (*components*) are

required to translate between the profile and the operating system configuration files. Normally, a one-way translation is sufficient (as in LCFG [5.2], since the configuration tool provides the definitive resources; however, in some cases a two-way translation is provided so that profiles can be recreated from operating system configurations (for example, see COAS [5.5]).

Those tools which do construct an explicit profile separate the (machine-dependent) tasks of implementing the profile from the (machine-independent) task of generating the profiles from the HLD. This has the following advantages:

- ❑ It is easier to understand and validate individual configurations.
- ❑ It is possible to reason about the entire system by considering the set of profiles which describe the whole site in a declarative way; for example, it is possible to answer questions such as “how many machines use server X as their DNS server”.
- ❑ The system is easier to extend and port, since this involves only modifications to the components, and profile schema, not the compiler.
- ❑ It is possible to have multiple languages and views on the configuration by compiling the profiles from different languages²

Note however, that the format of any explicit profile, and even its existence in a particular implementation is not essential to the following discussions of the HLD.

4 High-Level Descriptions

A typical organisation may have hundreds or thousands of individual machines, each requiring a profile with hundreds or thousands of configuration parameters. In most cases, it will be unrealistic to assume that any one individual (or group) will be fully responsible for all aspects of this configuration. In practice, a more devolved management approach is necessary and many different people at different levels, may need input to the configuration process:

- ❑ Junior technical staff may want to create new machines by specifying their configurations in simple ways, such as creating many machines similar to some existing machine, or creating a machine as a “web server”.

²Multiple simultaneous views of the same data require the ability to translate in both directions which may not be possible.

- ❑ More senior administrators may want to create more complex specific configurations, as well as the “templates” that are used by junior staff to specify the default behaviour of whole classes of machines.
- ❑ Server and network administrators will want to change configuration parameters that have a global effect on many different systems, such as changing the network servers for a whole class of machines.
- ❑ Different development staff will want to create and maintain new configuration components and their associated schema, for services for which they are responsible.
- ❑ End-users may want to modify some aspects of their machine’s behaviour (if permitted by the system administrator), such as the particular set of application packages loaded.

A HLD language therefore needs to be sufficiently modular to allow these people to control the configuration aspects for which they are responsible without disturbing, or necessarily understanding, unrelated aspects. In some cases, it may be necessary to enforce this separation with access controls.

4.1 Procedural Descriptions

Many configuration tools use a procedural approach to describing groups of configurations; in the basic form, a single script-like specification defines various resources with conditional code to select alternatives depending on some property of a particular machine, as shown in figure 2.

Purely procedural code is clearly impractical, since the addition of a single new machine involves additions to the conditionals spread throughout the entire specification. Most such tools augment the code with some form of *class*, so that machines can be allocated to classes in a declarative way (for example, [Bur95, Mic97, Phi]) and the conditionals can test for class membership. Class membership may also be defined implicitly, such as the Sun class in the above example.

As with conventional code, this type of specification can be written in such a way that new components can be developed in a modular way. However, changes to the specification at any of the above levels are deeply intertwined, and this approach does not scale well to the devolved management of large numbers of diverse systems.

Figure 2: Procedural Specification

```

...
some common resources
...
if this is machine X
    R1 = A
else
    R1 = B
fi
if this machine is a Sun
    R2 = C
else
    R2 = D
fi

```

4.2 Declarative Descriptions

By listing the resources specific to each class, it is possible to define configurations in a more modular, declarative way; for example, the above configuration can be expressed as shown in figure 3.

Figure 3: Declarative Specification

```

default:
    some common resources
    R1 = B
    R2 = D

sun:
    include default
    R2 = C

machine X:
    include sun
    R1 = A

```

In this case, different aspects of the configuration are isolated into separate *prototypes* which can be managed by different people (the “maintainer” of machine X, and the local “Sun expert”). Notice that resources defined explicitly within a prototype *override* the corresponding values supplied by any included prototype. Systems such as LCFG [5.2] and Arusha [5.4] have demonstrated that this is a feasible approach, but there are a number of difficult issues when using this as the basis for a large practical system. Section 6 explores

this in more detail.

