

An LCFGng Specification Language

- Context
- LCFG Configuration principles
- The need for a specification language
- Problems
 - Prototypes & combining
 - Procedural code
 - Other issues
- Current state
- Examples

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

Configuration with LCFG

- The disk image for a machine is constructed from:
 - A software repository
 - A resource map
- The same software repository is used for all machines
- Each machine has a unique (declarative) resource map specifying the configuration of that machine
- A set of software modules on each machine continuously adjust the configuration to conform to the resource map

Advantages

- Failed machines can be rebuilt to the same state with no effort
- Machines can be cloned by copying maps
- Machines are guaranteed to have the correct configuration
- No system backups are required
- It is possible to reason about the configuration of individual machines and their relationships
- It is possible to generate configurations automatically from higher-level descriptions

The Resource Map

- The resource map contains comparatively low-level configuration parameters
- The structure is also simple:
 - Key-value pairs
 - Explicit lists and sub-groups are useful
- A typical map has the potential for about 2000 parameters in our system
- In practice, the use of defaults reduces the explicitly defined resources by a factor of about 10
- The declarative nature of the resource is a particular feature of LCFG

A Resource Map Example

- A machine with two disks having 2 and 1 partitions respectively:

```
hardware.disks           one two
hardware.one.parts      a b
hardware.one.a.size     300
hardware.one.a.mount    /
hardware.one.b.size     400
hardware.one.b.mount    /usr
hardware.two.parts      c
hardware.two.c.size     500
hardware.two.c.mount    /home
```

- (This syntax is sanitized)

A Possible HTML Representation

```
<hardware>
  <disk>
    <partitions>
      <partition size=300 mount=/>
      <partition size=400 mount=/usr>
    </partitions>
  </disk>
  <disk>
    <partitions>
      <partition size=500
mount=/home>
    </partitions>
  </disk>
```

The Problem

- We need a high-level way of describing configurations of entire sites that can be compiled down into individual resource maps:
- To make the system comprehensible
- To make it easy to create machines similar to existing ones (not necessary identical)
- To abstract certain aspects (eg. Security policy, or web-server configuration) so that they can be managed independently
- To automatically generate consistent configurations

Prototypes

- Subsets of parameters are contained in separate files which can be included in individual machine descriptions (and nested)
- A typical machine description includes about 20 of these files and several tens of individual resources
 - A file defining a prototype machine for the class
 - A file defining the hardware model
 - A file defining the software packages required
 - A file defining a special feature (wireless network)
 -
- Constructing machines from "prototypes" like this seems very natural

Prototype Combining

- Prototypes may specify different aspects of the system, but often contain overlapping resources. Combining these is not obvious:

- Priority

- Set union

- List appending (what order?)

- More complex "mutation"

- Supporting component datatypes is complicated.
- Variable namespaces are not obvious.
- The LCFGng papers & talk slides include some gory details of these problems:
<http://www.dcs.ed.ac.uk/~paul/publications/>

Procedural Code

- Arbitrary procedural code in the specification can be used to solve the combining problem and other difficulties.
- We would like to avoid this where possible because it is difficult to reason about.
- Some areas which are difficult without procedural code include:
 - "Mutation"
 - Schemes
 - Dependencies (intra - and inter-machine)
- Some uses are OK:
 - Method version matching
 - Validation

Other Issues

- Creating Configurations:

 - Authors

 - Feature authors
 - Prototype authors
 - Machine configuration authors

 - Multiple views (editors? languages?)

- Distributed management (by aspect)

- Access control (by aspect)

- Implementation issues

 - Re-compilation

 - Transport & caching

 - Security (signing)

Current State

- Language design is the main issue
 - Is the prototype model good?
 - Can we manage without (much) arbitrary procedural code?
 - What should the language look like?
 - Do we want an intermediate language?
- Very little relevant literature
 - This is not a programming language!
 - Existing configuration systems are not addressing these problems
- Intent to implement a framework into which we can place prototype compilers
 - Including schema for the resource maps

Example: 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";  
} = "/";
```

Example: 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; }
```

Example: Inheritance

- 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"; }  
}
```

Example: 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 (!)

Example: 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.