

An lcfgNG Specification Language

- Configuration Parameters
- Existing LCFG
- lcfgNG
 - Components and syntax
 - Prototypes
 - Inheritance
 - Lists
 - Namespaces & scope

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

Configuration Parameters

- Storing configuration information centrally for all machines has a lot of benefits:
 - Persistent (if machine crashes)
 - No need for system backups
 - Can clone machines
 - Can do offline analysis and synthesis
 - ...
- The representation and structure of the information is very important to allow the configuration to be viewed at a complete system level.
- The abstract configuration parameters are turned into concrete configurations by code on the machine.

Existing LCFG

- Configuration parameters are stored as key-value pairs ("resources"):

```
machine.subsystem.key: value
```

- C preprocessor is used to provide common sets of resources which can be overridden:

```
#include "staff_machine.h"
```

- Lists are represented by convention:

```
*.install.partitions: root home
```

```
*.install.size_root: 2000
```

```
*.install.size_home: 3000
```

- "Mutation" allows modification:

```
foo.install.partitions: !/(.*)/$1 extra/
```

IcfgNG Requirements

- Better inheritance & clear derivations
- Explicit lists
- Components
- Validation
- Delegated management & Access control
- Code methods stored with data?
(This prevents version mismatches)

Components

- A scalar (optional name)

```
size: rootsize = 2000;  
size = 2000;
```

- A Record

```
Partition: root {  
  size = 2000;  
  mount = "/";  
}
```

- These are instances of the same general structure:

```
mount {  
  method: validate = "some code";  
} = "/";
```

Prototypes

- A prototype defines the structure of the type and provides default values:

```
partition: <PROTOTYPE> {  
    size { method: validate="some-code" }  
        = undef;  
    mount { method: validate="some-code" }  
        = "/";  
}
```

- Individual instances "inherit" default values from the prototype, usually "overriding" some of them:

```
partition: smallroot { size=1000; }
```

Inheritance (or "reference"?)

- By allowing an object to "inherit" other objects as well as the prototype, we can create components which can be re-used:

```
partition: usr {  
  size = 2000 ; mount = "/usr";  
}
```

```
partition(usr);  
partition(usr) : bigusr { size=3000; }
```

- And we can do "mutation":

```
partition(usr) : bigusr {  
  size { method: mutate="$0+1000"; }  
}
```

Evaluating Inheritance

- Multiple inheritance is supported by walking the inheritance tree in order and applying the "overrides".
- The resulting sequence of overrides (and mutations) is very useful for debugging. We call this the "derivation".
- There are two problems:
 - Common ancestors appear more than once in the derivation and this causes problems.
 - An object inherits from any explicit parents, as well as matching components of the enclosing object (!)

Common Ancestors

- We only want to include the first occurrence of any unique object in the derivation, because common ancestors are a problem:

```
partition: <PROTOTYPE> {  
    size = 1000; mount = "/";  
}  
partition: usr { mount = "/usr"; }  
partition: big { size = "2000"; }  
partition(big,usr): bigusr;
```

- The simple derivation is:

```
prototype, big, prototype, usr,  
prototype, bigusr
```

- This sets the final mountpoint to "/" which is not what we expect.

Inheriting from enclosing Objects

- A component may inherit from matching components of the enclosing object:

```
partition: root { size=1000; mount="/"; }
partition: big   { size=2000; }
```

```
disk: system {
    Partition(root) : systemroot;
}
disk: mydisk(system) {
    Partition(big) : systemroot;
}
```

- Because `mydisk` inherits from `system`, the component `partition` also inherits from `root`.

Inheritance Issues

- Is this a useful model?
 - Is it adequate?
 - Is it too complex?
- Is there any precedent for this model?
 - The derivation can be viewed as a list of functions to be applied to the prototype (especially considering the mutation).
- Because various logical components want to group resources in different ways, there is no obvious way to structure the hierarchy and a lot of practical inheritance looks like involving the biggest enclosing object ("machine").

Lists

- An object may have several components of the same type. These may be named or anonymous, and the order may or may not be important:

```
disk {  
    partition: root { size = 2000; }  
    partition: home { mount = "/home"; }  
}
```

```
services {  
    service = "fontserver";  
    service = "X";  
}
```

- How do we specify this in the prototype?

Inheriting Lists

- When an object inherits a list, it may want to:
 - Change items in the list
 - Add items to the list (at a specified position)
 - Remove items from the list
- If the items are named, then named items in the object should override items with the corresponding name in the inherited list.
- Can/should we do this if items are not named?
- How can we specify insertion relative to existing items. Eg.
 - Insert X into the list somewhere after Y?

Namespaces

- The C preprocessor is currently used to provide simple global variables, and we would like a similar facility.
- Eg, we would like to be able to specify the "owner" of a machine somewhere in the machine definition and have that variable available for use in any of the components.
- If we want to allow variables to be declared in arbitrary components (rather than as globals) then it isn't clear how to handle the scope.
- Allowing variables to appear as parent names seems useful but can cause circularity.

Variables

- Variable references:

```
machine: foo {  
    ...  
    owner = "paul";  
    disktype = "special";  
    ...  
}  
  
partition: personal {  
mount="/disk/home/$owner" }  
  
Disk($disktype);
```

Access Control

- We would like to be able to delegate management of specific parts of the configuration.
- This involves restricting which resources can be edited by which people.
- The central repository allows us to control who can access which objects.
- The explicit derivations allow us to compute who is responsible for the final value of any particular resource.

(this may be more than one person)

Access Control Templates

- One idea for specifying access control is to associate an ACL with each resource.
- The final machine definition could inherit a template specifying a particular configuration of ACLs for various components.

Issues

- Syntax
- Inheritance & Prototypes
 - Adequate? Too complex?
 - Any good theoretical models?
- Lists
- Namespaces
- Access control