5 Existing Configuration Tools

The system administration community has created numerous tools for system configuration; [And97] provides references to a typical selection. Many of these deal with configuration at the file level and do not attempt to abstract the configuration information into a centrally-manageable format, and very few systems concentrate on the structure and representation of entire site configurations. The following selection of tools are presented because they illustrate some of the previously defined concepts; they are not intended to be particularly representative, or recommended.³

5.1 Cfengine

Cfengine [Bur95] is a very popular large-scale system configuration tool. The configuration specification is largely procedural, defining actions on the basis of classes. These actions operate directly on the operating-system configuration, without an intermediate profile, and the specification language includes a library of useful functions for operations such as editing configuration files and manipulating symbolic links. These operations have the useful property that the current state of the machine is examined and changes are only made when necessary, making it ideal for use as a dynamic system.

In the way that Cfengine is normally used, it is difficult to specify large, complex configurations in a declarative way, but the well developed functions for manipulating configuration files may make it a useful tool for implementing configuration components.

5.2 LCFG

LCFG [And94, AS00] is a configuration tool developed in the Division of Informatics at the University of Edinburgh. An explicit profile (the *resource map*) is distributed to clients as an NIS map. Components called *objects* (or *subsystems*) implement the profile using shell script. The (largely) declarative HLD is translated into the resource maps using a combination of Perl code and the C preprocessor.

Successful experience with the declarative specifications used in LCFG has motivated the investigation described in section 6.

³Note that most of the examples used in this paper refer to freely available Unix tools, although the concepts apply equally well to other platforms, such as Microsoft Windows.

5.3 SUE

SUE [CERa, CERc, CERb] is a configuration system developed at CERN which is partly procedural and partly declarative; components called *features* respond to *methods* for configuring and controlling particular aspects of a system. The configuration for a particular machine is composed by selecting a set of features (known as a *profile*). This provides a declarative specification, but only at a very coarse granularity. More fine grained configuration is performed by the procedural code in the features. An update process runs regularly to maintain the physical configuration in line with the specification. As with Cfengine, *SUE* provides a useful library of functions for testing the current state of the system and making only those changes that are necessary.

5.4 Arusha

The *Arusha* Project (ARK) [Par] is an interesting framework for “collaborative systems administration”. Declarative configuration information is stored in XML descriptions which are created by composition of several prototypes. A hierarchy of prototypes can be used to share specifications between sites while allowing individual sites (or groups, or individual machines) to override certain parameters as required. Practical use of the tool has focused mainly on software package configuration so far, however it has been applied to host configuration and even documentation creation.

The XML configuration specifications contain attributes and methods (Python code). The prototype composition algorithm is based on priority which is well-suited to the *Arusha* philosophy, since “more local” parameters take precedence over inherited parameters. However, this is not sufficient for resolving conflicts between prototypes specifying different aspects (see section 6).

5.5 COAS

COAS [Cal] is a configuration system being developed by the Linux community, lead by Caldera. *COAS* uses an explicit profile representation to provide the opportunity for alternative specification and editing tools. Modules known as *CLAMs* define explicit schema for elements of the profile and include *mapper* code which can perform a two-way translation between the *profile* and the system configuration files.

This two-way translation ability allows *COAS* to use the system configuration files as the definitive copy of the configuration information. This is very useful for individual systems which may be hand-configured us-

ing other tools, but it is not appropriate for generating configurations from higher-level descriptions. Unfortunately, *COAS* does not really address this issue.

5.6 Alchemist

Alchemist [Red] is a new configuration tool from Redhat. At the time of writing, this is very new and there is very little documentation. However, the description reads:

“The alchemist is a back-end configuration architecture, which provides multi-sourced configuration at the data level, postponing translation to native format until the last stage. It uses XML as an intermediary data encoding, and can be extended to arbitrarily large configuration scenarios.”

6 A Declarative Approach

This section develops the idea introduced in section 4.2 of using prototypes as the basis for a declarative specification language. This appears to produce clearer, more modular specifications than the procedural equivalent, however it is not always obvious how to express some of the constructs that can be easily specified using a procedural language (especially where this allows access to the power of a general-purpose programming language). The following discussion explores the extent to which the purely declarative approach is possible, and discuss the various ways in which procedural code may be incorporated where necessary. There are several distinct ways in which this is typically used, and these should probably be treated differently. The experience of the person expected to author, or understand each particular code application is also considered; this may be the *component* author, the *prototype* author, or the end user of the configuration system.

6.1 Schema

The complete machine profile consists of elements corresponding to all the different configuration *components*. The *schema* for the entire profile is defined by union of the schema for these separate elements. In the previous example (figure 3), each prototype is allowed to define values for any resource in the profile, and hence all the prototypes have the same schema.

In a complex system, it would be useful to define other types of object (such as disks) so that we could create prototypes for these objects and re-use them as sub-elements of different machine configurations. However, this introduces extra difficulties

when combining prototypes containing sub-elements (see [And01b]) and it is not clear whether the extra complexity is worthwhile.

6.2 Prototype Combining

The declarative machine specification consists of a set of prototypes and explicit, machine-specific resources (see figure 3). Each prototype includes resources (and references to other prototypes) defining the configuration for some aspect of the system, such as:

- ❑ A prototype specifying parameters relevant to the particular hardware configuration.
- ❑ A prototype specifying parameters relating to some particular function of the machine (such as a Web server).
- ❑ A prototype specifying the the machine's connection to the network.
- ❑ A prototype specifying a particular security policy.

Figure 4 shows some typical prototypes from a current LCFG implementation.

Although most of these these are logically separate aspects, and may well be managed by different people, there will frequently be conflicts where several prototypes affect the same physical parameter. In a procedural specification language, the ordering of the conditionals gives priority to the properties of certain classes (see figure 2), and in Arusha, the resource values from the most “local” prototype *override* those from included prototypes. If prototypes are to be used extensively to represent fine-grained, overlapping aspects, then the resolution of these conflicts may sometimes require more than this simple prioritisation; for example, one configuration parameter might specify a list of features which is intended to be a composite of the lists specified by the included prototypes. The required semantics for merging the lists may be different in different cases (see [And01b] for a more detailed discussion), and may sometimes require explicit procedural code (see 6.3)⁴. This process of combining prototypes has some similarities with *aspect-oriented programming* (section 7.2), hence the use of the term *aspect*.

⁴This makes it important to be able to generate a meaningful description of how a particular resource has been derived from the prototypes (the *derivation* or *provenance*).

6.3 Mutation

Permitting arbitrary code in the prototypes provides an easy solution to the problems of combining, since each prototype can define code to specify how values from included prototypes should be handled. This is called *mutation* in LCFG. It is not clear that this can always be avoided; for example, two prototypes may specify the allocation of additional disk space on a particular partition, and the resulting disk size should be the sum of these allocations. Problems such as this are hard to solve without arbitrary code, but this is genuinely needed in only a small proportion of cases, and overuse can lead to unreadable and unreliable specifications.

6.4 Component Methods

In many cases, it is possible to avoid procedural code in the specification by defining declarative resources in the specification and using procedural code in a component method. This is preferable because the person responsible for the code is now the (presumably more experienced) component author, rather than the specification author, or end-user. The ability to reason about the specification is also retained. Arusha stores the procedural code for the component methods as resources in the profile. This has the advantage of avoiding version mismatches between the components present on a machine, and the the resource data that they process. It also allows component code to be easily updated and changed on a per-machine basis during development.

6.5 Validation

Prototype schema need to be able to include arbitrary code to validate the resources. This is very similar to a component method, except that it is insufficient to simply store this code as part of a component, because it must be possible to evaluate it at “compile time” (ie. when the prototype composition takes place, rather than when the profile is implemented). The validation code is *spectative* (does not change the configuration, only examines it). Like a component method, it is also the responsibility of the component author.

6.6 Variables

It is often useful to define the value of one resource in terms of some other resource, or to define several resources in terms of some common value. For example, the “owner” of a machine might be defined and used as the value of several different resources. LCFG provides a simple mechanism for defining global symbols by using the C preprocessor. Arusha and most procedural languages provide support for variables (often

Figure 4: Some Typical Prototypes

macros	-	Common definitions
linux_rh62	-	Redhat 6.2 operating system
linuxdef	-	Linux operating system
wireh	-	Use the services on Ethernet segment H
vmware	-	Support the VMware virtual machine
portable	-	Configure as a portable
irda	-	Support Infrared
localij	-	Support a local inkjet printer
ipchains	-	Configure network security for laptop
toshiba_4300	-	Toshiba 4300 parameters
hardware	-	Common hardware parameters
xntpd	-	Do time synchronisation
dns	-	DNS defaults
removeable	-	Allow user to handle removable media

inherited from the host programming language). However, defining variables within prototypes is difficult because of the evaluation order. It would be useful to have a more integrated variable semantics but it is not clear how the scope and namespaces should be handled when combining prototypes.

6.6.1 Dependencies: The above techniques allow much of the potential complexity implied by procedural code in the specification to be avoided. However, there will always be some cases where a resource depends on other resources in a way which is difficult to express in any other way; for example, it might be necessary to use one of several different partitioning schemes, depending on the size of a disk.

As well as dependencies between the resources of a single machine, there are also cases where dependencies exist between resources on different machines; for example, an NFS file server may require a resource specifying the clients that are permitted to connect to the server. Ideally, this list would be generated by collating the resources from the client machines which specify this machine as their server, but this is not currently supported in either Arusha, or LCFG. It is also useful to be able to generate some system-wide configuration tables from the set of machine specifications; for example, a list of IP addresses for the DNS, or a list of print servers. LCFG currently supports this in an ad-hoc way, but it could be viewed as a special case of the inter-machine dependency problem.

6.6.2 Schemes: In some cases, it is useful to have several variations of a machine configuration which ap-

ply in different circumstances (*schemes*). For example:

- A range of different parameters are likely to be needed depending on whether a laptop is connected to its home Ethernet, or via a remote dial-up.
- Different system configurations, might be desirable depending on the class of user at the console.

These could be solved by the use of procedural code, but this approach has one very undesirable property; it requires conditionals which depends on variables whose values are only available on the target machine, at runtime. Further, the configuration changes are likely to take place while the machine disconnected from the network. This means that we cannot evaluate the configuration for validation and consistency checking, unless all the possible values are known beforehand. There seems to be a strong case for disallowing code such as this, except in a very controlled way.

6.7 Access Control

In certain cases, it is useful to declare a resource value in a prototype as *immutable*. This is useful where some feature is essential to the configuration aspect defined by a particular prototype; if some other prototype attempts to override this with a conflicting value, then an error can be generated at compile time.

Extending this concept to assign *ACLs* to individual resources (or entire prototypes) provides *aspect-based access control*; figure 5 shows the creation of a `filter` prototype that defines which resources the user

John is permitted to change for a particular machine (managed Jane).

Figure 5: Access Control

```

default:  owned by Jane
  R1 = A (ACL=jane)
  R2 = B (ACL=jane)
  R3 = C (ACL=none)  immutable

filter:  owned by Jane
  R1 (ACL=john, jane)

user:    owned by John
  R1 = X ✓ ok
  R2 = Y ✗ error

machine: owned by Jane
  include default
  include filter
  include user

```

7 Related Work

Several concepts from programming language research appear to be relevant to the discussion in the previous section. In particular, *prototype-based languages* and *aspect-oriented programming*. However, a largely declarative configuration language is quite different from the programming languages usually addressed by this work, and not all of the concepts translate in a useful way.

7.1 Prototype-Based Languages

Several object-oriented programming languages have been developed which rely on the principle of copying prototype instances instead of creating new instances of a class [Bor86]. This work is motivated by the desire to simplify the creation and management of datatypes involved in object-oriented programming (for example, [Lal89]). Hence, discussions of multiple inheritance concentrate on the datatype of the resulting object, rather than any combining of data values, and this is not particularly appropriate to the system configuration problem. However, there are some interesting discussions of namespaces [UCCH91], and a number of other issues; for example, should prototypes be allowed to dynamically change their inheritance hierarchy; for example by specifying some variable or proce-

dural code? (this might be a useful way to encapsulate schemes (section 6.6.2).

7.2 Aspect-Based Languages

Aspect-oriented programming [KLM⁺97, Xera] is an attempt to address the “tyranny of the dominant decomposition”:

“Existing artifact formalisms generally provide only one “dominant” dimension along which concerns can be separated ... These concerns often overlap and interact with one another ...” [OT99b].

This problem is very close to the problem of combining prototypes discussed in section 6.2; however the data in the profile is structured, there will be prototypes for certain aspects which affect parameters that are scattered throughout this structure, and there will be parameters which are affected by more than one prototype.

Most aspect-oriented work relates to object-oriented languages; typically, different aspects of a problem are specified in an extension of some existing language and a pre-compiler such as AspectJ [Xerb] is used to *weave* these into a Java program. *Aspect-oriented process engineering (ASOPE)* [OT99a] is one example where these techniques have been applied to something other than a programming language; in this case, a business process description. Recent work on “multi-dimensional separation of concerns” [OT99b, IBMa] appears worth further investigation; IBM have used this to produce a tool for matching and reconciling XML documents [IBMb].

7.3 Configuration Management

In the literature, the term *configuration management* usually refers to configuration of large software systems. This involves constructing a version of a software package from appropriate components and managing the changes in those components (for example, [HHW98]). This does not appear to have anything particularly relevant to the system configuration problem. However, it might be interesting to follow up the analogy between the application of “change sets” to a software package, and the specification of a machine configuration as sets of variations from some default prototype; both of these have the requirement to resolve conflicting differences. Recent work [Men] has attempted to specify the changes as syntactic or semantic merge operations, rather than simple textual differences.

8 Implementation Issues

The proposed declarative language design supports de-veloped configuration management, in the sense that different aspects are isolated so that they can easily be managed by different people. This needs to be supported by the implementation which must provide distributed storage and access control for the prototypes, since different prototypes may be managed by different groups within an organisation. The implementation also needs to provide robust mechanisms for distributing configuration data to client machines, including support for *disconnected operation*; this is important for managing laptops, and for providing robustness against network failures.

8.1 Creating and Editing Configurations

As noted in section 4, a wide range of people with different levels of experience will be involved in creating and editing machine configurations and prototypes. In theory, it should be possible to provide different languages, customised to particular needs, but if the same data is to be manipulated using different languages, then the definitive copy of the data must be stored in a common form which can be translated in both directions between all the languages being used, and this is probably not practical. However, the modular nature of the proposed declarative language makes it suitable for use at various different levels:⁵

- Developers can write independent component code and schema to a standard API, in some programming language.
- Senior system administrators can develop profiles for particular classes of machine, or particular aspects of the system.
- Junior system administrators can create standard machine configurations by composing prototypes, with access control at the prototype level restricting the possibility for serious errors.
- End-users could use a GUI interface to select pre-defined prototypes and resource values.

8.2 Compilation & Validation

To generate a particular machine profile, the compiler need to request all the included prototypes (possibly from remote machines). If the compilation does not take place on the client itself, then the resulting profile needs to be transmitted to the client. This leads to a

⁵[And01a] shows some examples of a possible concrete syntax.

number of problems: (1) it is impossible to change the configuration of machines while they are disconnected, and (2) it is difficult to support *schemes* where the compilation process is affected by variables whose values are only known at runtime, on the client machine.

However, if the compilation takes place on the client, then validation of the profile is not possible at the time the prototypes are created. Since there will be no central repository of profiles, it is also impossible to reason about the site configuration from the complete set of profiles. For these reasons, compilation on the server is probably preferable. In those cases where it is necessary to change the configuration of a disconnected machine (usually during development), then the machine may be able to run a local configuration server process. If the set of schemes is small, then it should also be possible to provide support by generating multiple profiles (one per scheme), or by permitting some very controlled conditional code in the profile.

One major practical problem with the compilation process is the tracking of dependencies between prototypes. Changing a single prototype may imply changes in hundreds or thousands of machine profiles, and it would be very useful to know when these changes invalidate some dependent profile. This involves locating, compiling, and validating all dependents, preferably before committing the changes to the prototype. Locating all the dependents is not straightforward, since profiles may be exported to other servers and used to configure their clients. Recompilation and validation on a single server is also likely to take a considerable time.

When a profile changes, then the client needs to download the updated version. Since there are a large number of clients that might be disconnected, a *pull* operation is more suitable than a *push*, but it would be useful to be able to notify the client that the new profile is available. A similar trigger is required for the generation of any system-wide configuration tables (section 6.6.1).

8.3 Profile & Prototype Distribution

If the compilation takes place on a server, then the client needs to download the profile⁶; in practice, this is likely to consist of several different files such as an XML resource list, and perhaps separate files of method code. The server may also need to download profiles from other servers. Since the client must be capable of disconnected operation, the profile needs to be cached, and it should probably also be validated before

⁶If the compilation takes place on the client, then the client needs to download the prototypes.

being installed.

Protocols such as LDAP would be suitable for requesting individual resources. However, in this case, it is possible to enumerate the entire profile and it seems more appropriate to download the whole profile in a single transaction. HTTP would appear to be a suitable candidate for distribution of all the necessary objects (resource lists, code files, and prototypes). This has the following advantages:

- Secure and authenticated transport is available (SSL).
- Existing Web server technology can be used for the servers.
- Client libraries are available for most languages.
- Caching proxies can be used to provide replicated servers for performance, and full (or partial) sets of profiles can easily be replicated for resilience, either by regularly seeding the cache, or using some external protocol to synchronise the profile data.

In some cases, it might be useful for clients to be able to communicate information back to the configuration system; for example, the definitive source of information about the machine hardware is clearly the machine itself. To be able to guarantee that machine configurations have been successfully updated, it would also be useful for configuration changes to trigger an acknowledgement to some central logging facility.

8.4 Components

Developers can independently create a new component for configuring a particular subsystem. This consists of a code module for translating a profile element into the appropriate configuration files, together with a schema for the profile element, and a prototype defining default values. The component code could be stored as methods in the prototype, allowing the configuration specification to request specific versions of the component to match the other configuration data.

The component code should probably be supported by a common library providing a standard API for obtaining resources from the profile element, and general functions for manipulating the machine configuration such as those in Cfengine and SUE.

Appendix A Glossary

ACL: Access Control List; a list of users permitted to access a particular resource.

Alchemist: a Redhat Linux configuration tool (see section 5.6).

Arusha: see section 5.4.

ASOPE: aspect-oriented process engineering.

aspect-based access control: the ability to define who is allowed to control particular aspects of a machine's configuration.

aspect-oriented process engineering: an application of aspect-oriented techniques to business processes.

aspect-oriented programming: an extension of object-oriented programming which allows the independent specification of different *aspects*. Usually, the aspects affect overlapping, and widely-separated regions of the program.

Cfengine: see section 5.1.

CLAM: Caldera Loadable Administration Module; a COAS term.

class: a feature provided by many configuration tools for specifying groups of machines with similar properties; the class membership may be declared explicitly, or it may be computed automatically, based on some property of the machine such as the platform.

clone: an identical copy of some existing machine, often obtained simply by copying the image of the system disk. For pure clones to be useful, care is needed to ensure that no difference are required in the configuration; this involves the use of DHCP, for example rather than hard-wired network addresses.

COAS: Caldera Open Administration System (see section 5.5).

component: a software module for translating configuration information from an explicit *profile* into the necessary configuration files for a particular operating system (and possibly, visa-versa).

configuration management: management of version control, and construction of large software systems from component modules.

derivation: the steps taken to derive a particular *resource* value from the values specified in included *prototypes*.

disconnected operation: the ability of a machine to continue functioning when not connected to a network; for example, a laptop or a desktop during a network failure.

dynamic: a dynamic configuration tool is one which can regularly monitor the system configuration and adjust it to conform to the specification.

features: A SUE term for modules similar to *components* which implement various configuration features.

HLD: high-level description; a specification of the configuration of an entire group of related machines.

immutable: cannot be changed.

LCFG: Local ConFiGuration system (see section 5.2).

mapper: a COAS term for a *component* which can perform a two-way translation between the *profile* and the system configuration file.

method: a term from object oriented programming referring to a named operation that can be performed on an object. In the context of configuration tools, the object is usually a *component*.

mutation: the use of some algorithm, other than simply *overriding*, to compute the value of a resource by merging values included from several prototypes.

object: an LCFG *component*.

override: if a resource value in a *prototype* completely replaces any corresponding values from included prototypes, it is said to *override* them.

profile: (1) a machine-specific set of configuration parameters which completely specifies the system software to be loaded onto the machine, and how that software should be configured. (2) a SUE term for a set of *features*.

prototype: a collection of *resources* and references to other prototypes which may be included (by reference) in other specifications.

prototype-based language: a type of object-oriented programming language where new objects are created by cloning prototypes (or *exemplars*) rather than creating new instances of a class.

provenance: see *derivation*.

pull: a data transfer initiated by the client.

push: a data transfer initiated by the server.

resource: an individual configuration parameter, represented as a key-value pair.

resource map: an LCFG *profile* distributed as an NIS map.

rot: the degradation of a system configuration due to inaction; for example, failure to change network parameters to match a change in network services, or failure to install software updates necessary for security.

schema: a set of rules defining the *resources* that may

appear in a particular *prototype* (or *profile*), and their legal values.

scheme: a particular variation of a machine configuration to be used under certain circumstances; for example, a `dialup` scheme might specify some variations to the `default` scheme to be used when a machine is connected to the network over a slow dial-up connection.

spectative: code which simply examines an object (such as a prototype) without changing it.

static: a static configuration tool is capable of configuring a machine according to some specification, but is not capable of automatically maintaining the configuration if the specification later changes.

subsystems: an alternative term for an LCFG *component* (*object*).

SUE: Standardised Unix Environment (see section 5.3).

weaver: a program used to process *aspect-oriented* programs by merging code for the various different *aspects* into a single program.

Appendix B **References**

- [And94] Paul Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, 1994. Usenix. http://www.lcfg.org/doc/LISA8_Paper.pdf.
- [And97] Paul Anderson. System configuration and installation. SANS Invited Talk, 1997. <http://homepages.inf.ed.ac.uk/dcspaul/publications/Config.pdf>.
- [And01a] Paul Anderson. An LCFGng specification language. LFCS Talk, 2001. <http://homepages.inf.ed.ac.uk/dcspaul/publications/lcfgng.pdf>.
- [And01b] Paul Anderson. Some thoughts on a new resource specification language for LCFGng. Discussion Document, 2001. http://homepages.inf.ed.ac.uk/dcspaul/publications/lcfgng_thoughts.pdf.
- [AS00] Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In *Proceedings of the Atlanta Linux Showcase*, pages 363–372, Berkeley, CA, 2000. Usenix. <http://www.lcfg.org/doc/ALS2000.pdf>.
- [Bor86] Alan Borning. Classes versus prototype in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40. ACM/IEEE, November 1986. <http://www.cs.washington.edu/research/constraints/object-oriented/fjcc-86.html>.
- [Bur95] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995. <http://www.iu.hioslo.no/~mark/research/cfarticle/cfarticle.html>.
- [Bur98a] Mark Burgess. Automated system administration with feedback regulation. *Software-Practice and Experience*, 28, 1998. <http://www.iu.hioslo.no/~mark/research/feedback/feedback.html>.
- [Bur98b] Mark Burgess. Computer immunology. In *Proceedings of the 12th Large Installations Systems Administration (LISA) Conference*, page 283, Berkeley, CA, 1998. Usenix. http://www.usenix.org/publications/library/proceedings/lisa98/full_papers/burgess/burgess.pdf.
- [Cal] Caldera. COAS. Web page. <http://www.coas.org>.
- [CERa] CERN. SUE. Web page. <http://wwwinfo.cern.ch/pdp/ose/sue/doc/sue.html>.
- [CERb] CERN. SUE feature writer’s guide. Web page. http://wwwinfo.cern.ch/pdp/ose/sue/fwg/full_page.html.
- [CERc] CERN. SUE user’s guide. Web page. http://wwwinfo.cern.ch/pdp/ose/sue/users_guide/full_page.html.
- [HHW98] Andre Van Der Hoek, Dennis Heimberger, and Alexander L Wolf. Software architecture, configuration management, and configurable distributed systems: A menage a trois, 1998. <http://citeseer.nj.nec.com/22041.html>.
- [IBMa] IBM Research. Multi-dimensional separation of concerns: Software engineering using hyperspaces. Web page. <http://www.research.ibm.com/hyperspace/index.htm>.
- [IBMb] IBM Research. XML transformation: Matching & reconciliation. Web page. <http://www.research.ibm.com/hyperspace/mr/index.htm>.

- [KLM⁺97] G Kiczales, J Lamping, A Mendhekar, C Maeda, C Lopes, J Lointier, and J Irwin. Aspect oriented programming. In *Proceedings of ECOOP'97*, 1997.
[http://www.parc.xerox.com/csl/groups/sda/publications/papers/ ... Kiczales-ECOOP97/for-web.pdf](http://www.parc.xerox.com/csl/groups/sda/publications/papers/...Kiczales-ECOOP97/for-web.pdf).
- [Lal89] Wilf R Lalonde. Designing families of datatypes using exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.
- [Men] Tom Mens. A domain-independent formal approach to semantic merging of software artifacts.
<http://progwww.vub.ac.be/PoolMembers/TomMens/Research-Abstract.html>.
- [Mic97] Sun Microsystems. Preparing custom Jumpstart installations. In *Solaris 2.6 Advanced Installation Guide*, pages 71–126. Sun Microsystems, 1997.
- [OT99a] Brain Odgers and Simon Thompson. Aspect-oriented process engineering (ASOPE). In *Proceedings of the AOP Workshop at ECOOP'99*, 1999.
[http://www.parc.xerox.com/csl/projects/aop/workshops/ecoop99/ ... aop-ecoop99-proceedings.pdf](http://www.parc.xerox.com/csl/projects/aop/workshops/ecoop99/...aop-ecoop99-proceedings.pdf).
- [OT99b] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspae. In *Proceedings of the AOP Workshop at ECOOP'99*, 1999.
[http://www.parc.xerox.com/csl/projects/aop/workshops/ecoop99/ ... aop-ecoop99-proceedings.pdf](http://www.parc.xerox.com/csl/projects/aop/workshops/ecoop99/...aop-ecoop99-proceedings.pdf).
- [Par] Will Partain. The Arusha Project home page. Web page.
<http://ark.sourceforge.net/>.
- [Pen] Havoc Pennington. GConf. Web page.
<http://cvs.gnome.org/lxr/source/gconf/>.
- [Phi] Nils Philippsen. RACE. Web page.
<http://www-stud.fht-esslingen.de/race/>.
- [Red] Redhat. Alchemist. Software package (SRPM).
[http://wwwinfo.cern.ch/pdp/ose/linux/lsr/rawhide/SRPMs/SRPMs/ ... alchemist-0.11-1.src.rpm](http://wwwinfo.cern.ch/pdp/ose/linux/lsr/rawhide/SRPMs/SRPMs/...alchemist-0.11-1.src.rpm).
- [Sco01] Alastair Scobie. CS-TN-58 : Linux for syssies. Technical report, Division of Informatics, University of Edinburgh, 2001.
<http://www.dcs.ed.ac.uk/home/ajs/linux/tn58.pdf>.
- [UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Holzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 1991.
<http://www.sun.com/research/self/papers/organizing-programs.html>.
- [Xera] Xerox. Aspect oriented programming. Web page.
<http://www.parc.xerox.com/csl/projects/aop/>.
- [Xerb] Xerox. AspectJ. Web page.
<http://www.aspectj.org>.